PROLOG AS A LANGUAGE FOR PROTOTYPING OF INFORMATION SYSTEMS

Raf Venken
Belgian Institute of Management
Kwikstraat 4
B - 3078 Everberg
and

Maurice Bruynooghe
K.U. Leuven
Celestijnenlaan 200A
B - 3030 Heverlee

Table of Contents

Abstract

1. Introduction
2. Inherent Virtues of Prolog for Rapid Prototyping
 - 2.1 Compactness
 - 2.2 Data Structures
 - 2.3 In-Core Data Base
 - 2.4 Modularity
 - 2.5 Program Generation
3. Current Prototypes
 - 3.1 User Interface
 - 3.2 Functional Decomposition of the System
 - 3.3 The Data Base
4. A Prolog System for Generating Prototypes
 - 4.1 Robustness
 - 4.2 Performance
 - 4.3 Extensible Set of Built-In Functions
 - 4.4 Data Base Interface
5. The Programming Environment
 - 5.1 Type Checking
 - 5.2 Data Flow Checking
 - 5.3 Modules
 - 5.4 Syntax-Oriented Editor
 - 5.5 Program Generators
 - 5.6 Debugging
6. Final Remarks, Realizations and Future Work
7. References

Abstract

Prolog an acronym for Pro-gramming in Log-ic, being a product of Artificial Intelligence research, has some inherent qualities which make it suitable for rapid prototyping. In addition to the description of Prolog's virtues, we propose some extensions to enhance Prolog in its role of a tool for rapid prototyping and of a general purpose programming language for real-life commercial information systems.

The development process of these information systems involves the generation of successive prototypes where each successor is closer to the user requirements. The final prototype becomes either the production system or a subsystem into the total system. The tools used to generate successive prototypes are called application generators and program generators. We claim that Prolog suits well for implementing these tools and the prototypes.

1. Introduction

Until recently, the use of Prolog was limited to a small community of Artificial Intelligence researchers, where it appeared as an alternative to the traditional AI-language Lisp. However the interest in it has exploded since it was announced to be considered as one of the major building blocks of the Japanese Fifth Generation Computer Project.

Although being a powerful very high level programming language, in its current form, Prolog has the outlook of an assembler and its programming environment is very poor. This is mainly due to sociological reasons: the small society of Prolog programmers consists of skillful experts with little interest in those down-to-earth issues. Hence there exist neither tools nor guidelines for developing good and efficient Prolog programs and learning to program in Prolog by a conventional formed programmer, requires a profound conversion of mind. For the moment, it is not at all clear, whether programming non-trivial problems in Prolog is feasible for the majority of those programmers.

Since 1975, we have been studying different aspects of logic programming, including the integration with (relational) data bases and tools for program development. As a continuation of this research we started to consider the use of Prolog for ordinary business applications (management information system). In fact, after some time, we were using Prolog for rapid prototyping and now we are looking for enhancements of Prolog to suit the programming of real-life commercial applications by means of successive prototypes.

This paper describes our experiences with Prolog as a tool for rapid prototyping. Section 2 describes the relevant virtues of Prolog; Section 3 mentions the system components covered by our current prototypes; Section 4 sketches the requirements of a Prolog implementation to allow the extension of its use as a prototyping tool; Section 5 proposes a programming environment for Prolog. We finish with some general remarks in Section 6.

2. Inherent Virtues of Prolog for Rapid Prototyping

2.1 Compactness

Prolog has a very concise and compact notation: our experience shows that Prolog programs are about 10 times shorter than equivalent programs in Pascal, Fortran or Basic. Due to its syntactical compactness and its semantical power, Prolog bridges the gap between formal specification languages and conventional programming languages.

2.2 Data Structures

Prolog offers an extreme ease of introducing and manipulating complex data structures. Here Prolog is at his best, the unification process (the basic step which matches actual parameters to formal parameters on invoking a Prolog rule) permits complex assignments and selections of data structures, resulting in simple and efficient programs, e.g. unifying `test(a, test(X,Y))`, where `X` and `Y` are free variables, with `test(X', test(X',b))`, `X'` being a free variable, results in the following assignments : `X'←a`, `X←X'←a` and `Y←b`.

2.3 In-Core Data Base

It is widely accepted that the tuples in a relational data base are syntactical variants of a subset of Horn clauses: variable and function free assertions.

e.g.

Father

John	Paul		Father (John, Paul) ←
John	Peter		Father (John, Peter) ←
Paul	Jim		Father (Paul, Jim) ←
Peter	Tim		Father (Peter, Tim) ←
Peter	Tom		Father (Peter, Tom) ←

Prolog provides a default in-core representation of such relations without the users having to worry about the maximal size of the relation (c.f. the maximal size of an array in Pascal). This is very attractive to handle heavily used but small (parts of) relations residing in a data base.

2.4 Modularity

The absence of global data (except the data base part of the program) and the single assignment structure (a variable refers to a particular object, its value is, possibly stepwise, instantiated (assigned-refined) during the execution of a program) result in extreme modularity and easily readable and reusable programs. Three gains that can only be achieved in conventional programming languages at the price of a non negligible effort.

