Logic Programming: The Evolving Algebra Approach

Egon Börger[a]

[a] Dip. di Informatica, Università di Pisa, Corso Italia 40, I-56125 PISA,
boerger@di.unipi.it

Keyword Codes: D.2.2; F.1.1; F.3.2
Keywords: Software Engineering, Tools and Techniques;
Computation by Abstract Devices, Models of Computations;
Logics and Meanings of Programs, Semantics of Programming Languages

The paper surveys the work which has been done from 1986-1994 on specifications of logic programming systems by evolving algebras. [1]

1. Introduction

It is with pleasure that I have accepted the invitation to report on the use of evolving algebras in the area of logic programming. Because of the time and space limits, I refer those interested in technical developments to the two Prolog and WAM papers by Dean Rosenzweig and myself [BoeRos94a,94b]. All references are to [**?**] [2]. The best I can do here is to sketch the development of ideas in my work on the subject since 1986. I hope the audience will forgive that this account is largely personal.

2. The Problem

It all started after I had come to Pisa, an internationally known logic programming nest. Under the influence of this environment, my previous mainly complexity theoretical interest in Prolog[3] changed and quickly focussed on the simple question: *What is the definition of Prolog*?

I did not mean Horn clause logic whose model and complexity theory I knew well from my life as a logician. I meant the programming language with all its so called dirty features. I was puzzled and dissatisfied to hear from software engineers that the force of Prolog lies in logic and that one has simply to accept that control features— imposed by needs of implementations—cannot be fully reflected in this beautiful "declarative" character of the language. I thought that given the relative simplicity of Prolog with respect to other programming languages, it should be possible to build an execution

[1] Invited lecture to the workshop on *Evolving Algebras*, in: B. Pehrson and I. Simon (Eds.), *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam.

[2] As latex file the bibliography can also be obtained by anonymous ftp from the machine apollo.di.unipi.it in the directory pub/Papers/boerger.

[3] See my papers in Fund.Inf. X, 1987, 1-34; Springer LNCS 270, 1987, 37-48; the Udine book 1988, 59-94.

oriented model for the full language which is precise, has manageable size, is abstract and which reflects the logical content of Prolog programs in a transparent user-usable way.

I digged into the great variety of semantical and specification approaches in the literature. I tried them all—but none of them gave me what I was looking for. The typical experience was that many elegant and concise descriptions of Horn clause logic broke down (simply failed or combinatorially exploded) when it came to Prolog's very clear tree search strategies and its characteristic built–in predicates (for database operations, control, solution collecting, term manipulation, etc.). In the Spring of 1987, Yuri Gurevich visited Pisa and presented in a series of lectures the world première of the concept of evolving algebras: motivation, definition, application to specifications and proofs of simple properties for Turing and stack machines, for Pascal programs. (This material went into Part II of Gurevich's paper in the 1988 Udine book [Gurevich88a] where evolving algebras appear for the first time in the literature.)

3. The Solution

I was influenced by these lectures and the exciting conversations with Yuri Gurevich on semantics of programming languages in general and in particular on the concept of evolving algebras which I was lucky to learn in statu nascendi. Thus I tried evolving algebras for the Prolog project: after various discouraging failures it eventually worked (see my three Prolog papers written up in 1989/90 [Boerger90a,90b,92]). Compared to today's definition (see the 1994 Prolog paper with Dean Rosenzweig [BoeRos94b]) that model of the 1980s appears clumsy. But it was a precise, simple, abstract and in a very strong sense "logical" model for the full programming language, eligible as a primary mathematical definition for the semantics of the programming language.

By a *primary model* of a system I mean a mathematical model which formalizes the basic intuitions, concepts and operations of this sytem directly, without encoding, in such a way that the model can be recognized "by inspection" and thus justified as a faithful adequate precise representative of the system. Good primary models play a crucial role for provably correct specifications of complex systems. A provably correct specification transforms a given model A into another model B—an "implementation" of A— and proves that this implementation is correct; but such a specification remains an intellectual exercise if we do not know that A is correct. Ultimately, in a chain of such provably correct specifications, the first model must be primary in the above sense in order to provide, from the pragmatic point of view, a safe foundation for the whole specification hierarchy. The flexibility of the concept of evolving algebras makes it easy to construct satisfactory primary specifications. I see here great possibilities for reliability of requirement specifications, in particular in connection with developing scientifically sound warranty criteria for safety critical software systems.

