A Survey of the Proof-Theoretic Foundations of Logic Programming

Dale Miller

Inria-Saclay & LIX, Ecole Polytechnique, Palaiseau, France

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Draft: 29 October 2021

Abstract

Several formal systems, such as resolution and minimal model semantics, provide a framework for logic programming. In this paper, we will survey the use of *structural proof theory* as an alternative foundation. Researchers have been using this foundation for the past 35 years to elevate logic programming from its roots in first-order classical logic into higher-order versions of intuitionistic and linear logic. These more expressive logic programming languages allow for capturing stateful computations and rich forms of abstractions, including higher-order programming, modularity, and abstract data types. Term-level bindings are another kind of abstraction, and these are given an elegant and direct treatment within both proof theory and these extended logic programming languages. Logic programming has also inspired new results in proof theory, such as those involving polarity and focused proofs. These recent results provide a high-level means for presenting the differences between forward-chaining and backward-chaining style inferences. Anchoring logic programming in proof theory has also helped identify its connections and differences with functional programming, deductive databases, and model checking. [To appear in *Theory and Practice of Logic Programming (TPLP)*.]

1 Introduction

There are two broad approaches to relating logic with computational systems (Miller 2006). On the one hand, there is the *computation-as-model* approach in which computations determine models represented via mathematical structures containing such items as nodes, transitions, and state. Logic is used in an external sense to make statements *about* those structures. That is, computations are models, and logical expressions are evaluated over such models. Intensional operators, such as the modal operators of temporal and dynamic logics or the triples in Hoare logic, are often employed to express propositions about the state change. This use of logic to represent and reason about computation is probably the oldest and most successful use of logic with computation.

On the other hand, the *computation-as-deduction* approach uses pieces of logic's syntax (e.g., types, terms, formulas, and proofs) directly as elements of the specified computation. There are two different approaches to modeling computation in this much more rarefied setting depending on how they use *proofs*. The *proof normalization* approach views the state of a computation as a proof term and the process of computing as normalization (via β-reduction or cut-elimination). This approach to computing is based on the *Curry-Howard correspondence* (Curry 1934; Howard 1980; Sørensen and Urzyczyn

2006) and can provide a theoretical framework for functional programming (Martin-Löf 1982). The *proof search* approach views the state of a computation as a *sequent* (a particular structured collection of formulas) and the process of computing as the search for a proof of a sequent: the changes that take place in sequents capture the dynamics of computation. In the broadest sense, proof search can be a foundation for interactive and automatic theorem proving, model checking, and logic programming. This paper shall survey how the proof search interpretation of the sequent calculus has been used to give a foundation to logic programming.

Unifying the two most foundational perspectives of logic—model theory and proof theory—was the goal of some of the earliest work on the foundations of logic programming. However, these two perspectives on logic have their own concerns and internal structure and results. As a result, divergence appeared when these two perspectives were used to motivate new designs and theories about programming with logic. Taking models as primary, along with the direct treatment of negation available in model theory, has led to new logic programming languages, such as the *answer set programming* approach to declarative programming (Lifschitz 2008; Brewka et al. 2011) (see also Section 10.6). These developments have led to new applications of logic in subjects such as databases, default reasoning, planning, and constraint solving. In this paper, we survey, instead, the development of new logic programming language designs and theories where proof theory is taken as primary. Some application areas of these designs have been type systems, proof assistants, proof checking, and the specification of operational semantics.

Symbolic logic is an appealing place to define a high-level programming language for several reasons. First, it is a well-studied and mature formal language. As a result, it has rich properties that enable manipulating and transforming its syntax in meaning-preserving ways. Such manipulations include substitution into quantified expressions, the unfolding of recursive definitions, and conversion to normal forms (such as conjunctive normal form or negation normal form). Second, logics generally have multiple ways to look at what a theorem is. For example, soundness and completeness results allow us to identify theorems as those formulas that have a proof and are true in all models. Finally, even for logics where model-theoretic approaches are less commonly used, such as linear logic (Girard 1987), other deep principles, for example, cut-elimination, are available.

