Solutions to Logic Problems in APL2

James A. Brown

IBM Santa Teresa Lab J88/E42
555 Bailey Ave.
P.O. Box 50020
San Jose, Calif. 95150 USA

Manuel Alfonseca

IBM Madrid Scientific Center
Paseo de la Castellana, 4
28043 Madrid, SPAIN

ABSTRACT

Logic problems are traditionally solved using notations invented for their solution. *APL* provides a different way to look at logic problems.

This paper looks at some logic problems and shows how they can be solved, at least conceptually, in parallel using *APL*. The solutions are contrasted with solutions to the same problems in PROLOG. Thus, it is shown that *APL* has logic programming capabilities in addition to its well known computational abilities.

INTRODUCTION

Logic programming has been with us for more than a decade. PROLOG-like languages are presented usually, in the literature on artificial intelligence, as the "proper" way to solve problems that involve some "reasoning" on the part of the machine, including all the set of what are generally known as "logic problems" (see references sm1 and sm2).

We can ask ourselves what is new about PROLOG, that makes it different to other, more traditional, programming languages. This question is easy to answer: PROLOG is non-procedural, meaning that PROLOG instructions (see cl1) do not need to be given in any pre-established order. The system (the "inference processor") has access to all the instructions at the same time, and selects by itself, depending on the actual values of the data, the instruction that must be executed at a given instant (the rule or axiom to be applied, in the PROLOG slang).

However, things are not so easy as that. In actual practice, in a real application, it is usually necessary to define the order of certain operations or to change their order depending on certain conditions. The three basic elements of structured programming (Block, If-Then-Else, and Do-While) must therefore be included somehow in the PROLOG structure. This has been done in a surreptitious way. "Cut", for instance, is a control structure that allows the programmer to simulate the If-Then-Else clause. Most PROLOG systems also include a REPEAT statement to simulate the Do-While.

In this paper we make the statement that the PROLOG-like non-procedural structure is not the only way of solving artificial intelligence problems in a "natural" way, that is to say, with a program that is legible, compact, and represents the problem immediately. We believe that *APL2* provides a different way of programming, actually a parallel way of programming, that is at least as compatible with our way of thinking as PROLOG is, but may be even more appropriate to the structure of future computing machines.

The following pages show some examples of logic programming -- an area where PROLOG is considered to be strong. Appendix 1 contains PROLOG solutions to the same problems.

BOOLEAN LOGIC

Let *P* be a logical proposition. It has a truth value, i.e. it is either false or true.

For example,

- "All men are yellow"

  is false

- "Some men are yellow"

  is true

- "If all men are yellow then men are alive"

  is true.

- "If all men are yellow then today is Monday"

  may be true or false depending on the day of the week we are in.

In logic programming languages, you write statements that are true and then draw conclusions from them. For example, in PROLOG you can say that "John likes Mary" with the following assertion:

```
likes (John, Mary).
```

meaning that "It is true that John likes Mary".

On the other hand, *APL2* uses a different computational approach to write logic statements. The truth values "false" and "true" are represented in *APL2* as the numbers 0 and 1, respectively and you should write expressions that represent all the possible combinations of truth and falsity for the variables involved. The evaluation of the resulting expression then gives a set of "ones" and "zeros". Each "one" corresponds to a combination of the variables that make the expression true. The "zeros" represent combinations that make the expression false.

Let *P* represent a logical proposition and let *P1* represent its possible truth values. The set of all possible truth values for *P* is the two item vector:

```
      P1←0 1
```

The "negation" of *P* is defined as another proposition *P'* such that *P'* is true when *P* is false and *P'* is false when *P* is true. This means that the truth value of *P'* will be 0 when the truth value of *P* is 1, and vice versa. Therefore, the set of all possible truth values of *P'* can be represented as:

```
      ~P1
1 0
```

Given this representation, trivial expressions about *P* can be computed. For example, a tautology is always true:

```
      P1 ∨ (~P1)
1 1
```

A contradiction is never true:

```
      P1 ∧ (~P1)
0 0
```

From this point on, no distinction between the "propositional variable" (what was called *P*) and its possible truth values (what was called *P1*) will be made.

