Using Classification as a Programming Language

Chris Mellish*and Ehud Reiter†
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
GREAT BRITAIN

Abstract

Our experience in the IDAS natural language generation project has shown us that IDAS's KL-ONE-like classifier, originally built solely to hold a domain knowledge base, could also be used to perform many of the computations required by a natural-language generation system; in fact it seems possible to use the classifier to encode and execute arbitrary programs. We discuss IDAS's classification system and how it differs from other such systems (perhaps most notably in the presence of 'template' constructs that enable recursion to be encoded); give examples of program fragments encoded in the classification system; and compare the classification approach to other AI programming paradigms (e.g., logic programming).

1 Introduction

This paper investigates a new approach to programming that is based on controlling a KL-ONE-type *classifier* [Brachman and Schmolze, 1985]. In this approach, 'inputs' are classes that are given to the classifier, 'programs' are the existing classes in the taxonomy maintained by the classifier, and 'outputs' are formed by classifying the input class into the program taxonomy, and reading off some of the attributes that the class inherits by virtue of its position in the taxonomy. We have used this approach to implement [Reiter and Mellish, 1992] most of the natural-language generation component of the IDAS on-line documentation system [Reiter *et al.*, 1992], and have noticed that it seems possible to use classification programming to implement many other kinds of programs as well. We mean this not just in the sense that our classification programming system is Turing-equivalent (which it is), but in the more important sense that it seems like a natural way to express certain kinds of computations; this has proven to be the case in IDAS, at any rate.

This work is perhaps best considered as an attempt to explore a poorly investigated portion of the space of possible programming languages; it is not yet clear whether this exploration will lead to a useful general-purpose programming language. Much of our work is inspired by the development of logic programming, which arose out of the realisation that controlled theorem-proving could be the basis of a programming language; our hope is to show that controlled classification can similarly be used to create a programming language. We would like to emphasize, incidentally, that although this paper presents work in progress, we have implemented the I1 system we discuss in this paper, and have used it to build a substantial portion of a documentation generation system; the system we are describing here is not simply a paper design. On the other hand, I1 was not developed originally as a general-purpose programming language and it is clear that it is deficient in a number of respects (e.g. syntax, existence of optimising compilers and debuggers) that prevent it from being (at present) a practical general programming language.

We will compare classification programming system mainly to logic programming, both because that is the programming paradigm we are most familiar with, and also because our experience to date with classification programming has suggested that such programming 'feels like' programming Prolog; at the most basic level, for example, both classification and logic programming are very declarative approaches to building programs. Classification programming also, however, contains characteristics that are more reminiscent of object-oriented programming and production systems, and this may make it useful in applications that are difficult to develop in Prolog.

Section 2 of this paper briefly describes what a classifier does. Section 3 introduces the I1 system and gives a simplified formal syntax and semantics for its core. Section 4 describes how I1 can be used to represent and execute programs, and gives examples of a simple Prolog-style `append` function and a rule from the IDAS surface

*E-mail `C.Mellish@ed.ac.uk`.
†E-mail `E.Reiter@ed.ac.uk`. Ehud Reiter is supported by SERC grant GR/F/36750

realisation grammar. Section 5 compares classification to related approaches, and section 6 presents some concluding comments about classification programming and its future prospects.

2 Classification

A classification system computes subsumption relationships between domain entities, and records these subsumption relationships by maintaining a classification taxonomy. For example, a classifier could determine that the class Bachelor, defined as [Person with attributes {sex:Male, age-status:Adult, marital-status:Unmarried}], was subsumed by the class Man, defined as [Person with attribute {sex:Male, age-status:Adult}]; and hence that Bachelor should go beneath Man in the taxonomy. Space does not permit a detailed discussion of basic classification technology and terminology; the interested reader may wish to read, for example, [Brachman and Schmolze, 1985] or the chapters in [Sowa, 1991]. We will assume in this paper that the reader is familiar with basic classification terminology, such as the terms *value restrictions* and *primitive class*.

