Influences between logic programming and proof theory

by Dale Miller, Inria-Saclay, January 1, 2018

The earliest and most popular use of logic in computer science views computation as something that happens independent of logic: e.g., registers change, tokens move in a Petri net, messages are buffered and retrieved, and a tape head advances along a tape. Logics (often modal or temporal logics) are used to make statements *about* such computations. Model checkers and Hoare proof systems employ this *computation-as-model* approach.

Early in the 20th century, some logicians invented various computational systems, such as Turing machines, Church's λ-calculus, and Post correspondence systems, which were shown to all compute the same set of *recursive functions*. With the introduction of high-level programming languages, such LISP, Pascal, Ada, and C, it was clear that any number of ad hoc computation systems could be designed to compute these same functions. Eventually, the large number of different programming languages were classified via the four paradigms of *imperative*, *object-oriented*, *functional*, and *logic* programming. The latter two can be viewed as an attempt to make less ad hoc computational systems by relying on aspects of symbolic logic. Unlike most programming languages, symbolic logic is a formal language that has well-defined semantics and which has been studied using model theory [17], category theory [8, 9], recursion theory [5, 6], and proof theory [3, 4]. The *computation-as-deduction* approach to programming languages takes as its computational elements objects from logic, namely, terms, formulas, and proofs. This approach has the potential to allow the direct application of logic's rich metatheory to proving properties of specific programs and entire programming languages.

The influence of proof theory on logic programming

The first thing that proof theory has offered to the logic programming paradigm is a clean and straightforward means to differentiate itself from functional programming. From the proof theory perspective, functional programs correspond to proofs (usually in natural deduction), and computation corresponds to *proof normalization*: that is, programs correspond to non-normal proofs and computation is seen as a series of normalization steps (using either β-convergence or cut-elimination). This programs-as-proof correspondence is known as the Curry-Howard isomorphism [16]. In contrast, *proof search* is a good charactorization of computation in logic programming. Here, quantificational formulas are used to encode both programs and goals (think about the rules and queries in database theory). Sequents are used to encode the state of a computation and (cut-free) proofs are used to encode computation traces: changes in sequents model the dynamics of computation. Although cut-elimination is not part of computation, it can be used to reason about computation. Also. the proof-theoretic notions of inference rule, schematic variable, proof checking, and proof search are directly implementable. The proof-normalization and the proof-search styles of computational specification remain distinct even in light of numerous recent developments in proof theory: for example, linear logic, game semantics, and higher-order quantification have all served to illustrate differences and not similarities between these two styles.

In the early days of logic programming, say from 1972-1985, the entire logic programming paradigm was described using just one particular logic: that of the first-order Horn theories in classical logic [7]. Also, the process of proof search was paradoxically described as refutation (using the resolution rule). Gentzen invented the sequent calculus for the explicit purpose of unifying proofs in classical and intuitionistic logic. Girard showed that the sequent calculus can naturally account for linear logic proofs as well [14, 13]. As a result of this unity, the sequent calculus provided logic programming a natural framework in which proof-search could be described for much richer logics (first-order and higher-order versions of classical, intuitionistic, and linear logics) than that underlying Prolog.

Another feature of the sequent calculus is its support for abstraction: that is, it provides mechanism for allowing some aspects of a program's specification to be hidden while other aspects are made explicit. The cut-elimination theorem can be used to match abstractions from actual implementations. In programming language terminology, such abstractions provide logic programming with modularity, abstract datatypes, and higher-order programming. The use of abstractions can significantly aid in establishing formal properties of programs.

The influence of logic programming on proof theory

Logic programming has also influenced proof theory. To explain Prolog's operational behaviour, resolution was restricted to SLD-resolution [2]. Similarly, sequent calculus proofs were restricted to *goal-directed* (or *uniform*) proofs in order to provide a general framework for logic programming. Such goal-directed programs were structured with alternating phases of inference rules: one phase reduced the goal formula (using right-introduction rules), and one phase performed backchaining steps (using left-introduction rules) [11, 14]. The resulting sequent calculus framework provided justifications for the design of logic programming languages within both classical and intuitionistic logics. After Girard's introduction of linear logic [4] in 1987, Andreoli defined *focused* linear logic proofs [1] in which cut-free proofs were restricted to alternating phases but this time, such proofs applied to *all* of linear logic. The completeness of focused proofs provided a complete analysis of which subsets of linear logic made good logic programming languages [13]. Several subsequent efforts have been made to provide flexible, focused proof systems for classical and intuitionistic logic all of which are captured by the LKF and LJF focused proofs system [10].

Logic programming makes heavy use of term structures and quantification. Thus a comprehensive analysis of the proof theory of logic programs forced proof theory to look beyond its usual concentration on propositional connectives and to look hard at quantification. For example, Skolemization is seldom a natural and efficient means for addressing quantifier alternation in a proof theory setting. Instead, the sequent calculus supports the concept of *mobility of binders* [12] in which term-level bindings are capable of moving to formula-level quantifiers which are then capable of moving to proof-level eigenvariables. Furthermore, the proof-theoretic treatment of inductive reasoning applied to logic programming specifications has lead to the appearance of the ∇-quantifier that is capable of capturing *generic* quantification [15].

References

[1] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[2] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. of the ACM*, 29(3):841–862, 1982.

[3] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[5] Kurt Gödel. On formally undecidable propositions of the principia mathematica and related systems. I. In Martin Davis, editor, *The Undecidable*. Raven Press, 1965.

[6] S. C. Kleene. A theory of positive integers in formal logic. part I. *American J. of Mathematics*, 57(1):153–173, January 1935.

[7] Robert A. Kowalski. Logic programming in the 1970s. In *Logic Programming and Nonmonotonic Reasoning, LPNMR 2013*, LNCS 8148, pages 11–22. Springer, 2013.

[8] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[9] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings National Academy of Sciences USA*, 50:869–872, 1963.

[10] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

[11] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

[12] Dale Miller. Bindings, mobility of bindings, and the ∇-quantifier. In the Proceedings of CSL 2004, pp, 24, LNCS 3210.

[13] Dale Miller. Overview of linear logic programming. In *Linear Logic in Computer Science*, *London Mathematical Society Lecture Note* 316, pages 119–150, 2004.

[14] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *A. of Pure and Applied Logic*, 51:125–157, 1991.

[15] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[16] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic*. Elsevier, 2006.

[17] A. Tarski. Contributions to the theory of models I,II. *Indagationes Mathematicae*, 16:572–588, 1954.