A SET EXPRESSION BASED INHERITANCE SYSTEM

by

Ido Dagan and Alon Itai

Technical Report #557

May 1989
A Set Expression Based Inheritance System

Ido Dagan
Alon Itai
Computer Science Department
Technion – Israel Institute of Technology
Haifa, Israel

ABSTRACT

This paper describes a new formalism for inheritance systems, based on the formal semantics of set expressions. Using the formalism, it is possible to define new semantic classes by arbitrary set expressions operating on previously defined classes. We present an efficient algorithm which follows these definitions to deduce the properties implied by the inheritance network, i.e., the properties of the classes containing a given element. The application which motivated the development of the formalism, namely semantic disambiguation of natural language, is also described. We conclude by raising several open problems concerning more advanced topics, such as IS-NOT-A links and conflicting properties.

Bitnet Address: itai@TECHSEL
1. Introduction

Inheritance systems have been investigated extensively in AI and in object oriented programming. This paper describes a new formalism for inheritance, based on the semantics of set expressions. Though the new formalism was motivated by a particular problem (semantic disambiguation in Natural Languages), we hope that it will be examined also for other usages in these domains.

A basic scheme of inheritance involves a domain of elements and an acyclic network of semantic classes. Various properties are attached to semantic classes, and elements inherit their properties by following IS-A links. A common semantics for the links of the network considers an IS-A link which connects an element with a class as set membership relation. An IS-A link which connects two classes is considered a set inclusion relation. These relations are often represented in first order logic ([Hayes 1979], [Hayes and Hendrix 1981], [Brewka 1987]). Using set theory terminology, a class is the union of its descendants.

Some properties of inheritance vary among different systems. The network might have a tree structure or contain multiple inheritance (several links emanating from a class). It might also have IS-NOT-A links, denoting that an element does not belong to class, or that two classes are disjoint. These links are used to block paths of inheritance. The NETL ([Fahlman 1979]) and TMOIS ([Touretzky 1986]) systems use the two types of links to denote exceptions.

A common property of the above networks is that an inheritance path follows increasingly larger classes. Consequently, an element should be connected directly to the most specific classes from which it can inherit properties. This fact may cause problems when maintaining a network containing many elements. Consider for example a variation of the famous network describing Clyde ([Touretzky et. al 1987], [Brewka 1987] and [Sandewall 1986]), as illustrated in Figure 1:

Figure 1

Figure 1(a) describes a network in which properties are inherited from the classes Elephant and Royal. Therefore, Clyde, which is a royal elephant, is linked to the two basic classes. Jumbo, who is just a circus elephant, does not inherit any royal properties. Now, suppose we want to update the
information that royal elephants are not grey. Since we would like elements to inherit properties also from the class RoyalElephant, we must link Clyde directly to the new class, to get the network of Figure 1(b). When new elephants are added to the network, we must remember that they should be connected to RoyalElephant if they are indeed royal, and to Elephant otherwise. Of course, the complexity increases if there are many Clydes and Jumbos, or when we would like to inherit special properties for a RoyalAfriicanElephants versus RoyalIndianElephant.

In general, there are two problems that arise because elements are linked to the most specific classes. First, if we view the classes pointed directly by an element as its basic semantic properties, we get a very messy and incoherent system of such properties. It is natural to consider Elephant, Royal and African as basic properties, but not specific combinations of those which vary from one element to another. (To increase coherency, one may think of maintaining some redundant links, such as from Clyde to both Elephant and RoyalElephant. However, this might lead to losing the effect of inheritance and also might cost a lot of space.) The second problem is that when a new specific class is added to the network, the links of all its elements should be modified. As the number of elements is usually much greater than the number of classes, the cost of the modification is large.

The new formalism we present overcomes these problems, as inheritance is not limited to follow only increasingly larger classes. For example, the class RoyalElephant can be defined as the intersection of Royal and Elephant, and the inheritance path will follow this definition for all the elements that belong to the two classes. When adding this new definition, it is not necessary to change any of the links of the elements. In a certain respect, the new formalism generalizes the traditional one, as a class may relate to its descendants using an arbitrary set expression, instead of always being their union.

Section 2 of the paper defines the inheritance system. Section 3 answers the basic computational question: how to find all the classes containing a given element and from which classes should it inherit its properties. We describe an efficient algorithm for this purpose. The complexity of the algorithm is a function of the number of classes containing the element, while the irrelevant parts of the network are not scanned. Section 4 describes the application of semantic disambiguation in natural language processing, which motivated the development of our inheritance system. Section 5 concludes by raising several open problems concerning more advanced topics.