A classification-based knowledge representation system usually contains an assertional component or 'A-box' as well as the subsumption reasoner ('T-box'). One of the simplest A-box's, and the one that will be of primary interest here, is a default inheritance system, where attributes are inherited from superclass to subclass in the subsumption taxonomy, unless they are overridden by an attribute value specified in a more specific class. These inherited attributes are typically distinguished from the attributes the classifier examines when performing subsumption calculations, since current-day classification algorithms cannot handle default attributes.

A classification-based KR system performs inferences by combining the abilities of its T-box and A-box. For example, suppose the system was informed that

- the class Adult-Person, defined as [Person with {age-status:Adult}], has the default attribute has-children:True;
- the class Unmarried-Adult-Person, defined as [Person with {age-status:Adult, marital-status:Unmarried}], had the default attribute has-children:False.
- John is a Person with attributes {sex:Male, age-status:Adult, marital-status:Unmarried}

John is subsumed by both Adult-Person and Unmarried-Adult-Person, and Unmarried-Adult-Person is subsumed by Adult-Person. If a query is issued for the value of has-children for John, the system will prefer the default attached to Unmarried-Adult-Person because this class is more specific than (i.e., is subsumed by) Adult-Person; hence, the system will conclude that John has the value False for the role has-children.

The above illustrates the simplest use of classification to make inferences and computations; the classifier uses its knowledge about the individual class John to place that class in the taxonomy, and then uses the class's computed taxonomy position to make inferences about its

attributes. Writing general-purpose programs requires adding facilities for procedure calls and data structuring; the techniques used by I1 to do this are based on adding *template* and *reference* constructs that allow new classes to be dynamically created, classified, and queried (for attribute values) while the system is responding to an attribute value query. These constructs are described in section 3, and their use for programming is discussed in section 4.

3 I1

I1 is the knowledge representation system used in IDAS, and includes:

- an automatic classifier that supports value restriction.[1]
- a default-inheritance system that gives precedence to defaults from subsumed (more specific) classes.
- various support tools, such as a graphical browser and editor.

I1 is probably not the best possible classification programming language, and if we were to start over we would undoubtably build a somewhat different system. But I1 has been the basis of our experiments with classification programming, and hence we will use it in this paper.

More formally, the open-class symbols used in specifying an I1 knowledge base are of three kinds. We will use $c \in C$ to denote a concept (class) name, $ra \in Ra$ an *assertional* role name and $rd \in Rd$ a *definitional* role name. Definitional roles are those seen by the classifier, whereas assertional roles are those whose values are inherited by the default mechanism. Definitional roles are not processed by the default-inheritance system (although they are inherited in a non-default manner), and assertional roles are ignored during classification.

For specifications of role values, we will use $va \in Va$ to denote an assertional role value specification and $vd \in Vd$ a definitional role value specification[2]:

$vd \rightarrow c$

$va \rightarrow c \mid < r_1 \ldots r_n > \mid c$ with $\{rd_1 : v_1, \ldots, rd_m : v_m\}$

We will refer to the second kind of va specification as a *reference*, and the third kind as a *template*. References specify the value of a role in terms of a KL-ONE-like role chain, while templates specify a parametrized class definition as a role value.

An I1 knowledge base is specified by a set of sentences ϕ as follows:

$\phi \rightarrow$

$c \Rightarrow \{ra_1{:}va_1, \ldots, ra_n{:}va_n\} \mid$

$c \equiv \{c_1, \ldots, c_m\}$ with $\{rd_1{:}vd_1, \ldots, rd_n{:}vd_n\}$

There are some global conditions that must be placed on the set of sentences that can occur in a knowledge base. For instance, each concept symbol c should only occur in one statement of the form $c \equiv \ldots$. For convenience we will define $R = Ra \cup Rd$, $V = Va \cup Vd$ and use $r \in R$ and $v \in V$.