2.5 Program Generation

The very simple structure of Prolog makes it easy to generate standard or families of programs from a small data base describing the application dependent features. One can distinguish two approaches: the compiled and the interpreted approach. The compiled approach consists of generating a program with the domain dependent features imbedded, where the interpreted approach is realized by writing a very general program that is driven by a small (in-core) data base describing those domain dependent features. E.g. we developed a general data-entry program, where the domain dependent information, that drives it, is generated from the file description and the screen-layout (as provided by existing commercial systems), an other example is provided by syntax-driven editors, where the syntax of the target language together with some semantical actions constitute the domain dependent information.

3. Current Prototypes

Until recently our prototypes were covering different components of a complete information system:

3.1 User Interface

One of our first applications was dedicated to the handling of the user interface, we wrote several modules for screen management and data entry that can be reused in completely different applications.

This package of screenmodules is accessible through different levels. The lowest level routines permit primitive actions like clearing parts of the screen, writing and reading strings of a specified length from specified positions of the screen, etc. Application programmers however prefer to attribute names to specific positions on the screen. Adding application dependent information describing the different fields of a screen and their relative order, one can easily develop general Prolog routines to display a screen, to write a string to a specific field, to read a string, to display values in all fields of a screen (the values being supplied in a list) to read values from all fields including terminator handling (to move to previous or next fields, or to abort the action), to read values supplying defaults, etc. The highest level provides a routine to interface with the file that corresponds with the screen. Supplying a description of the screen, the layout of the file, default values and constraints for the different fields of the file one gets a general data entry program that can be used by the final user to load his data bases and to validate the conceptual structure of the data.

3.2 Functional Decomposition of the System

Using a top-down approach together with the inherent modularity of Prolog, one can easily divide the system in a number of distinct modules each performing a specific task and thereby describe the functionality of the system. A description that serves as the basis for further development, because it is written in Prolog. To implement the lower level routines one can use either Prolog or an other suitable conventional programming language.

3.3 The Data Base

In his current form Prolog suits well to test the underlying (relational) data base

of an application to develop. By limited tests with small in-core data bases one can verify the consistency between the proposed file layout and the programs to develop. On the other hand, we recently established the interface between Prolog and a commercial filesystem. First performance results show it to be feasible to using our prototypes (with an interface to the user's filesystem) in the real user environment.

4. A Prolog System for Generating Prototypes

Using Prolog and appropriate tools (written in Prolog for portability), it is possible to develop and/or generate a working system which can be evaluated at the user's side. Once the user approved the functionality, modules with insufficient performance can be gradually replaced by modules written in more conventional languages (Pascal, Cobol, Basic, ...). This poses some requirements to the Prolog system and programming environment.

4.1 Robustness

The Prolog system must be able to handle nearly infinite computations without running out of memory. This requires the availability of a garbage collector to recuperate parts of the run-time structures but also to free space occupied by the chunks of information coming from the underlying data base. Facilities are needed to store and to update intermediate results in-core without wasting memory (ASSIGNMENT by updating an in-core relation).

4.2 Performance

One technique to improve the performance of programs consists of partial evaluation which reduces the amount of deductions at run-time by evaluating what can be evaluated at compile-time [KOMOROWSKI]. This technique permits the use of abstract data types (Ada packages) or generic programming (a general program that can be driven by domain dependent information) without loss of performance and resulting in very readable and maintainable programs.

Another technique consists of compilation [WARREN] and, within a few years, special hardware [NISHIKA].

4.3 Extensible Set of Built-in Functions

An easily extensible set of built-in functions allows to adapt the system to specific requirements of the application, e.g. a non standard screen-management, interfacing

with a particular filesystem or data base system, implementing well-chosen procedures at a lower level by built-in functions to improve performance.

4.4 Data Base Interface

Flexible development of applications requires an interface to commercial data bases with access at different levels: selection of a record at a time using a movable cursor (get, getnext), selection of whole sets of records (satisfying some criteria) at a time and the possibility of creating temporary relations by loading parts of a data base in-core.

5. The Programming Environment

In its current form, Prolog is a very high level language with the outlook of an assembler and has a poor programming environment. Recent research shows that major improvements to Prolog and the environment for developing Prolog programs are feasible.

5.1 Type Checking

Adding types and type checking to the language would be a major enhancement for rapid development of Prolog programs. The concept of polymorphic type checking [MILNER] suits well for Prolog.

E.g.

```
declare append (list(any), list(any), list(any))
append (.(X,L), M, .(X,N))
  ← append(L, M, N).
```

A polymorphic typechecker, provided with an appropriate list definition, would see that this Prolog statement that defines the concatenation of two lists is incorrect: this append can not be called with an empty list (which is a valid list(any)) as first argument. Being called with a non empty list, it will neither succeed, because the recursive call to append will finally have an empty list as first argument. Apparently a statement treating this empty list is missing:

```
append([], L, L)←
```

[MYCROFT, BRUYNOOGHE]

5.2 Data Flow Checking

Due to its single assignment structure (each variable is assigned a value only once during the execution of the program) it is possible to check the information flow through a program against inconsistencies. This technique consists of adding redundant information to the program describing the input or output pattern of the formal parameters, for which the program is designed [BRUYNNOOGHE]

E.g.

```
declare quicksort(i,o)
quicksort(.(X,L), S)←
  partition(.(X,L), L1, L2),
  quicksort(L1, S1),
  quicksort(L2, S2),
  append(S1, .(X,S2), S).

declare partition(i,o,o)
declare append(i,i,o)
```

Using IO-patterns which are different from the described ones can result in unexpected program behaviour. One could try to use the quicksort routine to obtain all permutations of a sorted list, which possibly would not work correctly if the program was not designed to use it this way too. The data flow checker analyses whether the actual calls are consistent with the declarations. Most typographical errors as well as conceptual errors can be detected this way, preventing exhaustive debugging.

