Influences between logic programming and proof theory

Dale Miller

Inria Saclay & LIX, École Polytechnique
Palaiseau, France

HaPoP 2018, Oxford, UK
23 March 2018

Influences between logic programming and proof theory

Dale Miller

Inria Saclay & LIX, École Polytechnique
Palaiseau, France

HaPoP 2018, Oxford, UK
23 March 2018

N.B.: I am neither a historian nor a philosopher but a participant in some of what I describe.

Logic in Computer Science

Logic has a clear and continuing impact on Computer Science. That impact is probably greater than for Mathematics.

There are major journal that publishes in this topic.

- The ACM Transactions on Computational Logic
- Logical Methods in Computer Science
- Journal on Automated Reasoning

There are several major conferences (LICS, CSL, CADE, IJCAR, FSCD) and many workshops.

Logic in Computer Science

Logic has a clear and continuing impact on Computer Science. That impact is probably greater than for Mathematics.

There are major journal that publishes in this topic.

- The ACM Transactions on Computational Logic
- Logical Methods in Computer Science
- Journal on Automated Reasoning

There are several major conferences (LICS, CSL, CADE, IJCAR, FSCD) and many workshops.

This topic also has its own "Unreasonable Effectiveness" paper:

"On the Unusual Effectiveness of Logic in Computer Science" by Halpern, Harper, Immerman, Kolaitis, Vardi, and Vianu (Bulletin of Symbolic Logic, ASL, June 2001).

Roles of Logic in CS

- computation-as-model
- computation-as-deduction
  - proof normalization (functional programming)
  - proof search (logic programming)

The *computation-as-model* role is the most popular use of logic in computer science: 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 are used to make statements *about* such computations.

Roles of Logic in CS: computation-as-deduction

Most programming languages (C, C++, Java, etc) are based on ad hoc principles and are hard to formalize.

Even when a formalization is achieved, there are no alternative view of that meaning. "The meaning of the C-program is whatever the gcc compiler does with it."

The *computation-as-deduction* approach uses pieces of the syntax of logic—formulas, terms, types, and proofs—directly as elements of computation.

There are multiple perspectives to, say, first-order classical logic: model theory, category theory, and multiple proof systems (sequent, tableaux, resolution refutation, etc).

Proof normalization vs proof search

In this setting of *computation-as-deduction*, there are two rather different approaches to modeling computation.

Proof normalization views proofs as programs and views proof normalization (β-reduction or cut-elimination) as computation. This use of logic provides a foundation to *functional programming*. The Curry-Howard correspondence is part of this approach.

Proof search views computation as the construction of a cut-free proof of sequents such as $\mathcal{P} \vdash G$ involving a program (a set of assumptions) $\mathcal{P}$ and a query G. This approach provides a foundation for *logic programming*. Here, cut-elimination can be used to reason about computation.

Briefly noted

A brief bibliography

- Gentzen, 1935, sequent calculus, cut-elimination
- Church, 1940, higher-order logic based on the simply typed λ-calculus
- Girard, 1987, linear logic

Two communities

- Structural proof theory: Prawitz, Schroeder-Heister, Negri, etc
- Logic programming: Kowalski, van Emden, Apt, etc

PT on LP: stuck on one example

In the beginning (1972-1985), the logic programming paradigm was described using just one particular logic:

first-order Horn theories in classical logic.

Prolog and Datalog are based on this fragment of logic.

The theory behind the interpretation of these languages was based on SLD-refutation (not proof).

The use of Robinson's resolution calculus as the foundations of logic programming forced

- the use of classical (first-order) logic and
- the elimination of quantifier alternations (via Skolemization).

PT on LP: switching from resolution to proof

Gentzen's sequent calculus provided an alternative to refutation.

Instead of arguing that

$$\mathit{cnf}(\mathit{skolem}(\mathcal{P})), \neg G \quad \text{leads to the empty clause } \Box,$$

one instead can attempt to find a cut-free proof of

$$\mathcal{P} \vdash G.$$

Now first-order quantification could be generalized to higher-order and classical logic could be replaced by intuitionistic and linear logics.

The sequent calculus provided a framework for logic programming to grow and mature.

PT on LP: from one example of LP to a framework

SLD-resolution was replaced by the more general notion of *goal-directed search* (in the sequent calculus).

An *abstract logic programming language* was a logic and set of theories where goal-directed search was complete.

In intuitionistic logic, *hereditary Harrop formulas* greatly generalized Horn clauses as a foundation for logic programming.

Higher-order versions of both Horn clauses and hereditary Harrop formulas (relying on Church's 1940 STT framework and results of Andrews, Huet, etc).

PT on LP: logical foundations for abstractions in LP

Sequent calculus, especially for intuitionistic logic, allows for explaining modular programming, abstract datatypes, and higher-order programming.

Various vendors of Prolog added some of these abstraction mechanism in different, ad hoc fashions but formal properties have seldom been studied.

λProlog, which was designed on top of higher-order hereditary Harrop formulas, provided logically motivated approaches to all of these abstractions/hiding mechanisms. Formal properties follow directly from cut-elimination.