The following example illustrates some templates with embedded reference role chains:

```
Open-door ≡ {Open} with {actor:Animate, actee:Door}
Open-door ⇒ {
 decomposition: Sequence with {
   1: Grasp with {actor:<actor>, actee:<handle actee>},
   2: Turn with {actor:<actor>, actee:<handle actee>},
   3: Pull with {actor:<actor>, actee:<handle actee>}}}
```

The $\equiv$ statement defines Open-door to be the class of all Open entities whose actor role has a filler subsumed by Animate, and whose actee role has a filler subsumed by Door. These are definitional value restrictions. The $\Rightarrow$ statement describes the value for an assertional role (decomposition) of Open-door. The value is specified by a template, which provides the name of a class subsuming the value (Sequence) and a set of specifications for role values of this class (here, the roles 1, 2 and 3). These role values are themselves specified by templates. Each of these templates defines a class whose ancestor is an action (Grasp, Turn, Pull) that has the same actor as the Open-Door action and that has an actee that is the filler of the handle role of the actee of the Open-Door action.

For example, if Open-12 was defined as an Open action with role fillers actor:Sam and actee:Door-6:

```
Open-12 ≡ {Open} with {actor:Sam, actee:Door-6}
```

then Open-12 would be classified beneath Open-Door by the classifier on the basis of its actor and actee values. If an inquiry was issued for the value of decomposition for Open-12, the above definition from Open-Door would be inherited, and, if Door-6 had Handle-6 as the filler of its handle role, the three templates in the Sequence would be expanded into three actions, (Grasp-12 Turn-12 Pull-12), each of which had an actor of Sam and an actee of Handle-6; these three subactions would then themselves be classified into the taxonomy.

We will now specify a simple formal semantics for the I1 syntax we have presented. This is purely for concreteness – apart, perhaps, from the case of templates (the last clause for *val* below), we take a standard approach, as exemplified, for instance, by [Levesque and Brachman, 1985]. This account fails to deal with the default inheritance aspects of I1, and must be read making the assumption that any class only inherits a single specification for any assertional role from its ancestors. An *interpretation* $\mathcal{I}$ is a pair $< D, F >$, where D is a set and F provides for each $c \in C$, a set $c^{\mathcal{I}} \subseteq D$ and for each $r \in R$, a function $r^{\mathcal{I}} : D \rightarrow D$. An interpretation

[1] The I1 classifier also supports other class-definition operators (e.g., a very limited form of role differentiation), but these will not be discussed in this paper.

[2] Note that for compactness this paper uses a different I1 syntax from that used in the implementation.

$\mathcal{I}$ satisfies a given sentence ϕ, $sat(\phi, \mathcal{I})$ in the following cases:

$sat(c \Rightarrow \{\ldots, r_i : v_i, \ldots\}, \mathcal{I})$ iff
  for each $x \in c^{\mathcal{I}}$, for each i,
    $r_i^{\mathcal{I}}(x) \in val(v_i, x, \mathcal{I})$
$sat(c \equiv \{\ldots, c_i, \ldots\}$ with $\{\ldots, r_j{:}v_j, \ldots\}, \mathcal{I})$ iff
  $x \in c^{\mathcal{I}}$ exactly when
    for all i, $x \in c_i^{\mathcal{I}}$ and
    for all j, $r_j^{\mathcal{I}}(x) \in val(v_j, x, \mathcal{I})$

where

$val(c, x, \mathcal{I}) = c^{\mathcal{I}}$
$val(<>, x, \mathcal{I}) = \{x\}$
$val(< r_1 \ldots r_n >, x, \mathcal{I}) =$
  $\{r_1^{\mathcal{I}}(y) \mid y \in val(< r_2, \ldots r_n >, x, \mathcal{I})\}$
$val(c$ with $\{\ldots, r_i : v_i, \ldots\}, x, \mathcal{I}) =$
  $\{y \in c^{\mathcal{I}} \mid$ for all i, $r_i^{\mathcal{I}}(y) \in val(v_i, x, \mathcal{I})\}$

An interpretation $\mathcal{I}$ satisfies a set of sentences KB iff it satisfies each sentence in KB.