2. Formalization of The Inheritance System

An inheritance system consists of a domain of elements and sets of elements called classes. The intended semantics for the network is that an element may inherit properties from all the classes to which it belongs. For each element $e$ in the domain, the singleton set, $\{e\}$, is defined implicitly. The remaining classes are defined by set expressions, using the operations union, intersection and difference, with their usual semantics. (The use of the completion operation is briefly discussed in
Section 5). The definitions are assumed to be acyclic, as usually assumed for inheritance networks. For example, a class $S_4$ may be defined as follows (see Figure 2):

$$S_4 = \{e_1\} \cup \{e_7\} \cup ((S_1 \cap S_2) - S_3).$$

Graphically, the inheritance system is represented as an acyclic network (directed acyclic graph). Each node represents a class, to which its defining expression is attached. An edge $(S_i, S_j)$ means that $S_i$ participates in the definition of $S_j$. $S_j$ is said to be a parent of $S_i$ and $S_i$ a child of $S_j$. An example of a network is given in Figure 2.

![Figure 2](image)

Notice that a traditional inheritance network, in which every class is a subclass of its parents, is a special case of our set expression formalism, using only union operations.

3. FCS — the Find Containing Sets algorithm

To use an inheritance network of any kind, we first need an algorithm that given an element $e$ finds the classes from which $e$ may inherit its properties (at this stage we ignore problems of conflicting properties, and are interested in an algorithm which finds all the candidates for inheritance). In traditional inheritance systems, such an algorithm is relatively trivial. For a tree-structured network it follows the path from the element to the root. In multiple inheritance networks, instead of a single path it is necessary to perform some search algorithm (as BFS) to find all the nodes connected to the element. If the network contains also IS-NOT-A links (a nonmonotonic inheritance system), these links prune the search.

For the set expression based system, the algorithm should find all the classes containing the element. In this case, a search which just follows the edges of the network is not sufficient. Because intersection and difference operations are involved, the element $e$ may belong to a certain class $S_i$, but not to its parent $S_j$. (For example, if $S_1 = \{e_1, e_2\}$, $S_2 = \{e_2, e_3\}$ and $S_3 = S_1 \cap S_2$ then $e_1$ belongs to $S_1$ but not to its parent $S_3$.) In general, in order to decide whether $e$ belongs to $S_j$, it is necessary to know the answer for each of $S_j$'s children.
When thinking of an algorithm which fulfills the last requirement, we have to notice two potential problems. The first is that we have to answer the membership question also for classes that do not have any path leading to them from the element in question $e$. This is because these classes appear in the defining expressions of other classes, for which a path from $e$ does exist. However, it is extremely undesirable to scan the whole network, which is much larger than the subnetwork connected to $e$. Moreover, it may be unnecessary to scan the entire subnetwork connected to $e$. If a certain path leads from $e$ to a class $S_j$ that does not contain $e$ (because some intersection or difference operation), we would like to prune the search at that point and avoid searching $S_j$'s ancestors. (This pruning is similar to the termination of inheritance paths by IS-NOT-A links in nonmonotonic networks.)

To overcome these problems, the FCS algorithm uses a topological order for the internal nodes of the network. A number is attached to each node, such that the number of a node is greater than the numbers of all its children. Topological ordering is discussed in the literature ([Knuth 1973], [Aho, Hopcroft and Ullman 1983]) and may be performed in $O(|E|)$ operations. The ordering is done only once, before the executions of the FCS algorithm, and should be modified only when the structure of the network is modified. Indeed, modifying the topological ordering in dynamic graphs in less than $O(|E|)$ is known as an open problem. We have assumed that this operation occurs less frequently than the executions of the FCS algorithm.

The FCS algorithm itself (see below) is a modified version of BFS. Its input is an element $e$, and the output is the list, $CLOSED$, of all classes containing $e$. (In BFS $CLOSED$ contains all the nodes accessible from the input node.) As in other search algorithms, we maintain a temporary list, $OPEN$, consisting of all classes which the algorithm checks to see whether they belong to $CLOSED$. To each class $S$ in the network we associate a variable $topo(S)$ containing the topological number—the number given to the class by the topological ordering. $CLOSED$ is initialized to $\{e\}$, and $OPEN$ to the list of all parents of $\{e\}$.

