Logic and Logic Programming: a personal account

Dale Miller
INRIA and LIX/Ecole Polytechnique, France

22 November 2005. Prepared for the
Newsletter of the Association for Logic Programming.

1 Introduction

I was asked to provide a personal perspective on some aspects of the development of logic programming. Since 2006 is the 20 year anniversary of my first paper on logic programming, I will use this invitation to reflect on those two decades and to list some lessons I have learned and some future directions for research.

In 1983, I tried to make the jump from being a student in Mathematics to a junior professor in Computer Science. I guessed that a good way to proceed was to apply what I had learned about logic and theorem proving as a graduate student to the topic of logic programming. Now 20 years later, I am still working at relating logic and computation, but now I commonly apply lessons that I have learned from computing to understanding logical principles.

2 λProlog

It might be hard to believe now, but automated deduction in higher-order logic was not a popular topic in the mid-1980's. At that time, there was just the odd paper about resolution, unification, Herbrand's theorem, etc, for higher-order logics, and, of course, there were few computer systems that implemented a higher-order logic. Basing logic programming on higher-order logic was a bit of a stretch.

The functional programming community was, however, making great strides then with all things higher-order: functions as first-class values, closures/thunks, higher-order types, polymorphism, etc. Since my PhD had been on higher-order logic, an obvious question for me was: could Prolog follow this modern trend and admit higher-orders and other forms of abstractions?

In 1985 and 1986, G. Nadathur and I started writing about and building prototype implementations for what we called λProlog: an extension of Prolog that contained polymorphic typing, higher-order quantification, and higher-order unification. By 1989, the language grew to its current size when it became based on hereditary Harrop formulas, thereby giving λProlog a logically supported notion of modular programming and abstract data-types.

While λProlog provides a clean, logical foundation for higher-order programming that interacts cleanly with its version of modular programming, it was another aspect of λProlog that caught the attention of its earliest users. λProlog was the first programming language to contain a simple and declarative treatment of expressions containing bound variables. Church in 1940 had described a higher-order logic that completely internalized bindings into terms, and equality, and deduction. Since λProlog was described as a subset of Church's logic, this treatment of binding comes for free. F. Pfenning and C. Elliott in 1988 gave this new style of syntactic representation the name "higher-order abstract syntax": to me, this new approach to computing on syntax is the most novel aspect of λProlog.

3 Sequent calculus

λProlog rests on higher-order intuitionistic logic, which was, at that time, a new logical foundations for logic programming. Such a shift in logical foundations was hard to sell 20 years ago. There was the fear then that if one leaves the well understood and familiar world of first-order classical logic for some other logic, then what was to stop the onslaught of "adjective pilings"? That is, if we admit that the center of the world might not be first-order classical logic, then how do we respond when someone claims that the world should revolve around "higher-order hyper-abductive modal fuzzy logic over finite constraints". This certainly sounded frightening to me as well, so I started to look for some discipline that could guide us.

In the mid-1980's, there was no general framework for understanding the role of logical constants and connectives in logic programming: instead, there was exactly one example in the literature, namely, first-order Horn clauses. Around this time, J. Jaffar, J.-L. Lassez, and others were developing the more general notion of constraints, but that work kept the centrality of Horn clauses while extending the way that first-order terms were constrained.

One could, of course, employ unrestricted logical formulas as logic programs and use a general purpose theorem prover for an interpreter. Such a design, however, misses the important point that searching for a proof in a logic programming language is a simple and structured activity. In particular, search is *goal-directed* and the role of the logic program is to prove atomic goal formulas by translating them, via backchaining, to other goal formulas. For example, SLD-resolution provides exactly that structure to proof search for Horn clauses. Unfortunately, resolution does not extend easily to other logics and its structure is not simple and transparent, in the sense that the resolution rule involves an interplay of quantification, conjunction, disjunction, negation, and equality, all at once.

The discipline that we finally settled on was to develop a theory for *proof search* using Gentzen's sequent calculus. We introduced the technical notion of *uniform proof* and defined a logic to be an *abstract logic programming language* if uniform proofs are complete. This definition provided a formalization of goal-directed search, in the sense that logical connectives in the goal were to be considered first; logic programs were considered only after a goal was reduced to an atomic formula. Such atomic goals where then established using a sequent calculus generalization of *backchaining*. The definitions of uniform proof and backchaining have been employed to classify some new logics as appropriate or inappropriate for logic programming.

Switching from viewing computation with logic programs as the search for a *resolution refutation* to the search for a *sequent calculus proof* is not only natural and pedagogic, it also frees logic programming from certain artifacts of first-order classical logic. Gone is the reliance on conjunctive normal forms, Skolemization, minimal models, and SLD-resolution. Instead, logic programming benefits from the more universal notions of cut-elimination, explicit structural rules, eigenvariables, and permutability of inference rules.

4 Linear Logic

In 1987, J.-Y. Girard introduced linear logic. For many of us working in computational logic, linear logic opened up broad new avenues for specifying computations.