Once a knowledge base has been established, it is necessary to be able to present queries to determine its consequences. A query can be expressed in the form:

$$(< r_1 r_2 \ldots r_n > \text{ of } c) \sqsubseteq c_1$$

("is the value of r_1 of r_2 of ... r_n of every instance of c also an instance of c_1?"). A knowledge base KB supports (a "yes" answer to) a query q, $KB \models q$, as follows:

$KB \models (< r_1, r_2 \ldots r_n > \text{ of } c) \sqsubseteq c_1$ iff
  for all $\mathcal{I}$ such that $sat(KB, \mathcal{I})$,
  for all $x \in c^{\mathcal{I}}$,
    $val(< r_1, r_2 \ldots r_n >, x, \mathcal{I}) \subseteq c_1^{\mathcal{I}}$

In practice, in I1, queries can only be expressed for classes c that have been declared as individuals. Role values are computed lazily (but with caching) as they are needed to answer a query.

4 Classification Programming

I1 was designed to represent a fairly conventional knowledge base of entities and actions, but we discovered that it could also be used to represent and execute general-purpose programs. To do this,

1. Classes in the taxonomy are regarded as rules, whose conditions are specified in the definitional roles (value restrictions) and whose conclusions are expressed in the assertional roles.

2. Queries are represented as new classes that must be classified.

3. The classifier acts as a pattern-matcher that selects a rule which fits the query.

The 'running' of a classification program is initiated by a request for the value of an assertional role for a class that has already been classified. The interesting case arises when this role value is specified by a template. In this case, the class specified by the template must be created and classified. If some role value for the new class is needed to satisfy the original request, then this may lead to more new classes being created and classified. In this way, the classification programmer has access to recursion.

Classes can also be used to represent data structures, with roles being used to represent and access the data structure's fields. Such data structures can be dynamically constructed by templates during the 'execution' of a classification program, and their components can be dynamically accessed with references. It is also possible to add special-purpose constructs for common structures such as lists, which may improve efficiency; this is similar to the approach taken by Prolog, which has a special syntax and compiler optimisations for constructing and accessing lists. It is important to note that programs and data in I1 are represented with the same kind of structure, namely classes, and are subject to the same operations, namely classification and role inheritance; in languages such as Prolog, in contrast, a real distinction is made between programs and data and the operations that make sense on each.

4.1 Converting Prolog Programs into Classification Programs

Simple Prolog programs can be fairly mechanically converted into I1 programs, provided that the clauses in the program distinguish between input and output arguments, and no deep backtracking is performed. These restrictions of course remove much of the power of Prolog, but we illustrate the process here in an attempt to give readers used to Prolog more of an intuition for classification programs.

To generate an equivalent I1 program for a simple Prolog program, the following must be performed:

- An I1 primitive class is generated for each Prolog predicate[3].

- A role is defined for each argument (input and output) of a predicate (argument names must be introduced, since classification requires keyword parameters instead of Prolog's positional arguments).

- A class is defined for each clause, which

[3]Primitiveness can be simulated in the simplified syntax by using a definition of the form:

C $\equiv$ ... with {c-q:True}

where c-q is a role not mentioned elsewhere in the knowledge base.

- Includes an appropriate value restriction for every constrained input argument.
- Includes a template for each 'call' of a predicate in the clause's right-hand-side, with embedded templates used to create any necessary complex terms (data structures).
- Includes templates and references that relate the output of the LHS predicate to the inputs of the LHS predicates and the outputs of the RHS predicates.

Such a program is queried by creating a class from the input query, classifying it, and requesting the value of any desired output arguments.

An example of this mapping is the following version of the Prolog `append` function. Since I1, unlike Prolog, does not have a special syntax for list manipulation, the interpretation below assumes lists are represented with Lisp-like cons cells, e.g., `[a,b]` is represented as `cons(a,cons(b,null))`. The components of a cons structure will be referred to with the `first` and `rest` roles. The following definitions also assume that the first two arguments of `append` are the inputs `input1` and `input2`, and the third argument is the output `output`.

```
; append(nil,Y,Y).
Append-null ≡ {Append} with {input1:Null}
Append-null ⇒ {output:<input2>}

; append(cons(X,Y),Z,cons(X,Z1)) :- append(Y,Z,Z1).
Append-non-null ≡ {Append} with {input1:Cons}
Append-non-null ⇒ {
  append-call: Append with
    {input1:<rest input1>,input2:<input2>}
  output: Cons with
    {first:<first input1>,rest:<output append-call>}}
```

