

A Possible Future History of Logic Programming

M.H. van Emden
Computer Science Dept, University of Victoria

Introduction

While it is well-known that it is hard to know anything about the future, it is less widely realized that it is also difficult to understand the present. It is therefore paradoxical that it sometimes helps to understand the present by means of a fictitious future history. In this article I am going to exploit this paradox.

I place the viewpoint well away into the future. Again this may seem paradoxical, because in computing everything is supposed to happen fast. Not everybody thinks so. Paul Graham has embarked on a project that he calls “The Hundred Year Language”, pointing out that it is only hardware that changes fast. Programming languages change slowly. This is because they are part of culture; they reflect how human minds tackle problems. Hence Graham’s “Hundred Year Language”.

It so happens that Graham is concerned with Lisp. This is not the only paradigm in need of the long-term perspective. Accordingly, I imagine a history of logic programming as it may be written in the 2020s.

As I’m already exercising your indulgence with speculation, I will not attempt the fictitious future history itself, but instead bring out the salient points in an equally fictitious review of this nonexistent history.

A Critical Review of Karl Senner’s “History of Logic Programming”

When the first edition of Senner’s history came out in 2020, it was widely praised for its compelling view of the development over many decades of logic programming. Reviewers praised it for its broad perspective, but deplored its lack of historical detail. Since then several collections of papers have made their way from estates via the auctions to various libraries. Senner has taken this opportunity to incorporate these recent findings in a new edition.

What has not changed in the new edition, and this is what I take issue with, is that Senner presents the development of logic programming as a relentless forward march towards an inevitable outcome. Of course, it *is* true that not a decade went by without some of the building blocks being fashioned that we take for granted as part of the majestic edifice that now dominates the landscape of programming.

The way Senner tells it, and his readers love him for it, is that there was a continuous forward movement. This certainly makes for a good story. But it does not recognize the fact that the received view of history is only as recent as the Tens. What I’m now asking the reader to imagine, Senner’s book in hand, what the situation of logic programming must have looked like in the mid Zeroes. I think you will agree that Senner’s story of a relentless march forward is an artifact of 20/20 hindsight.

Let us imagine it is 2004, to pick a representative year in the mid Zeroes. The great contributions of the Nineties are known; they were even widely celebrated. But were they known, in 2004, as contributions *to logic programming*? Senner, of course, does not say so. But he makes it easy to overlook the fact that in 2004 nobody seemed to recognize as such the contributions of the Nineties to logic programming. Let us consider three of these contributions here: the XML movement, the compilation/execution model based on virtual machines, and constraint programming.

XML The XML movement, which arose from the World-Wide Web in the mid Nineties, can be characterized as exploiting *the tree as universal data-structure*. We now view this as a useful further stage in a development that started in the early Seventies. But if you would have whispered “Colmerauer” into the ear of an XML devotee, you would have met with a blank stare. Following it up with a heavy hint like “Prolog” wouldn’t change anything.

Nor were the few remaining logic programmers excited by the XML move-

ment. They regarded it as part of the big, bad, ugly outside world that had robbed them of their lawful place in the limelight. From their point of view, XML and all its works was only something to reluctantly accommodate with yet another *ad hoc* interface, not something to be embraced as a tree technology to serve as the foundation of a new version of Prolog. From our present vantage point it is difficult to imagine how compartmentalized computing was at the time.

From our vantage point, the year 2004 is indeed a year of delicious ironies. XSLT was already gathering a following, with its adherents discovering that one could do all kinds of computational tasks as transformations on (XML) trees. Working from the other side, Yeow, working with WAM experts Horspool and Levy, had shown in a 2002 paper that parts of the WAM design could be used for a much faster XML parser. Yet the penny had not dropped.

Execution via virtual machine Another innovation in which the Prolog of the Eighties was hopelessly far ahead of its time was *execution via a virtual machine*. At the time, Prolog advocates pointed out that when Prolog returns the answer to the user in 100 milliseconds, it is irrelevant that a lower-level implementation executes in 10 milliseconds. To no avail. The very fact that Prolog was “interpreted” marked it as unfit for the “real world”.

One of the great contributions of the Nineties was the “real world” discovering the advantages of the compilation/execution model of WAM based Prolog in the form of the Java Virtual Machine¹. Perhaps in some quarters it was still somewhat suspect, SUN being a notoriously innovative company. The last doubts disappeared when this model was adopted for C# by a company that stood far above any suspicions of innovativity.

Constraint programming The third important development of the Nineties is *constraint programming*. Here especially Senner fails to give a sense of the many fumbling steps that were taken towards the software architecture of logic programming that we now take for granted. Indeed, we now take it so much for granted that a brief tutorial on the basics may be needed.