If you want to write logical expressions on two propositions, *P2*, and *Q2*, there are four possible combinations of truth values: If *Q2* is false, then *P2* may be false or true. If *Q2* is true, then *P2* may be false or true. Thus, for two variables, complete sets of values can be represented as four item vectors:

```
      P2← 0 1 0 1
      Q2← 0 0 1 1
```

Now non-trivial expressions can be written. The expression *P2∧Q2* is true only when both *P2* and *Q2* are true (the conjunction of *P2* and *Q2*):

```
      P2 ∧ Q2
0 0 0 1
```

The expression *P2 ∨ Q2* only fails to be true when both *P2* and *Q2* are false (the disjunction of *P2* and *Q2*):

```
      P2 ∨ Q2
0 1 1 1
```

De Morgan's law shows that the negation of a conjunction is a disjunction and vice versa. One formulation of this rule is:

```
P2 ∨ Q2 ↔ ~ (~P2) ∧ (~Q2)
```

Computationally this is

```
      (~P2) ∧ (~Q2)
1 0 0 0
      ~ (~P2) ∧ (~Q2)
0 1 1 1
```

which is the "or" function.

Logical implication is defined in propositional calculus as:

"*P2* implies *Q2* is logically equivalent to the disjunction of *Q2* and the negation of *P2*." The corresponding *APL2* expression will thus be:

```
      Q2 ∨ (~P2)
1 0 1 1
```

This result has a 1 wherever *Q2* is either greater or equal to *P2* and so implication could also be written with a single *APL2* primitive:

```
      Q2 ≥ P2
1 0 1 1
```

Finally, the statement "*P2* if and only if *Q2*" can be rephrased as "*P2* implies *Q2* and *Q2* implies *P2*." The formula is as follows:

```
      (Q2v(~P2)) ∧ (P2v(~Q2))
1 0 0 1
```

This expression is true precisely where both are false and both are true -- that is when the logic values match. Thus, this equivalence can be represented by the single *APL2* function EQUAL (=):

```
      P2 = Q2
1 0 0 1
```

Applying this to the previous expression of De Morgan's law you see that

```
     (P2 ∨ Q2) = ~ (~P2) ∧ (~Q2)
1 1 1 1
```

is a tautology. Thus, De Morgan's law is always true.

Expressions containing three variables have eight possible combinations of values:

```
      P3 ← 0 1 0 1 0 1 0 1
      Q3 ← 0 0 1 1 0 0 1 1
      R3 ← 0 0 0 0 1 1 1 1
```

Here is the computation of three different implications:

1. *P3* implies *Q3*

```
      Q3 ∨ (~P3)
1 0 1 1 1 0 1 1
```

2. *Q3* implies *R3*

```
      R3 ∨ (~Q3)
1 1 0 0 1 1 1 1
```

3. *P3* implies *R3*

```
      R3 ∨ (~P3)
1 0 1 0 1 1 1 1
```

Suppose that you claim that "*P3* implies *Q3*" and "*P3*" are simultaneously true (Modus Ponens):

```
      (Q3 ∨ (~P3)) ∧ P3
0 0 0 1 0 0 0 1
```

You might expect to see the representation of *Q3* from this computation (0 0 1 1 0 0 1 1). The answer differs from *Q3* where *P3* is false but *Q3* is true. Since it is claimed that *P3* is true, the boolean result is stronger than just *Q3*. It expresses the fact that both *P3* and *Q3* are true simultaneously.

Next, look at the chaining rule: If "*P3* implies *Q3*" and "*Q3* implies *R3*" then "*P3* implies *R3*". The results of the individual implications are already listed above. The computation of the chaining rule is:

```
      (Q3 ∨ (~P3)) ∧ (R3 ∨ (~Q3))
1 0 0 0 1 0 1 1
```

Again, you might expect the representation of "*P3* implies *R3*" (1 0 1 0 1 1 1 1) but again the result produced is stronger.

Suppose that, in addition to the chaining rule, you assert that *P3* is actually true:

```
      (Q3 ∨ (~P3)) ∧ (R3 ∨ (~Q3)) ∧ P3
0 0 0 0 0 0 0 1
```

This shows that *P3*, *Q3*, and *R3* are all simultaneously true. This is stronger than the result of "P" and "*P3* implies *R3*":

```
      (R3 ∨ (~P3)) ∧ P3
0 0 0 0 0 1 0 1
```

which makes no claim about the truth of *Q3*.

