Some Comments on "Pitfalls in Prolog Programming"

W F Clocksin
Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge CB2 3QG
England.

In their useful article "Pitfalls in Prolog Programming", Ng and Ma [1] discuss some ways that novice programmers can be confused while learning Prolog. They suggest that the confusions result from certain properties of Prolog, in particular the way that it diverges from the "ideal" of programming in pure logic. In fact, I suggest that Ng and Ma are attributing to Prolog powers that it simply does not have, and that confusions are usually the result of the novice's misplaced confidence in his abilities to reason about programs.

The Real Pitfalls

Ng and Ma motivate their article with the statement that, "If a user wants to write a Prolog program just according to his logical reasoning, he will probably fall into an unexpected trap of the language". Ng and Ma seem to be saying that the "logical reasoning" of the user is always valid, and if the user falls into a trap, then it is Prolog's fault. Whether or not the user's logical reasoning is always valid, the "ideal" of programming according to Ng and Ma is to "...reduce the gap between human reasoning and the machine's processing mechanism to the minimum". After considering several examples, Ng and Ma conclude that, "...programming in Prolog is not as simple as a direct transformation of the human-oriented reasoning." I believe there are three problems with the view taken by Ng and Ma:

1. It is wrong to assume that the user's reasoning is logical.
3. It is wrong to assume that "human oriented" reasoning, whether logical or not, can be directly translated into a program.
2. It is wrong to assume that the purpose of Prolog is to correctly execute directly translated "human oriented" reasoning.

The (mistaken) view -- that the logical reasoning of a programmer can be translated directly into a Prolog program that correctly executes the programmer's intention -- seems to be a prevailing undercurrent in much recent discussion about Prolog. How did this come about? From second-hand popular accounts of Japan's Fifth Generation programme? By uninformed sales pitches of purveyors of Prolog systems? Whatever the origin of this folk myth, it certainly *did not* come from the only reference [2] cited by Ng and Ma.

The Examples Revisited

I shall briefly discuss each example raised by Ng and Ma.

1. `factorial`. The `factorial` program as given by Ng and Ma cannot possibly be expected to compute inverses. The real pitfall is to assume that all programs contain their own inverses. This is not true even in pure logic, particularly where arithmetic is involved.

2. `collect_list`. Ng and Ma want the two versions to compute the same answer, but they don't. In fact, their reasoning is faulty, and cannot be corrected even by an appeal to pure logic. A list is ordered even in pure logic. The problem with their `collect_lists` is a simple conceptual bug that can appear in any program ranging from assembly language to pure logic. It is also worth mentioning that the point of this example has nothing to do with the extralogical status of the `retract` goal used in `collect_list`. The `retract` goal can be replaced with some logically pure generator of results: Ng's and Ma's (faulty) argument is not affected, and my reply is still valid.

3. `nextlevel`. It appears that Ng and Ma don't like having to find the `union` to create a new list. There are several points to make here. First, Ng and Ma think that "Prolog does not allow the instantiation of a group of elements at the head of a list at the same time", and this is why `union` is needed. This could mean several things. Prolog (and logic) allows variables to be instantiated to lists. But this gives a list of lists, which is not desired in this program. Prolog (and logic) allows pairs of variables to represent "difference lists". This technique could be used here, (and is often used to obviate an explicit use of `concatenate`). If it is required to remove duplicate son nodes, then the requirement here is for a true union. Even a pure logic program would need a union in that case.

4. `insert_sort`. Prolog's comma connective is not the same as a logical "and".

5. `adjacent`. Here a program for searching a directed acyclic graph is given, but Ng and Ma complain that it cannot search non-directed cyclic graphs. Again this is not a Prolog problem but a conceptual problem. One problem illuminated by this example is "hopeful naming". Naming some relation by the name `adjacent` and then complaining that the result does not adequately represent reflexivity is like defining the program

```
intelligent(A,[A|T]).
intelligent(A,[B|T]) :- intelligent(A,T)
```

and then being disappointed because it does not behave intelligently. Ng and Ma show how to correct their graph searcher, and correctly conclude that the problem is really due to implicit assumptions which are not stated by their original program. Their corrected program explicitly represents the various logical assumptions required to represent their original intent. It is therefore not surprising that Ng and Ma notice an "incongruity between the human reasoning and the programming". But this is not a problem with Prolog. It is a problem concerning the fluency with which they are reasoning about their requirements.

6. `findall`. It is true that Prolog makes available extralogical features. Ng and Ma complain that the user's time is spent dealing with the programming issues caused by extralogical features, instead of exploiting the logic of the problem. When Ng and Ma correctly conclude that "the non-programmer is required to have also programming knowledge and not only the possession of logical reasoning power", we must consider two consequential questions. First, it is reasonable to expect "non-programmers" to write programs? And, after what we have seen, can Ng and Ma really trust their "logical reasoning power"?

Conclusions

Many people, even programmers, do not have enough experience in thinking logically and carefully enough to expect that their every requirement can be easily translated into a correct program. This should not come as a surprise. This has nothing to do with whether some programming language (say Prolog) contains more or fewer extralogical features. Indeed, programming in pure logic does not make the job easier: it is more demanding of human reasoning, as there are even fewer implicit assumptions to guide design decisions.

References

[1] K.W. Ng and W.Y. Ma. Pitfalls in Prolog Programming, *SIGPLAN Notices* 21(4), 75-79, April 1986.

[2] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.