5.3 Modules

The module concept should be available providing the possibility to import or export routines and data structures, hiding local data structures and routines to the programmer and resolving name conflicts between local entities of different modules. The next step is a compiler which translates Prolog-modules to an intermediate code or to machine code, and an adequate linker.

5.4 Syntax-Oriented Editor

A syntax-oriented editor, incorporating the module concept as well as the type and dataflow checking into a user friendly system could offer a non negligible gain of productivity in writing Prolog programs, by avoiding typographical and other down-to-earth errors, by warning for non-common constructs and by supplying information about

user or system procedures and data types on demand. Integration with librarian functions and data types is possible and desirable.

5.5 Program Generators

A lot of modules, performing tasks as data entry, report writing, queries, etc., are indispensable parts of an information system and are easy subjects for program generators. An extensive set of program generators (written in Prolog for portability) and libraries of general Prolog procedures must be available. A special tool needs to be developed for retrieving or generating the right module on simple demand.

5.6 Debugging

A separate and eventually less performant Prolog system specially dedicated to debugging should be developed providing easy-to-use debugging aids, debugging principles used in conventional languages are, eventually adapted, usable in Prolog: tracing, spying, redoing, etc... [BYRD].

Also new approaches are feasible [SHAPIRO], which provide a more fundamental and systematic approach to the debugging problem.

6. Final Remarks, Realizations and Future Work

Prolog is not the perfect Logic Programming language. Mastering its non logical features (control) requires a certain skill. Nevertheless it provides a number of features promised by Logic Programming. We have experienced it as a very powerful tool for rapid development of prototypes. E.g. we developed a prototype for a document retrieval system in 3 man-days. This system includes a data base consisting of information about technical papers, authors, keywords and editors. To build this system we used the user interface, described in 3.1, and the data base interface described in 3.3, these two modules are driven by a small (in-core) data base describing the application data. The only manual coding consisted in programming the higher level menus and integrating the generated modules. Because of its IO-intensive nature this program has nearly the same performance and response time as similar applications written in Basic or Pascal, which use the same file-system.

These results seem promising for further investments and developments which should facilitate:

- The integration of Prolog-modules in a working commercial application.
- The development of large Prolog programs.

Of the enhancements to Prolog we suggested in Section 4 and 5, we already developed a prototype for the date base interface (4.4) and the data flow checking (5.2). For the moment we start a 3 year R&D project supported by the Belgian Government (Diensten voor de Programmatie van het Wetenschapsbeleid) which includes a Professional Prolog System (as described in Section 4) and Programming Environment (Section 5) and some typical applications, which will serve to test, refine and tune the system.

7. References

BRUYNNOOGHE, M.:

Adding Redundancy to obtain more Reliable and Readable Prolog Programs.

Proc. First International Logic Programming Conf. Marseille 1982, 129-133 (1982).

BYRD, L.:

Understanding the Control Flow of Prolog Programs.

(Research Paper 151)

Edinburgh: University of Edinburgh, Dept. of Artificial Intelligence 1980.

KOMOROWSKI, H. J.:

A Specification of an Abstract Prolog Machine and his Application to Partial Evaluation.

Linköping Studies in Science and Technology Dissertations No.69,

Linköping: University of Linköping, Software Systems Research Centre 1981.

MILNER, R.:

A Theory of Type Polymorphism in Programming.

Journal of Computer and System Sciences. 17(3), 348-375 (December 1978).

MYCROFT, A., AND R. O'KEEFE:

A Polymorphic Type System for Prolog.

Proc. Logic Programming Workshop, Portugal 1983, 107-121 (1983).

NISHIKA, H., M. YOKATA, A. YAMAMOTO, K. TAKI, AND S. UCHIDA:

The Personal Sequential Inference Machine (PSI): its Design Philosophy and Machine Architecture.

Proc. Logic Programming Workshop, Portugal 1983, 53-73 (1983).

SHAPIRO, E.:

Algorithmic Program Debugging.

ACM Distinguished Dissertations Series.

MIT Press 1983.

SZEREDI, P.:

Module Concept for Prolog.

Proc. Workshop on Prolog Programming Environments. Univ. Linkoping and Institute of Technology March 1982, 69-79 (1982).

WARREN, D. H.:

Implementing Prolog-compiling Logic Programs.

(1 and 2)

Edinburgh: University of Edinburgh, D.A.I. Research Report No.39, 40, 1977.