The 'Marseille Interpreter' – a personal perspective

F. Kluźniak, Warsaw University

The Marseille Interpreter (Battani and Meloni, 1973; Roussel, 1975) started it all, but many of the younger Prolog fans may not have even heard of it. In the second half of the seventies, we have grown to know it rather intimately while installing it, modifying it and using it in Warsaw. We therefore think it appropriate to relate – briefly – some of our experiences: though the tone of these remarks is light, they are intended as a tribute to an awe-inspiring achievement.

The first installation of Prolog in Warsaw was that of a version prepared by Le Gloan (1974) for the CDC 6000 series. The principal difference with respect to the original version (Battani and Meloni, 1973) was the manner in which the interpreter's tables were accessed. The CDC machines were very fast, but had only up to 128K words of memory; each word had 60 bits, though, so the potential amount of data items – addresses in various linked data structures – was three times as large. Le Gloan replaced all array accesses with calls to simple packing/unpacking routines.

The loss in execution speed was considerable. This was compounded by the fact that the interpreter was not a Prolog interpreter really: it could only interpret the internal representation of a Prolog program in its tables. The 'real' Prolog interpreter (i.e. the program which could read and store Prolog clauses, read and write terms, and the like) was itself a Prolog program (Roussel, 1975) executed by the FORTRAN interpreter.

The effect was that our CDC CYBER 73, which ran at approximately 1.2 million instructions per second, read in Prolog programs at the average speed of 5 seconds per clause. As the machine was constantly labouring under a heavy load of multiprocessed jobs flowing in from a number of remote card-reader/printer terminals, it was impracticable to run Prolog for longer than a minute or so. In spite of all the packing, one needed to run in a low-priority storage class (we needed at least 72 000 (octal) words, as opposed to the standard of 54 000 (octal) for FORTRAN compilations, etc.), so a one-minute job used to hang in the input queue for up to 10 hours. Longer runs had to wait until the weekend.

Fortunately, the interpreter's internal state could be saved on a file between successive runs, so in spite of all this, Prolog was used for various small tasks, even by students. S. Szpakowicz even wrote his PhD program – a parser for a significant subset of Polish – in Prolog. With the low turn-around, reading in

ten clauses at a time, it took him several months to get the program into the machine: spectacular evidence of Prolog's ability to captivate the mind!

In 1978 we obtained funding for porting Prolog to an ODRA 1305 (essentially an ICL 1900). The machine was much slower, but it had 24-bit words, so there was no question of packing: we had high hopes that the result would be a faster interpreter (in the end it turned out to be twice as fast as on the CYBER). The memory was also only 128K, but we could have all of it, as the machine had only a very simple executive program and was operated in open shop. There was even a certain measure of interaction: the card punch was in the very next room.

The Prolog system was in the form of four decks of cards. There was the interpreter proper, which consisted of about 2000 FORTRAN cards. Another FORTRAN program – about 350 cards – was used to create a binary file with the interpreter's internal state. This program, which we called 'The Initiator', could only read in Prolog programs in a very low-level form – essentially a character representation of the internal form of Prolog clauses. The interpreter used Prefix Polish representation of trees, so we called this low-level language Prefix Prolog. To give an example of its distinctive flavour, here is the well-known procedure APPEND†:

```
1.2APPEND3NIL000NIL0
3.2APPEND3.2012.203.2APPEND3123NIL0 .
```

The third deck – about 75 cards – started with a card defining the character set, followed by 17 cards of integer sequences defining the interpreter state's 'kernel' (the representation of NIL, etc.). Seven cards declared the non-character functors and predicates used in the Prefix Prolog program which followed, 'The Bootstrapper', which could read and execute programs written in what we called 'Prolog B'. This was rather primitive, but already similar to full Prolog ('Prolog C'). One could write:

```
+ APPEND(NIL,*L,*L)
+ APPEND(.(*EL,*L),*L2,.(*EL,*L3))
    - APPEND(*L,*L2,*L3) .
```

The last deck consisted of about 400 cards in Prolog B, defining the full Prolog Monitor (interpreter with 'real' diagnostics, high-level input/output routines, etc.). The Monitor was written in a style apparently designed to squeeze the last ounce of advantage from unification's ability to deal with multi-purpose arguments. Despite repeated attempts to read it, we could not at first understand more than small isolated fragments of this program, so for a long time we did it no harm apart from changing French diagnostic messages to Polish.