Given that we choose to work with symbolic logic, how should we connect logic with logic programming? Clearly, the logical foundation of Prolog—*first-order Horn clauses*—should be taken as an example of logic programming. Nevertheless, the notion of *proof search* is a broad term, including, for example, interactive and automated theorem provers where considerable cleverness is needed to discover lemmas and inductive invariants. Obviously, including the discovery of lemmas and invariants should not be expected of an interpreter or compiler of a logic program. Thus, it seems necessary to draw a line between proof search in full logic and some simpler, automatable subset of logic.

2 The need for more expressive logic programming

Horn clauses are formulas of the form

$$\forall x_1 \ldots \forall x_n [A_1 \wedge \ldots \wedge A_m \supset A_0] \qquad (n, m \geq 0). \tag{1}$$

Here, the symbol A (with or without subscripts and superscripts) is used as a syntactic variable ranging over atomic formulas. Notice that this formula can also be written without conjunctions as

$$\forall x_1 \ldots \forall x_n [A_1 \supset \ldots \supset A_m \supset A_0],$$

where the bracketing of implications is to the right. In both cases, if $m = 0$ then we do not write the implication. A simple generalization of Horn clauses can be given by the following grammar-like description of two classes of formulas using the syntactic variables G (for *goal* formulas) and D (for *definite* formulas).

$$G := A \mid \top \mid G \wedge G \mid \bot \mid G \vee G \mid \exists x.G \tag{2}$$

$$D := A \mid G \supset D \mid \top \mid D \wedge D \mid \forall x.D \tag{3}$$

Here, G-formulas are freely generated from atomic formulas, $\top$ (true), $\bot$ (false), conjunction, disjunction, and existential quantifiers. A D-formula is a generalization of Horn clauses and these are such that any subformula occurrence to the left of an implication is a G-formula. Using simple equivalences (which hold in classical and intuitionistic logics), it is easy to show that a D-formula is logically equivalent to a conjunction of formulas that are of the form (1) above.[1]

While the logic programs that can be written using first-order Horn clauses are Turing complete (Tärnlund 1977), programming languages, such as Prolog, based on Horn clauses have various weaknesses that have been pointed out in the literature. A list of some of these shortcomings is below.

Constraints: The usual approach to data structures in Prolog encodes them as first-order terms using uninterpreted symbols. Occasionally, certain domains contain values that are much better handled by special-purpose algorithms instead of unification and strict syntactic equality. Constraint logic programming (Jaffar and Lassez 1987) is a general framework for organizing the treatment of such domains.

Negation-as-failure: The simplest theories of Horn clauses do not include negation. Different versions of negation, such as negation-as-failure (Clark 1978), have been added to most versions of Prolog.

Control of search: Prolog implements depth-first search, which provides a natural procedural interpretation of many Horn clause specifications while providing expensive or non-terminating interpretations for other specifications. Prolog has evolved several control mechanisms, such as `!` (cut), ancestor checking, and tabled deduction.

Side-effects: For a specification language to become a programming language, it seems necessary to accommodate primitives for side-effects and communications with other components of modern computer systems. Primitives have been added to Prolog to allow side effects (e.g., `assert` and `retract`) and input and output.

Abstraction mechanisms: The logic behind Prolog does not directly support modern notions of abstractions, such as modules, abstract data types, higher-order programming, and binding structures. Various extensions to Prolog addressing modular programming have been developed (ISO.org 2000) and incorporated into most modern implementations of Prolog.

[1] Throughout this paper, the equivalence (in classical, intuitionistic, or linear logic) of two formulas B and C means that the two entailments $B \vdash C$ and $C \vdash B$ are provable (in the respective logic).

As this list shows, the development of programming language features on top of Horn clauses has resulted in adding more to an exciting but weak core logic setting. The work that this survey explores takes a different perspective to logic programming language design. Instead of working with a simple and weak foundation, proof theory has been used to imagine large and more expressive logical foundations, even going as far as adding higher-order quantification and linear logic connectives. Given the generality of such a large framework, it is doubtful that the entire framework can be effectively implemented. However, the purpose of such imagining is not to provide the foundations of a single, grand, practical logic programming language but rather to develop a framework in which many different sublanguages can be extracted (only one of which consists of Horn clauses). Such sublanguages would inherit some properties of the larger framework, but their more narrow focus might allow for practical implementations. By way of analogy, consider the problem of building parsers. Context-free grammars (CFG) provide an important framework for declaratively describing the structure of some languages. Since that framework is flexible and high-level, general-purpose parsers are expensive: for example, the Earley parser has $O(n^3)$ complexity cost for strings of length n (Earley 1970). Since this complexity is too high for use in, say, compilers, many subsets of the general CFG framework have been developed, such as the $LR(k)$ and $LALR(k)$ grammars, which describe fewer languages but have parsers with better time and space complexity (Aho et al. 2007). As we shall note in Section 10, several subsets of the most general, abstract logic programming framework have been identified and implemented in different application areas.