PARALLEL BOOLEAN LOGIC

This section shows how you might go about using the application of the *APL2* logical functions to solve logic problems for all solutions in parallel.

THE MARCH HARE

The following logic problem is adapted from Raymond Smullyan's book "Alice in puzzleland" (sm2). It is solved by application of parallel boolean logic.

> "The jam had been stolen by either the March Hare, the Mad Hatter, or the Dormouse. They were arrested and each made one statement.
> They were:
>
> a.) The March Hare: I am not guilty.
>
> b.) The Mad Hatter: I am not guilty.
>
> c.) The Dormouse: At least one of the others speaks the truth.
>
> Further investigation produced the following conclusions:
>
> d.) The March Hare and the Dormouse didn't both say the truth.
>
> e.) Only one of them was guilty.
>
> Who was guilty?"

The solution to this puzzle uses the concepts of parallel logic previously developed. There are three logic variables to deal with:

- The Dormouse truth (*DT*),
- the Mad hatter truth (*HT*),
- and the March-hare truth (*MT*).

These can be represented by the following three vectors:

```
      DT←0 1 0 1 0 1 0 1
      HT←0 0 1 1 0 0 1 1
      MT←0 0 0 0 1 1 1 1
```

These three variables represent all possible combinations of truth and falsity for the three entities.

What the Dormouse said can be expressed logically as follows: *MT*∨*HT*. The Dormouse told the truth if and only if one of the others spoke the truth. This equivalence is written in *APL2* using EQUAL. Therefore the complete expression of the Dormouse's statement is *DT*=(*MT*∨*HT*). This relationship must be true.

Condition d. must be false if both *MT* and *DT* are true. This is written: ~(*MT*∧*DT*)

Condition e. says that only one is guilty. Since two said they were not guilty one of them is correct. This is expressed as *MT*∨*HT*.

Finally, we know that d. and e. are true. since the three preceding statements are simultaneously true, the individual expressions may be "anded" together giving one expression of truth:

```
      COND←(DT=MT∨HT)∧(~DT∧MT)∧(MT∨HT)
      COND
0 0 0 1 0 0 0 0
```

The result represents the case where what the Dormouse says is true, what the Mad Hatter says is true, but where the March Hare is lying. Thus the problem is solved. Here is the final program that solves this puzzle, which includes a print out of the results:

```
      ∇JAM
[1] ⍝ ALICE IN PUZZLE-LAND
[2]  DT←0 1 0 1 0 1 0 1
[3]  HT←0 0 1 1 0 0 1 1
[4]  MT←0 0 0 0 1 1 1 1
[5]  COND←(DT=MT∨HT)∧(~DT∧MT)∧(MT∨HT)
[6]  'HARE IS ',((COND/MT)/'NOT '),'GUILTY'
[7]  'HATTER IS ',((COND/HT)/'NOT '),'GUILTY'
[8]  'DORMOUSE IS ',((~(COND/MT)∧(COND/HT))/'NOT '),'GUILTY'
     ∇
```

Lines 6, 7 and 8 can be explained as follows: *COND* says which of the possible combinations of truth values is compatible with the data. Therefore, the expression *COND/MT* gives the truth value of the statement by the hare (i.e. whether the hare said the truth or lied). Since the hare actually stated its own innocence (statement a.) line 6 is immediate. So is line 7, which applies the same discussion to the hatter. Finally, the dormouse is guilty if and only if both the hare and the mad hatter are not guilty, that is to say, if both said the truth ((*COND/MT*)∧(*COND/HT*).)

Here is the execution of the program:

```
      JAM
HARE IS GUILTY
HATTER IS NOT GUILTY
DORMOUSE IS NOT GUILTY
```

ALICE

The following problem, taken from (sm1) is a little different:

> "When Alice entered the forest of forgetfulness, she did not forget everything, only certain things. She often forgot her name, and the most likely thing for her to forget was the day of the week. Now, the lion and the unicorn were frequent visitors to this forest. These two are strange creatures. The lion lies on Mondays, Tuesdays, and Wednesdays, and tells the truth on the other days of the week. The unicorn, on the other hand, lies on Thursdays, Fridays, and Saturdays, but tells the truth on the other days of the week.
>
> One day Alice met the lion and the unicorn resting under a tree. They made the following statements:
>
> LION: Yesterday was one of my lying days
> UNICORN: Yesterday was one of my lying days
>
> From these statements, Alice, who was a bright girl, was able to deduce the day of the week. What was it?"