One of the things we did understand – at a later stage – was the way its parser dealt with 'expressions' (i.e. terms containing infix, prefix or postfix functors, also known as 'operators'). These were treated in the classical way –

† The first digit is the number of variables. Each functor is followed by its arity and variables are represented by integer offsets (not ambiguous, as numbers greater than 9 are not allowed).

expressions contain terms, terms contain factors, factors contain parenthesised expressions – except that the number of syntactic levels reflecting different 'operator' priorities depended on the highest priority declared by the user. As each possible priority level was being taken care of by a separate invocation of the appropriate routine, the result was that one was heavily fined not for the number of operators one used, but for the magnitude of declared priorities. The fine was not only the danger of recursion stack overflow, but also a significant decrease in I/O speed. We solved the problem by making the Monitor issue a warning when it first encountered an 'operator' declaration with priority greater than 10.

But it had taken us some time before we were able to attempt such modifications.† The first steps were rather distressing: after we had managed to make the FORTRAN programs compilable,‡ the Initiator simply stopped halfway through its input. After ploughing through several hundred lines of very unreadable FORTRAN, we discovered that it did, indeed, contain another STOP statement. There was no error message whatsoever, which was only proper, as upon the correct termination there was a print-out that all went well.

The cause of the error, however, remained a mystery for several days. We were saved by the fact that Szpakowicz is a very talented proof-reader: the first card of the Bootstrapper was somehow different from the original listing. This card defined the collecting sequence of characters – proof of commendable concern for portability of Prolog programs – and after it was shredded by the card-reader we had to reproduce it from the listing. What we didn't see at the time was that the first character was – what else? – a blank. The Initiator failed upon reading in a blank and not being able to find the proper entry in its dictionary.§

Our inability to divine such things from the FORTRAN text – the documentation was very brief indeed, and its technical contents consisted in a single drawing illustrating the principal data structures (it was inaccurate) – was caused by the fact that the coding was absolutely atrocious. Apparently, regard for portability led its authors to adopt a 'standard' FORTRAN subset which did not even contain the logical IF.¶ We very quickly had to decide that the best thing we could do was to spend several days at the keypunch, systematically changing the arithmetic IFs to logical IFs. This decreased the number of statement labels to the extent that we were able to trace flow of control for more than a few lines. We then had to repunch half of the IFs a second time and shuffle cards so that there was at least some resemblance to **if** ... **then** ... **else** and the bodies of loops were contained within those loops (the program's authors having evidently been

---

† We eventually learned enough to implement a significant extension of metamorphosis grammars: 'floating' terminals (Bień, *et al.*, 1980).

‡ Always a battle when switching from one 'standard' FORTRAN implementation to another. The new generations of programmers don't know what they are missing.

§ All this reflects on our amateurism at that time. We later took pains to expand in-line the main resolution driver, which was called once from the main program: this could save us five microseconds on a several-minute run.

¶ This is my opportunity to follow well-established tradition by alluding scathingly to a leading manufacturer's contributions to our field.

unaware of the notion of programming style). It was only then that we could start to read the program and begin to understand it.

What we saw was very illuminating – for instance, it was a joy to discover the principle of structure-sharing! Sometimes it was also irritating. To give an example, the Prefix Polish representation of trees made it rather costly to traverse a term (for example, to access the *k*th argument), but was probably justified by being more tightly-packed. Yet the representation of print-names was incredibly wasteful: the strings were not only not packed several characters per word (which would be acceptable to ease portability), but were stored as normal representations of terms which were 'dotted' lists of their characters. The resultant saving in the complexity of term composition/decomposition (UNIV) was certainly not worth the price, but representation details were not localised in small parts of the program, so there was simply no way to change such things.

The control state representation was more amendable to modification. The activation frame stack had a simple structure – variables being kept in a separate area – but the frames were intermingled with the trail list (whose entries were easy to recognise by being made negative). As a result, it was probably thought too inefficient to pop off unnecessary frames with each execution of the slash (now often referred to as 'the cut'). The opportunities for space-saving accorded by invocation of the 'ancestor slash' (which could cut off all choice-points between its invoker and a far-removed ancestor) were to difficult to resist, however. The side-effect was that at least one of the programs we had from Marseille was found to be specially contorted so that a simple slash could be replaced by a procedure call and an ancestor slash – this was as if Pascal's **while** ... **do** and **if** ... **then** ... **else** were implemented so inefficiently that in practice one had to rely on the **goto**! We found it very easy to add space-saving actions to the slash – and to determinate procedure exit as well – with no appreciable effect on execution speed.