3 Some formal frameworks for logic programming

A good formal framework for logic programming should satisfy some properties, such as those listed here.

1. It should provide multiple and broad avenues for reasoning about logic programs. We do not need new Turing machines because we do not need more specification languages that obviously compute but which do not come with support for addressing the correctness of specifications.
2. It should allow for the positioning of the logic programming paradigm among other programming and specification paradigms.
3. It should provide for a range of possible designs, leading to logic programming languages that go beyond the one acknowledged example based on Horn clauses. Hopefully, these new designs would address some of the shortcomings outlined in the previous section.

The following section focuses on the use of *structural proof theory* as a foundation for logic programming. The rest of this section describes three other popular approaches to the formal foundations for logic programming.

3.1 Resolution

Following Robinson's introduction of the resolution refutation method for automating first-order logic (Robinson 1965), several researchers developed strategies to tame the search for refutations. One of these strategies is linear resolution, which was developed independently by Loveland (1970), Luckham (1970), and Zamov and Sharonov (1971). After the first Prolog system was developed at the University of Aix-Marseilles in 1972 (Colmerauer and Roussel 1993), Kowalski (1974) formalized its operational behavior as linear resolution. Later, Apt and van Emden (1982) named this particular style of resolution *SLD-resolution* ("Selective Linear Definite clause resolution") and proved it to be complete when restricted to Horn clauses.

As a framework, resolution has been used to provide a treatment of some extensions of Prolog. For example, Clark (1978) introduced the *if-and-only-if* completion of Horn clauses. He showed how the failure of an exhaustive and finite search for an SLD-resolution refutation could be used to justify a proof of a negated goal. Clark's extended refutation procedure, now called SLDNF, received various descriptions and correctness proofs (see, for example, the papers by Apt and Bol (1994) and by Apt and Doets (1994)) and has been used to extend logic programming to include all of first-order classical logic (Lloyd and Topor 1984). Resolution also allows for a simple approach to the treatment of constraints and their flexible scheduling (Huet 1973). Minker and Rajasekar (1990) specialized resolution to serve as a proof strategy for disjunctive logic programming. Loveland's Near-Horn Prolog (Loveland 1987; Loveland and Reed 1991) was also described using resolution, although it was eventually given a description using sequent calculus proofs (Nadathur and Loveland 1995; Nadathur and Loveland 1998) in the style we shall see in Section 5.

Although resolution refutations had some successes as a framework for logic programming, this framework has been problematic for at least two reasons. First, it generally relies on normal forms, such as conjunctive normal form, negation normal form, and Skolem normal form. Such normal forms are not generally available outside of first-order classical logic: in particular, these normal forms are not sound for intuitionistic logic. Secondly, resolution is pedagogically flawed since it forces the attempt to *prove* the goal G from the program $\mathcal{P}$ into the attempt to *refute* the set of formulas $\mathcal{P} \cup \{\neg G\}$; switching from proving to refuting is unfortunate, unintuitive, and, as we shall see, unnecessary. As the author has argued elsewhere (2021), the use of Skolemization to simplify the structure of quantifiers in formulas appears to be the dominant reason for early automated theorem proving systems to rely on building refutations instead of proofs. Since structural proof theory provides an alternative to Skolemization, that framework can rely on proving instead of refuting.

3.2 Model theory

Given the success of denotational semantics to provide a mathematically precise notion of meaning for various programming languages (Scott 1970; Stoy 1977) and given that model theory for first-order classical logic was a well-developed topic before the advent of logic programming, it was natural to consider using model theory as a semantics for logic programming.