First the data must be defined. Here the variable *DAYS* is defined as the seven days of the week and *YEST* is defined as the day before each day of the week:

```
      DAYS←'Sun' 'Mon' 'Tue' 'Wed' 'Thu'
      DAYS←DAYS,'Fri' 'Sat'
      YEST←'Sat' 'Sun' 'Mon' 'Tue' 'Wed'
      YEST←YEST,'Thu' 'Fri'
```

Next, two variables are set up that describe the days when the lion lies (*LL*) and the days when the unicorn lies (*UL*):

```
      LL ← 'Mon' 'Tue' 'Wed'
      UL ← 'Thu' 'Fri' 'Sat'
```

Now you must write expressions that are true. There are two conditions under which the lion statement is coherent. Either this is one of his truth telling days and yesterday was a lying day or this is one of his lying days and yesterday was a truth telling day. Here are the boolean expressions that compute both of these:

```
     (~DAYS∈LL)     ∧     (YEST∈LL)
     1 0 0 0 1 1 1 ∧ 0 0 1 1 1 0 0
0 0 0 0 1 0 0
     (1 0 0 0 1 1 1 ∧ 0 0 1 1 1 0 0)/DAYS
Thu
     (DAYS∈LL)      ∧      (~YEST∈LL)
     0 1 1 1 0 0 0 ∧ 1 1 0 0 0 1 1
0 1 0 0 0 0 0
     (0 1 1 1 0 0 0 ∧ 1 1 0 0 0 1 1)/DAYS
Mon
```

This says that if the lion is telling the truth it could only be Thursday and if the Lion is lying then this could only be Monday. Thus, we may define a variable representing when the lion statement is coherent (*LC*):

```
     LC ← ((~DAYS∈LL)∧(YEST∈LL))
     LC←LC ∨ ((DAYS∈LL)∧(~YEST∈LL))
```

The same logic is true for the unicorn:

```
     (~DAYS∈UL)     ∧     (YEST∈UL)
     1 1 1 1 0 0 0 ∧ 1 0 0 0 0 1 1
1 0 0 0 0 0 0
     (1 1 1 1 0 0 0 ∧ 1 0 0 0 0 1 1)/DAYS
Sun
     (DAYS∈UL)      ∧     (~YEST∈UL)
     0 0 0 0 1 1 1 ∧ 0 1 1 1 1 0 0
0 0 0 0 1 0 0
     (0 0 0 0 1 1 1 ∧ 0 1 1 1 1 0 0)/DAYS
Thu
```

Here's the expression for the coherence of the unicorn statement (*UC*):

```
     UC ← ((~DAYS∈UL)∧(YEST∈UL))
     UC←UC ∨ ((DAYS∈UL)∧(~YEST∈UL))
```

By inspection you can see that only Thursday is true in both cases. Here, then is a summary of the solution in a more compact form:

```
    YEST ← ¯1⊖DAYS←'Sun' 'Mon' 'Tue' 'Wed' 'Thu' 'Fri' 'Sat'
  (LL UL) ← ('Mon' 'Tue' 'Wed')('Thu' 'Fri' 'Sat')
       LT ← ((~DAYS∈LL)∧(YEST∈LL)) ∨ ((DAYS∈LL)∧(~YEST∈LL))
       UT ← ((~DAYS∈UL)∧(YEST∈UL)) ∨ ((DAYS∈UL)∧(~YEST∈UL))
       (LT∧UT)/DAYS
Thu
```

This problem can therefore be solved using entirely boolean expressions in parallel written to describe precisely the problem as stated.

Sullivan and Fordyce (fo1) describe a clever scheme for implementing a production expert system in *APL* using Boolean logic.

AGES

Here is a puzzle that can be solved by logic programming languages but where *APL2* can express a much more direct and efficient solution:

> "The sum of the ages of John and Mary is 40, and their difference is 6.
>
> What are their ages?"

