PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

A LANGUAGE FOR DESCRIBING CONCEPTS AS PROGRAMS

Claude Sammut and Brian Cohen

Department of Computer Science
University of New South Wales

ABSTRACT

A learning program produces, as its output, a boolean function which describes a concept. The function returns true if and only if the argument is an object which satisfies the logical expression in the body of the function. The learning program's input is a set of objects which are instances of the concept to be learnt. A compiler/interpreter has been written which performs the reverse of the learning process. The concept description is regarded as a program which defines the set of objects which satisfy the given conditions. The interpreter takes as its input, a predicate and produces as its output, an object which belongs to the set.

1. INTRODUCTION

In the field of pattern recognition, there has been increasing interest in structural description languages. These are formal languages which are used to describe objects and the concepts (or patterns) to which they belong. Of further interest is the ability of programs to learn the description of concepts in terms of such description languages.

This paper describes such a language currently under development. The compiler which exists is intended to be used in conjunction with a learning algorithm similar to that developed by Cohen (1978). Although intended primarily for use by another program, the language illustrates many features included in new languages for Artificial Intelligence. We will discuss these features, comparing them, in particular, with similar features in QLISP (Rulifson et al, 1972) and PROLOG (Kowalski, 1974).

The syntax of the language is quite simple, being a form of predicate logic. An object description is passed as a parameter to a predicate function. If the conditional expressions which make up the body of the function are satisfied then the object is said to be recognized by the concept that the predicate represents.

For example, if we represent a binary number as a string of 1's and 0's (with no leading 0) then we define "number" as follows:

```
define number =
   [x: x.head = nil & x.tail = 1 | number(x.head) & digit(x.tail)]
```

where "x" is the object passed to the function, and digit is the concept:

```
[x: x = 0 | x = 1]
```

The recognition of a number is equivalent to parsing the string left to right. It is a recursive concept with the termination condition given in the first disjunct. Although recursive functions are not new to programming languages, few concept description languages include this important feature.

Concept descriptions are closed sentences. That is, the only variables known to a function are those passed as parameters. An object is described in terms of its properties. A physical object, say, may have properties such as size, shape, colour, etc. The description of such an object would consist of a set of property/value pairs, for example:

box : <shape: cube; colour: red; size: big>

The number 2 may be described as:

<head: one; tail: 0>

where one is:

<head: nil; tail: 1>

Values may be atomic (e.g. 1 and "nil") or structured (e.g. one).

2. QUANTIFIED VARIABLES

A learning program, such as Cohen's CONFUCIUS, takes as its input a number of objects as instances of a concept which is to be learnt. From these examples, a concept description is formed. That is, the output of the process is a predicate which is true if and only if the argument is an object which satisfies the logical expression.

The description of complex concepts may require a language with more expressive power than simple expressions joined by "and" or "or" operations. For example, the existential quantifier may be used in the description of the concept of "maximum number in a list":

```
define maximum =
[list, max:
    list.tail = nil & max = list.head                (1)
    | [E x: maximum(list.tail, x) &                  (2)
            (
              lt(list.head, x) & max = x             (3)
            | le(x, list.head) & max = list.head     (4)
            )
      ]
]
```

("list" and "max" are the two arguments to "maximum". "lt" is the relation "less than" and "le" is "less than or equal").

However, the introduction of quantifiers raises many propblems in the evaluation of an expression. How do we test a statement which claims the existence of an object? It would be impossible to search for such an object since the set of all possible objects may not be specified and may not even be finite. This leads us to ask, is it possible to construct an object which will satisfy the given condi-

tions? Thus, the task of our language is to take as input a predicate and attempt to produce, as output, an instance of the concept. This is the reverse of the learning process.

Let us consider briefly how we might find the maximum number in a list. The following statement may be used:

[E x: maximum(L, x)]

For this example, let us assume L = (1 5 3 2 4). The value of "L" is passed to "maximum"; a special "quantified variable" is created to represent x.

Initially, the tail of the list is not nil, therefore line (1) in the definition above is false, causing the second part of the disjunction to be evaluated. Now a new quantified variable is created in order to represent the maximum of the tail. On line (2) "maximum" is called recursively until list.tail = nil. At this point we test max = list.head. "max" is a quantified variable which has no value, yet. When an equality test involves an undefined quantified variable, the operation becomes an assignment rather than a boolean expression. Thus max is temporarily assigned the value of list.head, which happens to be 4.

The call, "maximum((4), 4)" now returns. Since x = list.head, line (3) fails, but line(4) is true. The function exits. The overall effect is that 4 is passed back as the maximum of the tail until the recursion returns to the level where list = (5 3 2 4). Whereas, in previous activations max = x was executed, now max = list.head is executed (i.e. the maximum is changed to 5). In fact this will be the final result.

3. BACKTRACKING

So far, we have quite a straightforward evaluation process. However, there is no guarantee that the learning program will construct the concept with the statements in the order written down above. Suppose the body of maximum is:

```
list.tail = nil & max = list.head
| [E x:
        ( lt(list.head, x) & max = x
        | le(x, list.head) & max = list.head
        )
        & maximum(list.tail, x)
  ]
```

The expression "lt(list.head, x)" is encountered before the call to maximum so the system would attempt to construct an "x" greater than list.head. Since the first element is 1, the number 2 might be constructed, then maximum is called again. In this call, the system would be required to construct an "x" greater than 5. However, no such construction would be successful since the number constructed must belong to the list but there is no number greater than 5 in the list. In the disjunction, the second disjunct should have been chosen rather than the first.