Since linear logic has a particularly elegant sequent calculus proof system, it took little time for several people to propose ways to exploit linear logic to make logic programming more expressive. J.-M. Andreoli and R. Pareschi developed a linear logic programming language, LO, that provided an account of concurrency and synchronization. J. Hodas and I took a different approach and designed Lolli, which extended λProlog with the addition of the linear implication. Probably a half dozen other subsets of linear logic programming languages appeared in the first years of the 1990's and many of these made use of uniform proofs to help justify their designs.

Given that LO and Lolli were based on two different subsets of linear logic, I formed their union, named it Forum (this was 1993), and tried to prove that uniform proofs were complete for it. While trying to do this proof, I remembered that in his 1990 PhD thesis, Andreoli had discovered a normal form for proof search in linear logic. He had found that proof search in linear logic could be conducted in two phases: one decomposes certain connectives invertibly (the *asynchronous* phase) and the other decomposes the remaining connectives in a constrained and focused fashion (the *synchronous* phase). Together, these two phases describe *focused proofs*. As it turned out, a focused proof naturally corresponds to a uniform proof: the asynchronous decomposition phase corresponds to goal-reduction and the synchronous decomposition phase corresponds to backchaining. Since focused proofs are complete for linear logic, all of linear logic (as well as Forum) can be viewed as an *abstract logic programming language*. This fact also meant that the search for new linear logic programming languages stopped quickly.

Linear logic is a modular refinement of both classical and intuitionistic logics. Modularity means, for example, that if a Lolli program is mostly a λProlog program except that it occasionally uses the linear implication, then the operational behavior of that program can be seen as being mostly intuitionistic except for occasional linear behaviors. Only when genuinely linear features are used do the novel operational aspects of linear logic get involved. Thus linear logic provides a single framework for describing logic programs based on Horn clauses, hereditary Harrop formulas, LO, Lolli, and Forum.

5 Reasoning about logic programs

Having found that the search for more expressive linear logic programming languages was finished, my attention turned from design to a more basic question: Why should anyone care about new logic programming languages? When trying to motivate λProlog and linear logic programming languages, we did, of course, present lots of examples of tasks that we could program elegantly and more declaratively than in weaker logics. While elegance and declarativeness might be ends in themselves for a theoretician, they are not, of course, the full story for judging the value of a programming language. One thing that seemed clear when using a high-level, declarative programming language is that the correctness of programs should be easier to validate than if those programs were written in a more low-level and less declarative language. This suggests a research project: once you have written a program in a logic program language, how do you know that it is correct? How can you formally prove that the program, as an *artifact*, has such-and-such properties? Do the new logical primitives of rich logic programming languages help in showing correctness?

Formal reasoning about Horn clause programs has been studied by several researchers. In this setting, inductive theorem proving using conventional provers can establish many properties. On the other hand, it seemed quite difficult to use conventional provers to reason about intuitionistic or linear logic programs that employed higher-order abstract syntax. Several attempts at doing so generally provided particular solutions involving heavy encodings and unnatural proofs. Many people who were attracted to higher-order abstract syntax for its naturalness of expressing computation were disappointed that reasoning with that style of syntactic representation was unsatisfactory in conventional provers. It seemed to me that formalized reasoning of logic programs required some new designs for logic and theorem proving.

Since 1995, I've been working with several colleagues to develop a new logic in which one can reason directly on logic programs in order to infer properties that they might satisfy. To date, our conclusions can be summarized by saying roughly that if we keep object-level and meta-level logics clearly separated, consider (meta-level) logic programs as denoting their "if-and-only-if" completion, provide for induction and co-induction proof rules, and introduce the ∇ (nabla) quantifier for the treatment of generic judgments (for handling higher-order abstract syntax), then we can start getting rather nice proofs of formal properties of logic programs. The resulting logic, developed over several years by R. McDowell, A. Tiu, and myself, is called LINC, and is the centerpiece of an INRIA project called Parsifal. Within this project, we are attempting to automate various subsets of LINC in order to help infer properties of logic programs and the tasks that they specify.

6 Some lessons I learned

I repeat here a few lessons that I have learned about doing research in computational logic.

Examples are central. The most important thing that one should do before seriously proposing a new logic or a new programming language designs is to collect examples, a lot of examples. Examples give one confidence that a design might be worth all the work involved in proving theorems (and having readers understand the proofs) and in building implementations. I can, of course, support this claim with the following two examples.

The intuitionistic foundations of λProlog allows programs to grow monotonically during computation. This feature of λProlog provided both elegant examples and troubling counter-examples in four different application areas. While λProlog can model a database in which new facts can be added, it could not model naturally a database where facts could be changed or deleted. A theorem prover in λProlog can elegantly encode hypothetical reasoning by allowing the addition of new hypotheses as new logic programming clauses. Unfortunately, once an assumption is used it could not be deleted, and our elegant theorem provers always go into cycles repeatedly reusing the same assumption. λProlog could model object-oriented programming by encapsulating methods and state, but that state could not be changed: a switch could not be moved from on to off but could only be made to have both the on and off values. Finally, in parsing relative clauses, gap threading could be modeled directly using hypothetical reasoning except that there was no way to enforce the restriction that the hypothetical gap was actually used. These counter-examples, however, proved valuable since they became *examples* of just the kind of thing you could expect to solve using linear logic. In fact, the Lolli language managed to maintain the good aspects of the above examples while fixing the counter-examples.