Apt, van Emden, and Kowalski provided the first steps to building such a semantics for logic programming. They connected SLD-resolution to fixed-point operators on models represented by sets of atomic formulas. In particular, the least-fixed point model semantics was shown to characterize provable atomic formulas, while negation-as-failure was shown to relate to the greatest-fixed point model (Apt and van Emden 1982; van Emden and Kowalski 1976). Model theory has also been used to provide various formal definitions of negation-as-failure, including *well-founded semantics* (Van Gelder et al. 1991) and *stable models* (Gelfond and Lifschitz 1988).

Model theory can sometimes be used to provide an equivalent perspective on provability. In particular, the familiar soundness and completeness theorems state that provable statements are exactly the valid statements. Such a result can convince us that a given proof system is not, in fact, ad hoc, inconsistent, or missing inferences or axioms. Such confidence indeed arises from the earliest completeness theorems, such as the ones given by Gödel (1930) and Henkin (1949). Today, however, experts in model theory and category theory have sufficient "muscle" so that they can build complicated and ad hoc semantic domains. As a result, soundness and completeness theorems are not as compelling as they once were. Fortunately, proof theory comes with its own principles, such as, for example, the cut-elimination theorem, which helps to rule out ad hoc and inconsistent inference principles.

3.3 Operational semantics

Semantics can be given for logic programming by providing a mathematical description of the language's behavior. One such approach has been to use *abstract machines*, such as the Warren Abstract Machine WAM (Warren 1983; Aït-Kaci 1991), to describe the behavior of logic programs. Such machines can be taken as formal models when they are given a formal specification (Börger and Rosenzweig 1995b).

A few high-level and formal specifications of parts of Prolog's operational semantics have been developed starting a couple of decades ago and using different techniques: e.g., Andrews (1997) used a combination of a multi-valued logic and a transition system, Li (1994) used the π-calculus, and Börger and Rosenzweig (1995a) used evolving algebras. Such approaches to specification have the advantage that they can describe the actual behavior of Prolog implementations when they need to deal with features such as the cut `!` control operator and the `assert` and `retract` predicates. Such features are difficult or impossible to address using resolution refutations or model theory.

Since these specification styles are formal, any attempt to reason about them also certainly requires using proof assistants. These specifications are used to address the question "How do we implement a language?" and not the more general question "What language should we implement?" While the former question is important, we shall focus on a framework that addresses the latter question.

4 The trajectory of proof theory investigations

The term "proof theory" is often used in the logic programming literature to refer to some characterization of provability $\vdash$ in contrast to validity $\models$. However, in many texts, provability is characterized indirectly using resolution refutations. In this paper, the term "proof theory" is used exclusively to refer to the systems and methods introduced by Gentzen in his famous paper (1935). In that paper, Gentzen introduced both natural deduction and the sequent calculus and proved the cut-elimination theorem for classical and intuitionistic logics. Gentzen's proof systems have been applied in many different settings during the past several decades. In mathematics, they have been used to prove the consistency of various logical and arithmetic systems (Gentzen 1935); in logic, they have been used to define various modal logics (Ono 1998; Wansing 2002); in linguistics, they have been used to describe the structure of sentences (Lambek 1958); and in computational logic, they have been used to provide the formal setting for discussing both computing via proof normalization and proof search (see Section 1). We shall be particularly interested in using Gentzen's proof systems to analyze the syntax and structure of proofs themselves. With this emphasis, this topic is often called *structural proof theory*. Good background material on this topic can be found in the papers by Gallier (1986), Girard et al. (1989), Buss (1998), and Negri and von Plato (2001). Since the λ-calculus will also be associated with our discussion of proof theory, the reader unfamiliar with the basics of the λ-calculus can find good background material in the work of Barendregt (1984), Huet (1975), Miller and Nadathur (2012), and Barendregt et al. (2013).

The following list of key applications of structural proof theory to logic programming helps to provide an outline of the rest of this survey.