In the mid Seventies, it was customary to distinguish “Prolog” from “Pure Prolog”. The latter was almost the same as the use of SLD resolution with Horn clause programs. In Pure Prolog one can perform transformations of

¹Is it a coincidence that Tim Lindholm, a big contributor to Quintus’s WAM implementation, reappeared to play a similar role in JVM?

trees containing symbolic values only. Counting and arithmetic have to be simulated by such symbolic computation. To allow access to the processor's arithmetic, Colmerauer and Roussel had already added extra-logical features to Pure Prolog. Though this resulted in a practical language, it made it difficult to characterize the class of program-query pairs that yield logical consequences.

Colmerauer himself would probably not have agreed to this analysis of Prolog. For him Prolog was not part of logic programming. For him Pure Prolog was not an inviolable given just because it was resolution theorem-proving. He clarified his position by showing that Pure Prolog itself should be decomposed by regarding *unification as constraint-solving over terms*. The immediate use of this was to clarify the controversy over the occurrence check in unification. Colmerauer showed that constraint solving over finite terms (which are the constituents of Herbrand universes) corresponds to unification with occurrence check. He presented a unification algorithm that is more efficient by omitting the occurrence check and showed that this algorithm can be viewed as constraint solving over rational terms.

Let us denote by $CP(S)$ constraint programming over a structure S . $CP(S)$ embraces not only a structure in the sense of model theory (consisting of a carrier, constant elements, constants, relation symbols, function symbols), but also a system of efficiently representable sets of values and efficiently computable contraction operators for the constraints.

A good way to illustrate the distinction between $CP(S)$ and S is to consider the structure R of the reals. Constraint programming over R adds to R a system of selecting certain sets of reals as efficiently representable as intervals. It also adds contraction operators for these intervals corresponding to constraints over the reals such as the ternary sum constraint $x + y = z$ and the ternary product constraint $x \times y = z$.

Similarly the structure of ground Herbrand terms can be enriched by adding a system for efficiently representing certain sets of ground terms. The preferred system is to use a term to represent the set of its variable-free instances. This efficiently represents a sufficiently rich repertoire of sets of ground terms. The constraint is equality and the efficiently computable contraction operator is unification. This gives $CP(FT)$. The same method, but starting with the structure of rational ground terms gives $CP(RT)$.

The deconstruction of Pure Prolog suggested by Colmerauer's work would

now be written as:

$$\text{Pure Prolog} = \text{Schematic Prolog} + CP(FT).$$

Colmerauer was more interested in

$$\text{Schematic Prolog} + CP(RT)$$

and was not concerned with the fact that it is not a form of resolution theorem proving.

Colmerauer's work in the late Seventies was concerned with justifying unification without the occurrence check. The deconstruction of Pure Prolog is implied. It was made explicit by Jaffar and Lassez in their CLP Scheme. It is for this reason that the core that remains of Pure Prolog after separating unification is called "Schematic Prolog". The CLP Scheme explicitly allows the addition of any number of constraint programming structures to Schematic Prolog. For example,

$$\text{Schematic Prolog} + CP(FT) + CP(R) \tag{1}$$

is an interesting Prolog, as it uses the techniques of interval arithmetic to obtain a logically sound method for the approximate solution of equations over the reals. The answer constraints are in the form of membership of intervals with floating point numbers as bounds. This soundness of these answers is not affected by the inevitable rounding errors that occur in the computation of these answers. By incorporating machine arithmetic into Prolog in this way, an important motivation for the extra-logical features of the original Prolog had disappeared.

By 2004 several important CP structures were known. We already mentioned $CP(FT)$, $CP(RT)$, and $CP(R)$. Finite-domain constraints fit in this framework. The representable sets include the entire powerset. The constraints are equality, disequality, and *alldiff*. We call the resulting system $CP(FD)$. Boolean constraints were well known. Although constraint programming over integers had played an important role since CHIP, it was still not soundly implemented because integer overflow invalidated the result if it occurred. But of course, $CP(R)$ plus the unary integer constraint in principle made $CP(Z)$ available. Thus, in 2004, there were no obstacles to a Prolog that extends (1) by including also $CP(RT)$, $CP(B)$, $CP(Z)$, and $CP(FD)$.

Yet the only forms of Prolog that existed at the time were basically the Prolog that emerged in the Seventies. The innovations beyond that were

restricted to implementation in WAM. The connection to constraint programming only existed by Prolog being a front-end to CP implementations. The software and logic architecture suggested by the equation was still to come. So much for Senner's suggestion of the development of Prolog as a relentless march forward. A decade went by before the great advances of the Nineties to logic programming were recognized as such and exploited accordingly. It must have been a dark decade in which XML, virtual machines, and constraint programming wandered around aimlessly, unaware of their true destiny.