It may be useful to illustrate the above definition with some example inputs:

```
List1 ≡ Cons with {first:A,rest:Null}
List2 ≡ Cons with {first:B,rest:Null}
Append1 ≡ {Append} with {input1:List1, input2:List2}
```

Issuing a request for the value of output of Append1 results in the following computations:

- Append1 is classified beneath Append-non-null, since its input1 is a Cons.
- The output definition of Append-non-null is processed, causing a new Cons to be created whose first element is A (the first element of List1, the input1 to Append1).
- The rest element of this Cons is computed by expanding the template for append-call and computing its output. The template defines an Append class whose first element is Null and whose rest element is List2. This class is classified beneath Append-null since its input1 is Null, and hence its output will be its input2, i.e., List2.
- The final result (i.e., the value of output for Append1) is a Cons whose first element is A and whose rest element is List2; this represents the list `[A,B]`, as desired.

4.2 Programming a Realisation Grammar

A more complex (and perhaps realistic) example of classification programming is its use to represent and process a grammar for realising semantic representations as English sentences. Here are some simplified I1 definitions for classes of sentences:

```
Sentence ≡
  {Complete-phrase} with {semantics:Predication}
Sentence ⇒ {
  realisation: Sequence with {
      1:<realisation subject>,
      2:<realisation predicate>},
Imperative ≡ {Sentence} with {semantics:Command}
Imperative ⇒ {subject:EmptyPhrase}
Declarative ≡ {Sentence} with {semantics:Statement}
Declarative ⇒ {subject:<deep-subject>}
Active ≡ {Sentence} with {semantics:ActorTheme}
Active ⇒ {...,
  deep-subject:Noun-phrase with
      {semantics:<actor semantics>}}
Passive ≡ {Sentence} with {semantics:ActeeTheme}
Passive ⇒ {...,
  deep-subject:Noun-phrase with
      {semantics:<actee semantics>}}
```

Informally, a Sentence is a phrase whose semantics is a Predication. The realisation of a Sentence consists of the realisation of its subject and the realisation of its predicate (we will not consider the latter here). An Imperative sentence (when the semantics is a Command) has an empty subject. A Declarative sentence uses its deep-subject as its subject. This will always be a Noun-phrase, whose semantics is either the actor or actee specified in the sentence semantics. These definitions are used in the following way to realise a semantic representation α. A class Target is defined as follows:

```
Target ≡ {Sentence } with {semantics:α}
```

and then queries about the realisation of Target are posed.

In this instance, the semantic distinctions ActorTheme/ActeeTheme and Command/Statement are independent. A given (complete) semantic representation α will be classified under one of each of these pairs. It follows that Target will be classified *both* under one of Active/Passive and also under one of Imperative/Declarative. The appropriate value will then be inherited for the subject (either an empty phrase, or the value of the deep-subject, which will be a Noun-phrase with a semantics taken from an appropriate point in α).

Instead of giving separate definitions for Declarative and Active, the programmer could have associated the assertional statements about them (e.g., {subject:<deep-subject>}) with Sentence. Under I1's default inheritance mechanism, this means that a Sentence would by default be assumed to have these assertional properties unless its definitional information (i.e., the semantics

value) was such that it was classified beneath the more specialised class Imperative or Passive. Using default inheritance in this manner allows the grammar to process incomplete semantic inputs, which can be useful.

5 Comparison to Other Systems

There are interesting similarities between classification programming and a number of existing programming schemes. Space does not permit a detailed comparison here, but we will attempt to summarise the most important points. We have already discussed briefly some similarities between classification programming and Prolog, though the rigid distinction between inputs and outputs suggests a closer match in some respects with functional languages such as ML [Milner *et al.*, 1990]. The use of (multiple) inheritance in I1 reminds one naturally of object-oriented languages, including Smalltalk [Goldberg and Robson, 1983] and the feature-oriented and declarative DATR language [Evans and Gazdar, 1989] for lexical description. The use of complex functional descriptions is reminiscent of recent feature logics and extensions to unification grammars, such as TFS [Zajac and Emele, 1990] and FUF [Elhadad, 1991], as well as of the type specification languages of Aït-Kaci and his associates — KBL [Aït-Kaci, 1984] and LOGIN [Aït-Kaci and Nasr, 1986]. Indeed, papers on TFS, FUF and KBL all present versions of the Prolog append definition that are similar to ours. Our discussion of I1 definitions as pattern-action rules suggests a connection with production systems such as OPS5 [Brownstone *et al.*, 1985].