4. Extensions

Once the evolving algebra model of Prolog was there, it quickly showed its potential. Its essential parts went as semantics definition into the international Prolog standard (see the ISO WG 17 report with K. Dässler from April 1990 [BoeDae90]). It provided a framework to formally specify the main database update views for Prolog; this settled a major and longstanding issue in the standardization debate (see the 1991 papers with B.Demoen [BoeDem91] and D.Rosenzweig [BoeRos91a]). The model also allowed us to clarify the

fundamental semantical issues for the standardizing definition of the problematic solution-collecting predicates (see the 1993 paper with D.Rosenzweig [BoeRos93]).

During the work with the implementors in the ISO standardization committee I realized how flexible this evolving algebra machinery is when it comes to a change of a design decision, even at an advanced stage. With evolving algebras one gets *extendability*—a major concern for specifications—for free. Gurevich's definition allows us, given any system, to tailor in an explicit way a formal model at the corresponding level of abstraction. Surely such modeling is partly an art; it doesn't come by itself. But evolving algebras allow you to express your ideas once you have them, and you can do this without any methodological overhead.

As a matter of fact the evolving algebra Prolog model could easily be modified in such a way that it provided models for various well known extensions of Prolog: by general constraints (see the Prolog III paper with P.Schmitt [BoeSch91]), by type constraints (see the Protos-L papers with C.Beierle [BeiBoe91,92]), by functional elements (see the Babel paper with F. J. López-Fraguas and M. Rodríguez-Artalejo in this workshop [BoLoRo94b]), by object-oriented features (Müller's paper in this workshop). In the meantime Gurevich and Moss had developed in their CSL'89 paper [GurMos90] an evolving algebra treatment of concurrency—which could also be adapted to the Prolog model and enabled us to model Parlog, Concurrent Prolog, GHC, Pandora (see E.Riccobene's PhD thesis [Riccobene92] and [BoeRic93] [4]). The accumulated results and the experience gained through this modeling activity have allowed us to turn evolving algebra formalizations of (at least sequential) systems into a *routine* job. However this know-how has still to be built into *algorithmic tools* to support the development of such specifications (by providing consistency checkers, by integration into interactive theorem proving or compiler generating environments, etc.).

Today we know that this ease with extendability of evolving algebra models, first experienced in building models for logic programming languages (see also the definition of an explicit interface between logic and control components in the formalization of the semantics of Gödel programs, obtained with E. Riccobene in 1992/93 through an extensive discussion of various models with Lloyd and Hill, the father and the mother of the language; see [BoeRic94]), is only one facette of the apparently unlimited applicability of evolving algebras for simple descriptions of languages, algorithms, protocols, architectures, ... This workshop shows examples of the rich variety.

The *simplicity* of evolving algebra specifications reveals an important property which distinguishes the methodology from many other formal specification frameworks. As I have experienced since 1990 with numerous programmers, system designers and implementors, formalizations by evolving algebras can be understood by the working computer scientist without any formal training in logic. It needs not more than half an hour of explanation, through simple examples, to convey to a hardware or software engineer the idea and the formal definition of evolving algebras in such a way that he can start to produce his own precise evolving algebra models. I would be very much surprised indeed should this extraordinary potential of the evolving algebra methodology not change drastically the industrial future of formal specifications.

[4] A propos, this crucial second step for the development of the concept of evolving algebras had too been presented by Yuri Gurevich in a series of lectures in Pisa in May 1990. Elvinia Riccobene had come from Catania to attend these lectures and the work with her on evolving algebra models for parallel logic programming systems grew out of there.

5. Proving Theorems about Real-Life Systems

Back to the Prolog model. Once the full model was there, the question was: can we *use the model to prove interesting theorems on real-life systems*? The crucial challenge came from Michael Hanus in the Summer of 1990: specify the WAM and prove the compilation of Prolog programs on the WAM to be correct. I was lucky to interest Dean Rosenzweig in this question in the Fall of 1990; given his expertise with the WAM, we succeeded within less than four months to completely solve the problem using evolving algebras. (Obviously we still had a way to go from that first solution to the present streamlined form of the specification and the correctness proof. See details in the WAM paper with D.Rosenzweig [BoeRos94a].)