We now describe the FCS algorithm in a top-down fashion, further details follow.

\[
\text{while } OPEN \text{ is not empty do}
\]
\[
\begin{align*}
(1) & \quad \text{Remove the least member, } S, \text{ from } OPEN, \text{ (that with least } topo). \\
(2) & \quad \text{Use the defining expression of } S \text{ to check whether } e \in S. \\
(3) & \quad \text{If } e \in S \text{ then} \\
& \quad (a) \quad \text{Add } S \text{ to } CLOSED. \\
& \quad (b) \quad \text{Add the parents of } S \text{ to } OPEN.
\end{align*}
\]
Further Details:

(1) \textit{OPEN} is maintained as a heap [Aho, Hopcroft and Ullman 1983], ordered by \textit{topo}.

(2) The defining expression of \textit{S} is evaluated like a boolean expression in the obvious way, (if, for example, \textit{S} = \textit{S}_i \cup \textit{S}_j \text{ then } \epsilon \in \textit{S} if and only if } \epsilon \in \textit{S}_i \text{ or } \epsilon \in \textit{S}_j ; \text{ if } \textit{S} = \textit{S}_i \cap \textit{S}_j \text{ then } \epsilon \in \textit{S} if and only if both } \epsilon \in \textit{S}_i \text{ and } \epsilon \in \textit{S}_j \text{ ). We shall show that when } \textit{S} \text{ is evaluated then all its children that contain } \epsilon \text{ are already in \textit{CLOSED}. Thus if a child } \textit{S}_i \text{ is not already in \textit{CLOSED} then } \epsilon \not\in \textit{S}_i.

(3) To facilitate checking if a child \textit{S}_i \in \textit{CLOSED}, we associate with each class \textit{S} a boolean variable \textit{in-closed}(\textit{S}), indicating whether } \textit{S} \in \textit{CLOSED}. At the beginning of each invocation, for all \textit{S} \notin \{\epsilon\}, \textit{in-closed}(\textit{S}) = \textit{false}, and it is set to \textit{true} when \textit{S} is added to \textit{CLOSED}, (in step (3a) of the algorithm).

(4) We wish to add each class to \textit{OPEN} at most once (a class, \textit{S}, might have several children in \textit{CLOSED} each attempting to add \textit{S} to \textit{OPEN} – such multiple copies might slow-down our algorithm). Therefore, we associate with each class \textit{S} another boolean variable, \textit{in-open}(\textit{S}), initialized to \textit{false}. In step (3b) we add a parent \textit{P} only if \textit{in-open}(\textit{P}) = \textit{false}. If \textit{P} is indeed added to \textit{OPEN} \textit{in-open}(\textit{P}) is set to \textit{true}. When \textit{S} is removed from \textit{OPEN} in step (1) then \textit{in-open}(\textit{S}) is set to \textit{false}.

(5) The variables \textit{in-open}(\textit{S}) and \textit{in-closed}(\textit{S}), associated with the classes, are static, i.e., they are retained between invocations. Before each invocation \textit{OPEN} and \textit{CLOSED} are empty, so for all \textit{S} \textit{in-open}(\textit{S}) and \textit{in-closed}(\textit{S}) should receive the value \textit{false}. To achieve this, before the first invocation, all such variables are all set to \textit{false}. At the end of each invocation, \textit{OPEN} is empty and, therefore, for each class \textit{S}, \textit{in-open}(\textit{S}) = \textit{false}. However, for all classes \textit{S} \in \textit{CLOSED}, \textit{in-closed}(\textit{S}) = \textit{true}. Therefore, at the end of the invocation \textit{CLOSED} is traversed and for each class \textit{S} \in \textit{CLOSED} \textit{in-closed}(\textit{S}) is set to \textit{false}. The reinitialization time is proportional to the execution time of the invocation and, therefore, does not increase the time complexity of the algorithm.

Proof of correctness:

\textbf{Lemma:} A class \textit{S} is placed in \textit{CLOSED} iff } \epsilon \in \textit{S}.

\textbf{Proof:} We add the following claim:

\textbf{Claim:} When \textit{S} is evaluated any class \textit{S}' with smaller topological number containing } \epsilon \text{ already belongs to \textit{CLOSED}.}

The Lemma and the Claim are proved together by induction on \textit{topo}(\textit{S}).

\textbf{Basis:} \textit{topo}(\textit{S})=1: \textit{S} is a singleton, the Lemma follows from the algorithm and the Claim is vacuous.

\textbf{Induction Step:} There are two cases depending on whether \textit{S} was ever placed in \textit{OPEN}.