5.1 Classification

The major way in which I1 differs from other systems providing inheritance, such as Smalltalk, DATR, TFS, FUF, LOGIN and KBL, is by the incorporation of a non-trivial classifier. In these other systems, the types or classes from which information can be inherited are primitive (there are no non-trivial sufficient conditions for class membership). This is essentially like running I1 without any $\equiv$ statements, with the programmer performing the necessary classification by hand[4].

[4]In fact, a limited kind of classification seems possible in KBL via the lattice ordering on the type signature. If a type which is equivalent to a conjunction of simpler types (and hence not primitive) can be associated with a KBL "definition", then a kind of classification would be needed to associate this "definition" with a class that was subsumed by both of the classes in the conjunction. But we have seen no examples of this being done in the papers on KBL and this in any case would involve only a very simple notion of classification (which does not take account of role values). NB KBL definitions associate only *necessary* conditions with classes and correspond roughly to I1 $\Rightarrow$ statements. Because this is a different sense of the word "definition" than our own, we will distinguish it by quotes.

5.2 Determinism

Prolog, TFS, FUF, LOGIN and KBL provide, in place of classification, unification and disjunction. A primitive class can be associated with a set of possible "definitions" and, when these are matched with the known role-value pairs of a more specific class, only certain combinations will turn out to be consistent. If more than one is possible, then non-determinism arises. In classification programming, classification determines uniquely where a class fits within the taxonomy and there is no non-determinism. In this respect it behaves more like a language like ML, where the first matching clause is always chosen (though in classification, it will be the most specific definition, regardless of the textual order). Classification programming sacrifices bidirectionality by this approach.

OPS5 provides disjunction implicitly by considering all productions whose conditions are satisfied. If an OPS5 program used a conflict-resolution strategy that preferred the most specific matching productions, then this might in some ways approximate the behaviour of a classification system; in practice OPS5 conflict-resolution strategies tend to also consider other factors, such as when a data element was created or last updated.

5.3 Object-Orientation

In classification programming, a given class may be classified beneath several parent classes, and thereby inherit will information from all of these classes. This is illustrated by our realisation grammar example, in which a sentence class may, for instance, be a child of both Declarative and Passive; such a sentence will inherit a value for subject from Declarative, and a value for deep-subject from Passive. Our append example (which looks close to similar definitions in Prolog, ML, TFS, FUF, KBL and LOGIN) is not really representative for this reason. When an I1 programmer produces definitions of subclasses of a given class these are not required to be disjoint. In the other languages, the multiple clauses for a predicate, function or type are interpreted as disjoint alternatives. The effect of multiple inheritance can be coded in other ways in these languages, of course. In the end, the difference is a matter of orientation. I1 classes are in this respect more like classes in Smalltalk and DATR (and, to some extent, productions in OPS5), and I1 references can be seen as a form of message-passing. The use of default inheritance in I1 is another feature in common with object-oriented systems.

5.4 Declarativeness

Although classification programming has an object-oriented orientation, unlike most OOP languages (with the exception of DATR) it is purely declarative. It also differs from OPS5 in this respect.

5.5 Recursion

The template mechanism in I1 is what distinguishes it from other classification systems; in particular, templates permit the I1 programmer to implement recursion and procedure calls. Several systems have been built that combined classification and forward chaining production rules, including CLASP [Yen *et al.*, 1991], CLASSIC [Brachman *et al.*, 1991], and CONSUL [Mark, 1980]; these rules could perhaps be used to achieve some of the same functionality as I1's templates, but we are not aware of any attempt to use these systems as we are using I1. CONSUL also seems to have had some template-like capabilities, but the details are unclear, and again this facility was not used in the way we use templates in I1.