The crucial methodological outcome of this work on the WAM was that we learned, through an example from real life, how to incorporate the principle of *stepwise refinement* into specifications by evolving algebras. Evolving algebras allowed us a successful systematic use of the *verify–while–develop* paradigm: going in many small steps (instead of a few big ones) we could prove theorems about correctness for a class of real compilers without being knocked down by combinatorial explosion.

A very important effect was that through developing many intermediate models between Prolog and the WAM, we explicitly and precisely defined various orthogonal parts of the complex system. By *orthogonality* we mean that each of these parts is defined with a precise interface to the rest of the system—typically through functions which are external for the single parts and represent for them the environment into which they are embedded. Since these interfaces are fully described in formal (mathematical) terms, they provide the possibility to do the following:

1. replace a part by a new implementation of its internal behaviour which does not affect the interface,

2. prove that the new system thus obtained is equivalent but better than the original system in terms of parameters which are precisely defined in the model.

The definition of orthogonal WAM parts allowed us also to not only *reuse the specification* but also to *reuse the correctness proofs* for extensions of the WAM which have been developed by including type constraints (PAM work with C.Beierle [BeiBoe91,92]) or arithmetical constraints (CLAM work with R.Salamone [BoeSal94]); see also the paper presented by C.Beierle in this workshop.

Thus our work on the WAM shows the possibility to use evolving algebra specifications for *reverse engineering*. Dean Rosenzweig will explain in his talk another interesting experiment in this direction which happened to come to our attention after the successful experience with the WAM and the transfer, mutatis mutandis, of the specification and proof methodology to Occam and the Transputer (see our joint work with I.Durdanovic [BoRoDu94]): a formal specification of the APE100 parallel architecture which at present exists only as C-code and as hardware. The evolving algebra models are intended to be used for the development of the next generation of this architecture. (See the joint work with G.Del Castillo and P.Glavan in this workshop [BoDeGR94].)

There are various implications, for the theory of logic programming, of the mathematical definition of Prolog and the WAM layers provided by our specification. We have defined separate modules around the simple Prolog nutshell for user-defined predicates, each for a different group of built-in predicates. These modules provide the mathematical basis to

extend abstract analysis from Horn clauses to real Prolog programs which contain built-in predicates formalized in those modules. The WAM layers (for predicate structure, clause structure, term structure, etc.) make mathematical study of WAM related execution or implementation issues possible. By the way it came to me as a surprise that in the logic programming community there is a widely shared belief that a Prolog program is "verified" or "correct" by the very fact that it comes as a logical formula (whose meaning is automatically provided by logic). The evolving algebra modeling of Prolog allows us to overcome this rather inappropriate view and to give correctness proofs for real Prolog programs with respect to these models.

6. Thanks

In conclusion I want to take the opportunity to thank first of all Yuri Gurevich for sharing with me the troubles and joys of his discovery from its very beginning. Then I want to thank the numerous persons mentioned in this talk who during these years of work with evolving algebras gave me the pleasure to divide with them the fatigue and the enthusiasm in discovering and breaking fresh ground for successful applications of formal methods to complex real systems. Last not least I want to express my thanks to the following colleagues for their valuable criticism and comments upon earlier versions of this summary: Christoph Beierle, Antonio Brogi, Uwe Glässer, Michael Hanus, Lutz Plümer, Robert Stärk. I also want to thank Imre Simon for having urged me to write up these notes which imposed upon me some useful reflections.

REFERENCES

1. Annotated Bibliography on Evolving Algebras. *Specification and Validation Methods*, E. Börger, editor, Oxford University Press, 1994, to appear.

One problem in the introduction of logic programming to the wider programmer community is disentangling the notion of Logic (with a capital â€˜Lâ€™) from its less formal meanings, and in doing so acknowledging that a word like â€˜logicâ€™ has many different definitions to many different audiences, from the lay person, through the programmer, to the mathematician and the logician.Â While evolving from different needs, set theory, with its set algebra, was found to be the exact same things a propositional logic, with its boolean algebra. Joining is Conjunction (AND) in a Clause.Â Prolog and other logic approaches were part of some of the original AI research in decades long past and have become outshined in recent years by the success of statistical and probabalistic models.