\textbf{Case 1:} \textit{S} was placed in \textit{OPEN}.
After $S$ is evaluated only classes whose topological number is greater than $\text{topo}(S)$ may be placed in $OPEN$. Thus if $S'$ is ever placed in $OPEN$ it must have been placed there before $S$ was evaluated. Since $\text{topo}(S') < \text{topo}(S)$, if $S'$ is ever placed in $OPEN$ it is also evaluated before $S$. Thus if $S'$ is placed in $CLOSED$ it must have been added before $S$ was evaluated. Using the induction hypothesis for $S'$, $S' \in CLOSED$ iff $e \in S'$ – proving the Claim. If the Claim holds for $S$ then the algorithm correctly decides which children of $S$ contain $e$ (their topological number is less than $\text{topo}(S)$) and consequently correctly evaluates $S$.

**Case 2:** $S$ was never placed in $OPEN$.

In this case, $S$ is never placed in $CLOSED$ and we have to show that $e \notin S$. Since $S$ was never placed in $OPEN$, none of its children was placed in $CLOSED$. Since their topological number is less than $\text{topo}(S)$, by the induction hypothesis $e$ does not belong to any of them. Thus, $e \notin S$. □

An Implementation Note:

Every class expression can be written as $S = (\{e_1\} \cup \{e_2\} \cup \cdots \cup \{e_k\}) \cup \text{exp}'$. The (possibly empty) union of the singletons is called the *direct membership* part and $\text{exp}'$ the *remainder*. Note that $S$ contains each $e_i$ regardless of the remainder. In this case, we say that $S$ *directly contains* any such $e_i$. For some classes the direct membership part of the expression is long. The following implementation allows us to avoid reading (and evaluating) this part.

For each element $e$ we maintain a list of pointers to classes directly containing $e$. (These pointers correspond to the IS-A links emanating from the elements in conventional inheritance networks.) At the beginning of an invocation, all classes $S$ pointed at by a link from $\{e\}$ are placed in $OPEN$ and a special boolean variable, $\text{directly-contains}(S)$ is set to true. This variable is consulted before the class is evaluated and is turned off when the class is placed in $CLOSED$.

This mechanism implies that the cost of evaluating the defining expressions depends only on the length of the remainder ($\text{exp}'$).

The complexity of FCS:

The complexity of FCS is a function of the following arguments:

- $m$: The number of classes containing $e$, (in $CLOSED$ 1 at the end of the execution);
- $e_m$: The number of edges leaving the above $m$ nodes;
- $n$: The $m$ classes containing $e$ and their parents, (the number of classes inserted to $OPEN$);
- $x$: The maximal length of a remainder ($\text{exp}'$) of a defining expression.

To implement $OPEN$ we use a heap [Aho, Hopcroft and Ullman 1983]. Using $in-open(S)$, each class is inserted to the heap only once, so the cost of insertion is $O(1)$. Removing a class from $OPEN$ costs $O(\log n)$. The overall complexity is:

$$O(n \times + n \log n + e_m),$$

where $n \times$ is the cost of evaluating the $n$ defining expressions, $n \log n$ the cost of $n$ removals from
The heap and $e_m$ is the number of insertions to OPEN.

It is reasonable to assume that the average length of the defining expressions is a small constant, as the expressions are used to define meaningful semantic classes. Using this assumption, we get a complexity of $O(n \log n + e_m)$. This complexity is not much larger than that of the BFS used in traditional inheritance system ($O(e_m)$), and when $n \log n < e_m$, the complexity is the same. The conclusion is that we pay very little for generalizing the inheritance system using set expressions, and sometimes we even don't pay at all.

4. An Example Application

Natural language processing systems often use inheritance systems for semantic disambiguation. Various semantic constraints are stated in terms of semantic classes. When analyzing a word, the system looks for the semantic classes to which this word belongs and uses the constraints involving these classes. For example, when treating the word "green", the system may use a constraint which states that an adjective of the class "color" can modify a noun of the class "physical object". In terms of an inheritance system, the domain of elements is the words in the lexicon, the semantic classes represent sets of words and the inherited properties are the semantic constraints.

In many systems, the semantic classes are organized hierarchically, by the traditional semantics of inclusion relations. Figure 3 gives a portion of a semantic hierarchy for nouns used by the Japanese Government Project for Machine Translation ([Nagao et al. 1985]).