5.6 Laziness

In I1, assertional roles are only accessed (and templates expanded) when this is necessary to answer a query. This lazy evaluation strategy, which is not the only mechanism that could be used in classification programming, differs from the forward chaining rules of OPS5 and CLASSIC. It is similar to lazy evaluation as it appears in functional languages.

5.7 Procedures vs Data

As with TFS, FUF and KBL, I1 makes no distinction between procedures and data, the same operations being applicable to both. In this respect it differs from LOGIN, Prolog, ML and OPS5.

6 Conclusion

This paper has shown how a KL-ONE-like classification system can be used to execute general-purpose programs, if it is augmented with a default inheritance system and constructs that allow recursion to be programmed (e.g., templates). The resultant programming system has some similarities to logic programming, production-rules systems, and object-oriented approaches, but does not fully fall into any of these categories. Classification programming has proven to be a very useful tool in the IDAS system, and we expect that it will be similarly useful in other knowledge-centred applications which require integrating some algorithmic reasoning with a KL-ONE-like domain knowledge base. Whether the classification approach will lead to a general-purpose language that is as useful as, say, Prolog is unclear at this point in time; at minimum, however, classification programming provides a novel and interesting perspective on what constitutes programming, and on the relationship between knowledge and reasoning.

References

[Aït-Kaci, 1984] Hassan Aït-Kaci. *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures.* PhD thesis, University of Pennsylvania, 1984.

[Aït-Kaci and Nasr, 1986] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[Brachman and Schmolze, 1985] Ronald Brachman and James Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[Brachman *et al.*, 1991] Ronald Brachman *at al.*. "Living with CLASSIC: When and How to Use a KL-ONE-Like Language". In [Sowa, 1991].

[Brownstone *et al.*, 1985] Lee Brownstone, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5.* Addison-Wesley, 1985.

[Elhadad, 1991] Michael Elhadad. "FUF User's Manual - Version 5.0". Technical Report CUCS-038-91, Columbia University, 1991.

[Evans and Gazdar, 1989] Roger Evans and Gerald Gazdar. Inference in DATR. In *Proceedings of Fourth Meeting of the European Chapter of the Association for Computational Linguistics (EACL-1989)*, pages 66–71, 1989.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *SMALLTALK-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Levesque and Brachman, 1985] Hector Levesque and Ronald Brachman. "A Fundamental Tradeoff in Knowledge Representation and Reasoning". In *Readings in Knowledge Representation*, Eds. R. J. Brachman and H. J. Levesque. Morgan Kaufmann, 1985.

[Mark, 1980] William Mark. Rule-based inference in large knowledge bases. In *Proceedings of the First National Conference on Artificial Intelligence (AAAI-1980)*, pages 190–194, 1980.

[Milner *et al.*, 1990] Robin Milner, Mads Tofte and Robert Harper. *The Definition of Standard ML*, MIT Press, 1990.

[Reiter and Mellish, 1992] Ehud Reiter and Chris Mellish. Using classification to generate text. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL-1992)*, pages 265–272, 1992.

[Reiter *et al.*, 1992] Ehud Reiter, Chris Mellish, and John Levine. Automatic generation of on-line documentation in the IDAS project. In *Proceedings of the Third Conference on Applied Natural Language Processing (ANLP-1992)*, pages 64–71, Trento, Italy, 1992.

[Sowa, 1991] John Sowa, editor. *Principles of Semantic Networks.* Morgan Kaufmann, 1991.

[Yen *et al.*, 1991] John Yen, Robert Neches, and Robert MacGregor. CLASP: Integrating Term Subsumption Systems and Production Systems. *IEEE Transactions on Knowledge and Data Engineering*, 3:25–32, 1991.

[Zajac and Emele, 1990] Rémi Zajac and Martin Emele. "Typed Unification Grammars" . In *Proceedings of the 13th International Conference on Computational Linguistics (COLING-1990)*, Helsinki, 1990.