PT on LP: linear logic provided new logic programs

Girard's linear logic (1987) adds expressiveness to classical and intuitionistic logics. It's integration with the sequent calculus is immediate and natural.

A number of *linear logic programming* were proposed. For example, *Lolli* and *Forum* provided extensions of λProlog.

Forum is actually a logic programming presentation of all of linear logic.

These languages have found use in treating state, concurrency, and various features in natural language parsing.

Some influences of logic programming on proof theory

The forces on a programming paradigm to evolve are strong: more efficient implementations; more expressiveness; more avenues for formal reasoning; better interoperatibility.

There are always short-term fixes, but:

> *"Beauty is the first test: there is no permanent place in the world for ugly mathematics."*
> *G. H. Hardy, A Mathematician's Apology*

The hack might get something to work today but they should not be permanent.

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 find logical principles behind such demands.

LP on PT: Focused proof systems

The identification of goal-direct proof was a challenge to proof theory.

The *uniform proofs* of M, Nadathur, Pfenning, Scedrov (1991) was a partial response.

Andreoli (1992) provided a satisfactory response for linear logic by inventing *focused proofs* (certain kinds of sequent calculus proofs).

Focused proofs have been generalized to classical and intuitionistic logics (Liang & M, 2009).

Focused proofs are the most important innovation in structural proof theory since the invention of linear logic.

LP on PT: Terms and term-level bindings matter too

Most proof theory concerns propositional logic connectives.

Some proof theory addressed "second-order propositional logic": e.g., $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$.

Church's 1940 *Simple Theory of Types* used typed λ-terms to represent higher-order quantification and term-level bindings (description/choice operators, function definitions).

The merging of Church with Gentzen needs to have bindings integrated into the sequent calculus.

$$\Sigma : \Gamma \vdash \Delta$$

Here, Σ is the *binding* of eigenvariables over the two (multi)sets of formula Γ and Δ.

LP on PT: Mobility of binders

During proof search, binders can be *instantiated* (using β implicitly)

$$\frac{\Sigma : \Delta, \mathit{typeof}\ c\ (\mathit{int} \to \mathit{int}) \vdash C}{\Sigma : \Delta, \forall\alpha(\mathit{typeof}\ c\ (\alpha \to \alpha)) \vdash C}\ \forall L$$

They also have *mobility* (they can move):

$$\cfrac{\cfrac{\Sigma, x : \Delta, \mathit{typeof}\ x\ \alpha \vdash \mathit{typeof}\ \lceil B \rceil\ \beta}{\Sigma : \Delta \vdash \forall x(\mathit{typeof}\ x\ \alpha \supset \mathit{typeof}\ \lceil B \rceil\ \beta)}\ \forall R}{\Sigma : \Delta \vdash \mathit{typeof}\ (\lambda x.B)\ (\alpha \to \beta)}$$

In this case, the binder named x moves from *term-level* (λx) to *formula-level* ($\forall x$) to *proof-level* (as an eigenvariable in Σ, x).

LP on PT: a new quantifier ∇

There is no (capture avoiding) substitution for w so that $(\lambda x.x = \lambda x.w)$: that is, the following should be provable.

$$\vdash \forall w \neg (\lambda x.x = \lambda x.w).$$

LP on PT: a new quantifier ∇

There is no (capture avoiding) substitution for w so that $(\lambda x.x = \lambda x.w)$: that is, the following should be provable.

$$\vdash \forall w \neg (\lambda x.x = \lambda x.w).$$

The ξ inference rule is usually written as

$$\frac{\forall x.t = s}{\lambda x.t = \lambda x.s} \quad \text{and} \quad (\forall x.t = s) \equiv (\lambda x.t = \lambda x.s)$$

That equivalence leads to the formula $\forall w \neg \forall x.x = w$ which cannot be proved since it is false in a singleton domain.

LP on PT: a new quantifier ∇

There is no (capture avoiding) substitution for w so that $(\lambda x.x = \lambda x.w)$: that is, the following should be provable.

$$\vdash \forall w \neg (\lambda x.x = \lambda x.w).$$

The ξ inference rule is usually written as

$$\frac{\forall x.t = s}{\lambda x.t = \lambda x.s} \quad \text{and} \quad (\forall x.t = s) \equiv (\lambda x.t = \lambda x.s)$$

That equivalence leads to the formula $\forall w \neg \forall x.x = w$ which cannot be proved since it is false in a singleton domain.
The solution is a new quantifier ∇ which revises the ξ equivalence

$$(\nabla x.t = s) \equiv (\lambda x.t = \lambda x.s)$$

and yields the theorem $\vdash \forall w \neg \nabla x.x = w$. Negation separates universal quantification into *extensional* $\forall$ and *generic* ∇.

Conclusion: Significant transfer between two communities

Proof theory

- Provided: deep designs and results concerning proofs
- Received: a new normal form of proof (goal-directed); a push to understand quantification and binding better; a new relevance.

Logic programming

- Provided: new phenomena that needed to be explained (modules, bindings, etc)
- Received: a framework; several new and more expressive languages; a certain depth.