The other changes we made – program tracing and so on – were too numerous to mention. There was one exotic modification: addition of the evaluable predicate NETT ('nettoyer'). As the interpreter had no tail-recursion optimisation – we had also not thought of it at the time, otherwise we would have easily implemented it† – the main reading loop created an activation record with each clause it processed. NETT was a version of the state-saving SAUVE routine: it simply destroyed the invocation stack, leaving only a frame or two at the bottom; a most effective optimisation, which was, alas, too difficult to apply to the useless representation of the Bootstrapper cluttering up the dictionary table.

All in all, the Marseille interpreter exuded a strong air of a very sophisticated and robust general design filled in and implemented by inexpert programmers. The design's robustness is evidenced by the extensiveness of our modifications: at one time or another we had rewritten well over 50% of the code, increasing its size by 25% with comments and routines of our own, and the program lived!‡

† Easily, because in Warren's terminology (Warren, 1977) all variables were global, anyway.

‡ In view of the system's size, this may not sound like a feat worth mentioning, but remember the primitive conditions and our own lack of expertise.

After we finished with the ODRA, a Pascal interpreter was written for the CYBER (Kluźniak, 1981). This used no bootstrapping and a different program representation,§ but otherwise – on a conceptual level – the general design of Marseille Prolog. It was very successful: reading time was about 20 clauses per second and small programs could be run in 54 000 (octal) words – the turn-around for our Prolog class was not worse than for FORTRAN. We used it quite extensively until the telephone line to the CYBER was cut off in December 1981.

Of course, the general design's sophistication may seem suspect in view of later developments. Yet, to our mind, they are but variations on the basic theme. True, backtracking was somewhat less rapid, as an activation record did not contain a pointer to the last choice-point – now the usual practice – but the record occupied less space. The importance of departing from the normal practice of programming language implementation by adopting the convention that a clause's variable instance frame be associated with the activation of its caller rather than activations of its body's literals – in spite of the indirect access – can only be appreciated by those who tried to do it differently. Implementation of Prolog in Prolog is now widely accepted as the method of choice, and if it causes difficulties in traditional (now: obsolete) computing environments, the principled decision to adopt it at the time merits the louder applause. The single important departure from that basic design – the introduction of local/global variable classification† – would probably not have been possible as a first step. It is difficult to imagine someone devising such subtlety without first being shown that the question is of some practical importance: that Prolog really works.

ACKNOWLEDGEMENT

The pronoun 'we' stands for the author and Stanisław Szpakowicz, who also read the paper carefully, suggested improvements to its style and refreshed my failing memory in several instances. Janusz S. Bień was instrumental in gaining a foothold for Prolog both in our environment and in our minds. Other people involved at various stages of the efforts are: Z. Jurkiewicz, J. H. Komorowski, M. Łaziński, S. Matwin and A. Szałas. I am grateful to all of them.

REFERENCES

Battani, G. and Meloni, H., (1973), *Interpreteur du langage de programmation Prolog*, Groupe d'Intelligence Artificielle, Marseille-Luminy.

Bień, J. S. and Laus-Mączyńska, K. and Szpakowicz, S., (1980), *Parsing free word order languages in Prolog*, in: COLING 80, proc. of the 8th International Conference on Computational Linguistics, Tokyo, pp. 346–349.

§ Direct tree representation, but tightly-packed print-names.

† Non-structure-sharing of Bruynooghe (1982) is competitive in this respect, though based on more conservative notions. While willing to concede that it may be best for present-day small computers, we consider it unprincipled. Its success seems to rest heavily on smaller variable-instance representation (cf. Mel, 1982): that a constant factor of 2 in memory size can be significant in terms of what is and what is not implementable seems a short-lived coincidence.

Bruynooghe, M., (1982), *The memory management of Prolog implementations*, in: Clark, K., *et al.* (1982).

Clark, K., Tärnlund, S.-Å., (eds.), (1982), *Logic Programming*, Academic Press, London.

Kluźniak, F., (1981), *IIUW-Prolog*, Logic Programming Newsletter 1.

Le Gloan, (1974), *Implantation de PROLOG*, Internal report, Universite de Montréal.

Mellish, C. S., (1982), *An alternative to structure sharing in the implementation of a Prolog interpreter*, in: Clark, K., *et al.* (1982).

Roussel, P., (1975), *Prolog Manuel de référence et d'utilisation*, Groupe d'Intelligence Artificielle, Marseille-Luminy.

Warren, D. H. D., (1977), *Implementing Prolog – compiling predicate logic programs*, D.A.I. Research Report Nos. 39 and 40, University of Edinburgh.