1. In Section 5.2, we position logic programming within the sequent calculus instead of resolution and then describe the nature of *goal-directed search* and *backward chaining* using sequent calculus inference rules.
2. Given that sequent calculus proof systems were known for first-order and higher-order classical and intuitionistic logics, the first proof-theoretic extensions of logic programming were investigated in these logics. For example, developing proof search results within higher-order intuitionistic logic provided logic programming with various forms of abstractions, including higher-order programming, modules, and abstract data types (see Sections 5.3 and 5.4). The sequent calculus also enables a new treatment of binding structures (within terms, formulas, and proofs): this treatment is described in Section 6.
3. The appearance of linear logic provides new and sometimes surprising avenues for extending logic programming to settings involving stateful and concurrent computations (see Section 7).
4. The proof theory of linear logic introduced the notions of *polarity* and *focused proofs* (see Section 8). When these notions are applied to logic programming, they allow for extending the notion of goal-directed proof. These notions also provide an elegant description of both forward-chaining and backward-chaining inference.

5 Intuitionistic logic and proof search

In the 1980s, there were some early attempts to use various proof systems as frameworks for logic programming based on extended versions of Horn clauses. For example, Hagiya and Sakurai (1984) used Martin-Löf's theory of iterated inductive definitions (1971) to describe Horn clause reasoning and negation-as-failure. There were also several attempts to extend Prolog to full first-order logic. In particular, Bowen (1982) described how sequent calculus and unification could be merged; Haridi and Sahlin (1983) described an implemented proof system using natural deduction; and Cellucci (1987) proposed using tableaux proof systems for the specification of logic programming.

In the second half of the 1980s, several researchers discovered roles for intuitionistic logic within computational logic that were not directly related to the Curry-Howard correspondence (briefly described in Section 1). Instead, these roles supported the proof-search paradigm. These discoveries, listed below, were made nearly simultaneously and largely independently.

- Gabbay and Reyle developed N-Prolog, an extension to Prolog with hypothetical goals (Gabbay and Reyle 1984; Gabbay 1985).
- The λProlog logic programming language by Nadathur and the author (1988; 2012) lifted the Prolog language to higher-order intuitionistic logic. This logic provided hypothetical and generic reasoning as well as higher-order programming for logic programming (Miller et al. 1991).
- McCarty (1988a; 1988b) used Kripke model semantics of intuitionistic logic to study an extension of logic programming that supported hypothetical reasoning.
- Paulson (1989) used natural deduction and intuitionistic logic to provide a framework for the generic theorem prover at the core of the Isabelle prover. Some design and implementation issues in that prover are closely related to design and implementation issues in the λProlog system.
- Hallnäs and Schroeder-Heister (1990; 1991) also explored a logic programming interpretation of hypothetical reasoning using the proof theory of intuitionistic logic.
- Mints and Tyugu (1990) used propositional intuitionistic logic to design and automate their PRIZ programming system.

Also, during this period, the dependently typed λ-calculus LF (Harper et al. 1993) was proposed as a framework for describing proof systems for intuitionistic logic: it was also given a λProlog-inspired implementation within the Elf system (Pfenning 1989).

In the *proof search* setting, the successful completion of a (non-deterministic) computation is encoded by a cut-free proof. Here, proof normalization and cut elimination are not part of the computation engine but instead can be involved in reasoning about computation. See Section 10.4 for a discussion about proof-theoretic methods for reasoning about logic programs.

5.1 Provability via the Sequent calculus

While we assume that the reader has some familiarity with the sequent calculus, we review some basic concepts. Formally, a *sequent* is a pair of multisets of formulas, written as $\Gamma \vdash \Delta$, and we speak of a formula occurrence in Γ as being on the left-hand side and a formula occurrence in Δ as being on the right-hand side of that sequent. Gentzen's proof system for classical logic, called **LK** (Gentzen 1935), allows any number of formulas in Δ, whereas his proof system for intuitionistic logic, called **LJ**, requires Δ to contain at most one formula. Otherwise, proofs in intuitionistic and classical logics use the same set of inference rules.