![Figure 3](image)

The classes defined in the hierarchy are used in lexical disambiguation rules that select the correct translation for ambiguous verbs. For example, a rule for a certain Japanese verb states that it should be translated to "take place" when its subject is a Social-phenomenon. Other types of subjects lead to other translations, such as "arise", "produce" etc. To use the rules, the words of the lexicon are classified into the appropriate semantic classes, and thus they inherit the various constraints. A similar system appears also in the ASCOF French-to-German translation system described in [Biewer
Semantic constraints are used also for syntactic disambiguation, to select the most appropriate parse tree of a sentence when several alternatives are produced by the parser. In the system of [Lang and Hirschman 1988], cooccurrence patterns that appear in a corpus are generalized for semantic classes. For example, the semantically valid noun compound "oil-pressure" is generalized to a constraint which states the validity of any compound of the form Fluid–ScalarQuantity. This constraint is later used to approve parses containing such compounds. The motivation for this paper arose from the work described in [Dagan and Itai 1989], which unifies the use of the constraints for both lexical and syntactic ambiguities.

The application of semantic disambiguation is a good example of the motivation for using set expressions. The domain of elements is very large, and the creation of the network is a laborious and continuous process, as new constraints relating to specific classes are required. It is most natural to keep a list of the basic classes to which each word belongs, while other classes, used for specific constraints, are defined later using the basic classes. For example, consider the basic class Aeroplane, whose members share many constraints, such as being valid subjects for "land", "fly" etc. Another basic class may be Weaponry, which appear in other constraints. Now, suppose we like to add a constraint which states that members of the class FighterCraft are valid subjects for the verb "bomb". In a traditional network, we will have to denote this class for all the relevant lexicon entries, and then link FighterCraft as a child of both Aeroplane and Weaponry. On the other hand, in the set expression based network all we have to do is to define the new class as the intersection of the two basic classes, and the inheritance will be performed automatically for all the relevant elements.

5. Discussion and Open Problems

The inheritance system presented in this paper is based on the assumption that an element should inherit its properties from all the semantic classes to which it belongs. Using set expressions, we have reduced the inheritance problem to that of finding all the sets which contain a given element. Our system, as well as traditional inheritance systems using the restricted subclass based semantics, use the closed world assumption. A class $S$ does not contain an element $e$ only if we cannot deduce that $e \in S$. However, other formalisms with different semantics have also been developed. For example, Touretzky (in [Touretzky 1986], [Touretzky et. al 1987]) uses defeasible links, with the meaning that members of one class are typically members of another class (a subclass relation does not necessarily hold). The comparison of this systems with ours, and the possibility to enhance the expressive power of the set expression formalism to represent similar links are still open questions.

The following paragraphs illustrate some open problems, with the purpose of further developing the set expression formalism and combining it with some issues discussed in the literature.
IS-NOT-A links:

A natural way to represent IS-NOT-A links is using difference operations. For example, the situation in Figure 4 can be represented in our system by defining GreyThing as Elephant - RoyalElephant. This captures the intuitive meaning of Figure 4.

However, due to the closed world assumption, the inherent ambiguity of situations such as the famous "Nixon diamond" cannot be represented in our system.

Conflicting properties:

Conflicts arise when several classes to which an element belongs disagree upon the value of a certain property. A common and intuitive rule for resolving conflicts is that the properties of a class override the properties of its superclasses (this intuitive rule is called the specialization principle in [Brewka 1987]). In traditional tree structured networks this rule resolves all the ambiguities, as properties are inherited from classes along a single path. In multiple inheritance systems, not all conflicts can thus be resolved. In such cases, a skeptical reasoner cannot deduce the value of the property. In our set expression based system the situation is even more complicated, as classes do not necessarily contain all their descendants. (The subclass relation $S_i \subseteq S_j$ holds when $S_j = S_i \cup S_k$, $S_i = S_j \cap S_k$ or $S_i = S_j - S_k$.) When a path leading from $S_i$ to $S_j$ involves operations of different types it may be impossible to decide whether a subclass relation holds. Therefore, the use of the specialization principle is more limited.

Efficient implementation of the complement operation:

The complement operation can be implemented in a straightforward manner using a special class $U$ containing all the elements of the domain. The complement of a class $S$ is then defined as $U - S$. The problem with this solution is that on every invocation of the FCS algorithm, all the parents of $U$ are inserted to OPEN, since every element is a member of $U$. The overhead thus added to each invocation is proportional to the number of classes whose
REFERENCES

[Aho, Hopcroft and Ullman 1983]

[Ait-Kaci and Lincoln 1988]

[Ait-Kaci and Nasr 1986]

[Biewer et. al 1985]

[Brewka 1987]

[Dagan and Itai 1989]

[Fahlman 1979]

[Hayes 1979]

[Hayes and Hendrix 1981]

[Knuth 1973]
[Lang and Hirschman 1988]

[Nagao et. al 1985]

[Sandewall 1986]

[Touretzky 1986]

[Touretzky et. al 1987]