The π-calculus has proved an invaluable example for me since I first heard R. Milner speak about it in 1991. This small, expressive calculus has many features that are challenging from the logic programming point-of-view. I initially tried to map the operational semantics of the π-calculus directly into linear logic: parts of the language seemed to be represented well but many other aspects of the full language could not be addressed in my initial approach. It was easy to show that the π-calculus's operational semantics could be written directly and elegantly as a λProlog program, but λProlog proved inadequate for doing the most basic reasoning about the π-calculus, namely, computing bisimulation. R. McDowell, C. Palamidessi, and I managed to use proof search to capture bisimulation but only for weaker languages (say, CCS) than the π-calculus. Finally, Tiu and I developed the ∇ quantifier and with that we could finally capture bisimulation for the full π-calculus. The π-calculus was a high quality example: it productively guided my research in logic programming for a number of years and its universal character also helped to ensure that our eventual solutions for it would also lead to successes in many other computation systems.

Prototype implementations help in finding and understanding examples. Given that examples are so important, prototype implementations are key for developing large and significant examples. Fortunately, there are now a number of high-level programming languages that make it possible to develop prototypes that are not much more complicated than a high-level definition of a system one might consider publishing. My own prototypes are usually written in the Teyjus implementation of λProlog or in some variant of ML.

Proof theory is an alternative to model theory for justifying logic programming. Clearly one needs to provide some structure to a language design if we are to call it a *logic* programming language. For a long time, that structure had come from model theory. In recent years, practitioners of Tarskian-style semantics and of category theory have developed sufficient semantic muscle to be able to build models that are sound and complete for almost any syntactic system. It would seem that one cannot simply refer to soundness and completeness results as the justification of a good design. As should be clear from my discussions above, proof theory can take a central role in helping to justify logic design. The demands on describing a logic within proof theory (for example, proving a cut-elimination theorem) seems to force deep connections between logic and its computational nature.

Elegance is not an option. Although it is quoted often, G. H. Hardy's words (from *A Mathematician's Apology*) are worth repeating here: "Beauty is the first test: there is no permanent place in the world for ugly mathematics." Work on logic programming should focus on staying declarative and being elegant: there is no permanent place for a hacked design. Of course, the hack might get something to work today, but it is important to find, understand, and exploit more universal lessons. Logic is a challenging framework for computation: much can be gained by rising to that challenge and trying to see the logical principles that can guide the development of computation systems and our ability to understand them.

7 Future directions

I find the topic of logic and computation healthy and exciting. Another 20 years of following where this topic takes me would be a pleasure. Below are a few directions I currently think are interesting future directions.

Model checking as deduction. There should be some nice consequences of identifying model checking with deduction. To the extent that this is possible, logic programming should provide avenues for not only implementing model checking but also extending it to more and more syntactic domains. Broader questions of logic should also help in reasoning about the correctness of model abstraction. Some of the performance engineering that goes into model checkers might also impact logic program implementations.

Better implementations of linear logic programming. It would be nice to see improvements to the implementations of linear logic programming languages. Proof search in full linear logic is quite complicated and a few researchers are looking at this problem. There might well be subsets of the full problem for which modular and low cost implementations could be built and possibly added to existing logic programming implementations.

Support for operational semantics. People who design and reason about specification languages and programming languages commonly employ high-level and declarative specification languages. For example, lexical analysis is described using finite automata and language syntax is described using grammars. The logic programming community has long been able to provide good quality implementations of both of these specification languages. The static semantics (such as typing) and dynamic semantics (such as evaluation) of programming languages are also described declaratively using inference rules in the style of *structured operational semantics*. The logic programming community can easily step into this domain of semantic specification and provide novel symbolic systems for executing, compiling, type checking, debugging, etc, other programming languages. Much recent work on operational semantics emphasizes program correctness and that is a topic to which implementors of *logic* should be able to contribute significantly.

A neutral approach to proof and refutation. Assume that I have a Horn clause program that doesn't contain any loops. If I ask for a proof of the (closed) goal formula G, then Prolog will terminate and report either "yes" or "no". The "yes" answer means that it has found a proof of G while the answer "no" usually means that no proof of G is found. If we employ the "if-and-only-if" completion of a logic program, however, the "no" response implies that there is a proof of $\neg G$ (from the completed program) and the full, finite failing search is, in fact, the proof of that negation. Thus, Prolog did one computation and depending on how that computation ended, it found a proof of G or of $\neg G$. This description of Prolog is a challenge to the proof search setting that I have described above. One can argue that Prolog was, in fact, *neutral* to whether or not it was doing proof or refutation, only deciding at the end of the computation. Proof search, as we currently understand it, is not neutral in this way: we are required to pick a formula or its negation prior to starting proof search. An interesting question is whether or not this "neutral approach to proof and refutation" can be applied to larger fragments of logic. Game theory might provide the appropriate generalization: for example, a winning strategy might provide a proof of G while a winning counter-strategy might provide a proof of $\neg G$.