Inference rules that deal directly with logical connectives are called *introduction rules* and are used to introduce logical connectives into the right or left sides of a sequent. The following three inference rules are used to introduce the conjunction, disjunction, and universal quantifier into the left-hand sides of sequents.

$$\frac{\Gamma, B, C \vdash E}{\Gamma, B \wedge C \vdash E}\ \wedge L \qquad \frac{\Gamma, B \vdash E \qquad \Gamma, C \vdash E}{\Gamma, B \vee C \vdash E}\ \vee L \qquad \frac{\Gamma, [t/x]B \vdash E}{\Gamma, \forall x.B \vdash E}\ \forall L$$

The $\wedge L$ rule says that one way to prove that E follows from $B \wedge C$ and Γ is to prove that E follows from B and C and Γ. The $\vee L$ rule is the sequent calculus version of the *rule of cases*: one way to prove that E follows from $B \vee C$ and Γ is to prove that E follows from B and Γ (the first case) and that E follows from C and Γ (the second case). The $\forall L$ rule says that one way to prove that E follows from $\forall x.B$ and Γ is to prove that E follows from $[t/x]B$ and Γ, where t is some term and $[t/x]B$ is the (capture avoiding) substitution of t for x in B. Figure 1 contains introduction rules for the implication and the universal quantifier.

At least three significant problems with the sequent calculus translate into difficulties using it as a foundation for logic programming. Unlike resolution refutations, the sequent calculus is not equipped with unification, which is recognized as an essential operation in logic programming. For example, in the $\forall L$ rule above, the substitution instance t for $\forall x.B$ must be chosen when this rule is applied, even though the exact nature of that term may not be known in detail until much later in the search for a proof. We set this problem aside until Section 6.

A second serious problem with applying the sequent calculus to logic programming is that its proofs are formless, low-level, and painful to use directly. To illustrate this problem, consider the situation where A is an atomic formula and Γ is a multiset of 998 non-atomic formulas, and where we wish to find a proof of the sequent

$$\Gamma, B_1 \vee B_2, C_1 \wedge C_2 \vdash A.$$

There can be 1000 choices of left introduction rules to attempt in order to prove this sequent.[2] Once one of those choices is made, it is likely that that rule yields at least one premise that again has about 1000 non-atomic formulas on the left. For example, first applying $\vee L$ and then applying $\wedge L$ on each premise can yield

$$\dfrac{\dfrac{\Gamma, B_1, C_1, C_2 \vdash A}{\Gamma, B_1, C_1 \wedge C_2 \vdash A}\ \wedge L \qquad \dfrac{\Gamma, B_2, C_1, C_2 \vdash A}{\Gamma, B_2, C_1 \wedge C_2 \vdash A}\ \wedge L}{\Gamma, B_1 \vee B_2, C_1 \wedge C_2 \vdash A}\ \vee L.$$

This tiny proof fragment is roughly one of about a million choices. Equally unfortunate is what happens if the search for a proof fails to find a proof of the left premise. The proof procedure could then choose to do these two inference rules in the opposite order, namely giving rise to the proof fragment

$$\dfrac{\dfrac{\Gamma, B_1, C_1, C_2 \vdash A \qquad \Gamma, B_2, C_1, C_2 \vdash A}{\Gamma, B_1 \vee B_2, C_1, C_2 \vdash A}\ \vee L}{\Gamma, B_1 \vee B_2, C_1 \wedge C_2 \vdash A}\ \wedge L.$$

However, this *permutation* of inference rules yields the same premises. As a result, proof search will again fail on the left branch. Clearly, switching the order of these rules is not important for completeness. Any high-level structure that sequent calculus proofs might contain needs to be pulled out by extensive inference rule permutation arguments. Such high-level structure in proofs will be more apparent when we upgrade sequent calculus proofs to *focused* proofs in Section 8.

[2] We shall use the left-hand side of a sequent to store a logic program. A logic program with 1000 formulas (clauses) is a small-to-medium-sized program.

Structural rules

$$\frac{\Gamma, B, B \vdash E}{\Gamma, B \vdash E}\ contr \qquad\qquad \frac{\Gamma \vdash E}{\Gamma, B \vdash E}\ weak$$

Identity rules

$$\frac{}{B \vdash B}\ init \qquad\qquad \frac{\Gamma_1 \vdash B \qquad \Gamma_2, B \vdash E}{\Gamma_1, \Gamma_2 \vdash E}\ cut$$

Introduction rules

$$\frac{\Gamma_1 \vdash B_1 \qquad \Gamma_2, B_2 \vdash E}{\Gamma_1, \Gamma_2, B_1 \supset B_2 \vdash E} \supset L \qquad \frac{\Gamma, B_1 \vdash B_2}{\Gamma \vdash B_1 \supset B_2} \supset R$$

$$\frac{\Gamma, [t/x]B \vdash E}{\Gamma, \forall x.B \vdash E}\ \forall L \qquad\qquad \frac{\Gamma \vdash [y/x]B}{\Gamma \vdash \forall x.B}\ \forall R$$

Fig. 1. The subset of Gentzen's **LJ** proof system that applies to only $\supset$ and $\forall$. In the $\forall R$ rule, the variable y is not free in any formula in the conclusion of that rule.