This kind of problem can be solved using the methods of logic programming but it can also be expressed as a set of linear equations as follows:

```
 40 = John + Mary
  6 = John - Mary
```

This is easily solved in *APL* as follows:

```
      COEFF ← 2 2⍴ 1 1 1 ¯1
      COEFF
1  1
1 ¯1

      40 6 ⌹ COEFF
23 17
```

This kind of problem, which requires a certain number-crunching capacity, is not amenable to an "effective" solution by a classical logic language such as PROLOG (see Appendix 1).

CONCLUSION

We believe that the above examples show that the applicability of *APL2* to Artificial Intelligence has probably been underestimated, and should be redefined. This is not to say that *APL2* should be the language of choice for every possible problem in Artificial Intelligence. No language is

good for everything, and all languages are specially suitable for something. A PROLOG solution may be a much better choice for a given application. What we are stating is that *APL2* should be considered as one of the standard possibilities for the design of Artificial Intelligence applications, to be selected or rejected on the basis of actual, practical considerations.

Finally, *APL2* itself could be extended in some way to make it more useful for those problems (if any) where it is not optimally applicable at the moment.

REFERENCES

- cl1: W.F. Clocksin, C.S. Mellish, "Programming in Prolog", Springer-Verlag, 1981.
- fo1: Fordyce, K., Sullivan, G. ,"Artificial Intelligence Development Aids (AIDA)", Proceedings of Lapl.APL85, *APL* Quote QUad, Vol. 15, No. 4, 1985, pp.106-113.
- sm1: Raymond Smullyan, "What is the name of this book?", Prentice Hall, 1978.
- sm2: Raymond Smullyan, "Alice in puzzle-land", Prentice Hall, 1982.

APPENDIX 1: PROLOG SOLUTIONS

This appendix gives possible solutions in PROLOG to the puzzles solved in the paper.

THE MARCH HARE

```
guilty(hare)<-¬true(hare).
guilty(hatter)<-¬true(hatter).
guilty(dormouse)<-¬guilty(hare)
                  & ¬guilty(hatter).
true(dormouse)<-true(hare)|true(hatter).
¬(true(hare)&true(dormouse)).
true(hatter)|true(hare).
```

Invocation is:

```
  <-guilty(*).
(i.e. who is guilty)
SUCCESS: guilty(hare).
```

ALICE

The following is a PROLOG solution of the problem of Alice in the forest of forgetfulness:

```
yest(sun,sat).
yest(mon,sun).
yest(tue,mon).
yest(wed,tue).
yest(thu,wed).
yest(fri,thu).
yest(sat,fri).
ll(mon).
ll(tue).
ll(wed).
ul(thu).
ul(fri).
ul(sat).
lc(*day) <- yest(*day,*yest) & ll(*day)
            & ¬ ll(*yest).
lc(*day) <- yest(*day,*yest) & ll(*yest)
            & ¬ ll(*day).
uc(*day) <- yest(*day,*yest) & ul(*day)
            & ¬ ul(*yest).
uc(*day) <- yest(*day,*yest) & ul(*yest)
            & ¬ ul(*day).
get(*day) <- lc(*day) & uc(*day).
<- get(*day).
```

Its structure is very similar to the APL2 solution.

AGES

The VM/PROLOG solution is as follows:

```
is_integer(1).

is_integer(*X) <- is_integer(*Y)
                  & sum (1,*Y,*X).

ages(*John,*Mary) <- is-integer(*John)
                     & sum(*John,*Mary,40)
                     & diff(*John,*Mary,6).
<- ages(*John,*Mary).
ages(23,17).
```

The problem with the PROLOG solution is the method of solution. First, the system assigns variable John a value of 1. Then it tries to find a value of Mary that complies with line 3. It can't, so it tries for John the value of 2. And so on, until it tries value 23 for John, the condition holds, and the answer is given. (But what if the answer would have been 1E6?. This program would need a million trials to reach it).

The following more direct PROLOG statement is rejected by all commercial PROLOG systems known to the writers.

```
ages(*John,*Mary) <- sum(*John,*Mary,40)
                     & diff(*John,*Mary,6).
```