These considerations require the implemention of the system to include backtracking facilities. Each simple logical expression may be considered as a goal to be achieved. Our problem is to find a set of goals which, together, satisfy the overall goal without conflicting, as happened in the example above. A conflict between two goals must be resolved by choosing an alternative if one exists. If one does not then the object cannot be constructed, therefore the quantified statement is false.

Execution is controlled by a queuing system:

1. When a simple expression is first encountered, a new goal is created and put on a "ready" queue.

2. The first goal in the "ready" queue is attempted. The effect of such an attempt is either to test the value of an object (or its properties) or, if no value exists, to give it one.

3. An expression which has given a symbol a value is said to control it. If it is found that a quantified variable is already controlled then the current value is checked. If this goal agrees with the controller, then the current goal is put on a queue associated with the variable. This queue contains all the controlling expressions.

4. If two goals are found to conflict then the current goal must temporarily be considered false. Therefore, it and the goals which belong to the same conjunct must be suspended. They go into a second queue associated with the variable, the "conflict" queue.

It is possible that, later on, a choice will be found to be incorrect. Therefore, the work of the expressions controlling the offending variable must be undone. A new value is then assigned to it by a set of goals taken from the conflict queue.

An example best illustrates this scheme.

```
[E x y z:
     (x = 0 | x = 1) &
     (x = 0 & y = 0 | x = 1 & y = 1) &
     (x = 1 & z = 0 | x = 1 & z = 1)
]
```

This is not a particularly useful statement, but it does demonstrate the need for backtracking quite well.

Let us briefly consider the execution of the concept. We may first choose "x" to be zero. This enables us to choose y = 0 also. However, with x = 0, no alternative in the last disjunct can be satisfied. Therefore, a new choice for "x" must be made since this is the variable which caused both conflicts. Since the value of "x" is to be changed, all statements depending on the old value must be considered false. Thus, the choice of y = 0 must also be undone. Now x = 1. The goals which govern the choice of alternatives are now reactivated. These result in y = 1 and either z = 0 or z = 1. (Since both values will satisfy the predicate, the choice depends only on the order in

which the goals are attempted).

Initially our system contains no knowledge. It does not even have "integer" as a basic data type. A concept of "number" must be learnt. However, given a powerful description language, the program can learn many useful concepts. "maximum" has already been described. Another is the concept of the sum of two numbers:

sum(x, y, z) is true if x + y = z

Once a definition of sum has been learnt it may be used as a program to perform an addition. For example:

[E z: sum(x, y, z)]

causes a "z" to be constructed such that it is the sum of x and y. A new concept may be learnt which has, in its description, a previously learnt concept. Thus, the language can be extended to include many useful concepts.

4. COMPARISONS WITH SOME AI LANGUAGES

It must be remembered that the initial intent of this language is that it be used by a learning program. It is very much a "computer language" since programs will be written by the learning system and executed without human intervention. Thus, the complete system represents a new approach to automatic program synthesis. Nevertheless, the language contains features that are useful in artificial intelligence and common to several high level languages. We will now consider some examples in PROLOG and QLISP.

The "maximum" program used before may be expressed in PROLOG as:

```
maximum(cons(x, nil), x).
maximum(cons(head, tail), x) :- maximum(tail, y),
                                swap(head, x, y).
swap(x, y, y) :- lt(x, y).
swap(x, x, y) :- le(y, x).
```

The left hand sides of the clauses above are goals. A goal can be achieved only if the expressions on the right hand side of the clause are true. When a function is called, and there is more than one occurrence of the goal expression (e.g. both "maximum" and "swap" occur twice on the left) then the occurrence whose arguments match those in the call is executed. Thus, if the list has a null tail, the first clause describing "maximum" is chosen. The choice of goal statements involves pattern matching. Note that in the example above "cons" is a list constructor as in LISP.

Although the notation here is somewhat different to that used in our language, the concept of using a goal directed form of programming is present. It also suffers from the problem that although, logically, the order of expressions in a clause is irrelevant, in practice the order can greatly affect efficiency. PROLOG is also based on predicate logic. It aims to provide a way of specifying a solution to

a problem in a completely machine independent way. Thus, the first decision in developing the language was to choose a convenient represention for humans rather than for computers.

QLISP is a language derived from LISP. It is intended to be used to write problem solving and theorem proving programs. Although it is a procedural language it also exhibits goal directed behaviour in its "GOAL" expression:

(GOAL goal-class goal)

Here the programmer asks the system to attempt some goal. For example, a robot planner might have an expression of the form:

(GOAL $DO (INROOM BOX1 ROOM4))

The system maintains a list, "goal-class", of programs which might help to achieve this goal. One such program is chosen . If it is not successful, the system backtracks and tries another alternative. If no alternative is left, a failure occurs.

These examples have in common the fact that a program tells the system what is to be done, but says very little about how it is to be done. That is left to the interpreter. That is, a statement in the language is regarded as a command to a problem solver. The ability to construct objects allows our system to solve problems also. For example, a concept may describe how a robot arm is to move a block from one place to another. The object which is produced then describes the state of the system after such a movement. The change in state may be interpreted by machinery resulting in a real block actually being moved.

It is hoped that work in the area of structural description languages such as the one described in this paper will result in powerful learning and programming systems, an important goal in artificial intelligence research.

5. REFERENCES

COHEN, B.L. (1978): "A Theory of Structural Concept Formation and Pattern Recognition". Ph.D. Thesis, Dept. of Computer Science, University of N.S.W.

KOWALSKI, R.A. (1974): "Predicate Logic as a Programming Language". 1974 IFIP Congress.

RULIFSON J.F., DERKSEN J.A., WALDINGER R.L. (1972): "QA4: A Procedural Calculus for Intuitive Reasoning". S.R.I. Artificial Intelligence Center, Technical Note 73. (Note: QLISP was originally known as QA4).