The third serious problem with applying the sequent calculus to logic programming is that its inference rules are too tiny and not the right inference rules in many settings. For example, consider a multiset of formulas Γ that contains the following two Horn clauses.

$$\forall x \forall y [adj\ x\ y \supset path\ x\ y]$$
$$\forall x \forall y \forall z [path\ x\ z \supset path\ z\ y \supset path\ x\ y]$$

The effect of using these formulas in a proof can naturally be viewed as describing inference rules directly. For example, the backward-chaining interpretation of these formulas seems best captured using the following pair of rules:

$$\frac{\Gamma \vdash adj\ x\ y}{\Gamma \vdash path\ x\ y} \qquad\qquad \frac{\Gamma \vdash path\ x\ z \quad \Gamma \vdash path\ z\ y}{\Gamma \vdash path\ x\ y}\ .$$

The forward-chaining interpretation of these formulas seems best captured using the following pair of rules:

$$\frac{\Gamma, adj\ x\ y, path\ x\ y \vdash A}{\Gamma, adj\ x\ y \vdash A} \qquad \frac{\Gamma, path\ x\ z, path\ z\ y, path\ x\ y \vdash A}{\Gamma, path\ x\ z, path\ z\ y \vdash A}\ .$$

Note that none of these rules explicitly contain occurrences of logical connectives. When we deal with *polarization* and *focused* proofs in Section 8, we will show how to construct these inference rules from Horn clauses and how polarization selects between rules following the backward-chaining or forward-chaining discipline.

5.2 Goal-directed proofs

Throughout this survey, we shall see several different sequent calculi presented as a collection of inference rules. To simplify the presentation and comparison of such systems, we shall usually restrict our attention to formulas containing just the logical connectives for universal quantification and implication. For example, Figure 1 contains the subset of Gentzen's **LJ** proof system (Gentzen 1935) that applies to only the logical connectives $\supset$ and $\forall$. Proof systems in the literature (for example, (Gentzen 1935)) usually contain more logical connectives (e.g., disjunction, conjunction, and existential quantifiers). The variable y used in the $\forall R$ rule is called the *eigenvariable* for that rule.

An early application of sequent calculus to logic programming was the development of the technical term *uniform proof* to capture the notion of *goal-directed search* (Miller et al. 1991). In particular, the sequent $\mathcal{P} \vdash G$ describes the obligation to prove the *goal formula* G from the *(logic) program* $\mathcal{P}$.

To formalize the fact that a proof attempt is goal-directed, we will insist that whenever the goal formula is non-atomic (hence, its top-level symbol is a logical connective), this sequent can only be proved using a right introduction rule. Even if the left-hand side $\mathcal{P}$ contains 1000 non-atomic formulas, a goal-directed proof must ignore the possibility of introducing those formulas and only allow the right-hand formula to be selected. Only when the top-level symbol of the goal formula is non-logical (that is, it is a predicate symbol) is the proof attempt permitted to consider the left-hand side. Such sequent calculus proofs were called *uniform proofs* (Miller et al. 1991). In general, a uniform proof is divided into two phases. One phase involves a sequence of right-introduction rules that performs goal reduction. The other phase—the *backward-chaining phase*—selects a formula from the left-hand side $\mathcal{P}$ and performs a sequence of left-introduction rules derived from that one formula.

To illustrate a backward-chaining phase, consider the following proof fragment. Here, $\mathcal{P}$ is a multiset of formulas that includes the two Horn clauses in the previous section that describe the *path* predicate.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{P} \vdash \mathit{path}\ a\ b \qquad \dfrac{\mathcal{P} \vdash \mathit{path}\ b\ c \qquad \dfrac{}{\underline{\mathit{path}\ a\ c} \vdash \mathit{path}\ a\ c}\ \mathit{init}}{\mathcal{P},\ \underline{\mathit{path}\ b\ c \supset \mathit{path}\ a\ c} \vdash \mathit{path}\ a\ c}\ \supset L}{\mathcal{P}, \mathcal{P},\ \underline{\mathit{path}\ a\ b \supset \mathit{path}\ b\ c \supset \mathit{path}\ a\ c} \vdash \mathit{path}\ a\ c}\ \supset L}{\mathcal{P},\ \underline{\mathit{path}\ a\ b \supset \mathit{path}\ b\ c \supset \mathit{path}\ a\ c} \vdash \mathit{path}\ a\ c}\ \mathit{contr}}{\mathcal{P},\ \underline{\forall x \forall y \forall z (\mathit{path}\ x\ y \supset \mathit{path}\ y\ z \supset \mathit{path}\ x\ z)} \vdash \mathit{path}\ a\ c}\ \forall L \times 3}{\mathcal{P} \vdash \mathit{path}\ a\ c}\ \mathit{contr}
$$

Here, a formula on the left is highlighted by underlining it. The backward-chaining phase has four important features. First, it is invoked only if the goal on the right is atomic. Second, only the highlighted formula is the site of a left-introduction rule. Third, if the highlighted formula is atomic, then the sequent in which it occurs must be the conclusion of the *init* rule: i.e., the highlighted formula and the goal must be equal. Fourth and finally, the contraction rule is responsible for selecting a formula on which to focus. Note that during all steps in building this phase, the contraction rule may have to make many choices: once that choice is taken, there is no longer any choice as to which left-introduction rule gets applied. If we erase all sequents in the fragment above containing an underlined formula, then the result is exactly one of the backward-chaining inference rules from the previous section.

It is now possible to put these various notions together and define an *abstract logic programming language* as a triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ such that for all finite subsets $\mathcal{P}$ of $\mathcal{D}$ and all formulas G of $\mathcal{G}$, $\mathcal{P} \vdash G$ is provable if and only if the sequent $\mathcal{P} \vdash G$ has a uniform proof.

Let $\mathcal{D}_1$ and $\mathcal{G}_1$ be collections of Horn clauses and goal formulas as described by lines

(2) and (3) in Section 2. Using basic proof theory arguments, it is easy to show that $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ and $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$ are both abstract logic programming languages (here, $\vdash_C$ and $\vdash_I$ denote provability in classical and intuitionistic logics, respectively). Thus, Horn clauses—using intuitionistic or classical logic—provide an example of an abstract logic programming language. In a sense, Horn clauses form a setting that is so weak that it cannot distinguish between classical and intuitionistic provability.

5.3 Higher-order Horn clauses

While the functional programming world has embraced higher-order programming since its inception, logic programmers have often held such programming style at an arm's length. For example, D. H. D. Warren (1982) argued that higher-order predicate quantification could be translated away and, as a result, an explicit higher-order extension to logic programming was not needed. Similarly, HiLog (Chen et al. 1993) added mild extensions to the syntax of Prolog to accommodate some aspects of higher-order programming, but HiLog was restricted to maintain the first-order aspects of the underlying implementation of Prolog.

Although Church did not use structural proof theory to introduce the higher-order logic he called the *Simple Theory of Types* (1940), several proof-theoretic treatments of the classical and intuitionistic versions of higher-order logic were developed in the decades following its introduction (Takeuti 1953; Takahashi 1967; Girard 1971). When Gentzen's notion of sequent calculus is used to describe the classical and intuitionistic versions of Church's Simple Theory of Types, one gets an elegant proof system for very expressive logics. There was also early work on implementing various aspects of theorem proving in Church's logic, including unification (Huet 1975), resolution (Andrews 1971; Huet 1973), and general theorem proving (Andrews et al. 1984; Paulson 1989).

Starting with that earlier work, Nadathur and the author worked on trying for a genuine, higher-order logic generalization to logic programming. They defined a notion of *higher-order Horn clauses* (*hohc*), proved that they formed an abstract logic programming language, and described the design of an interpreter for what was the basis of an early version of λProlog (Miller and Nadathur 1986a; Nadathur 1987; Nadathur and Miller 1990). In this new logic programming language, it is easy to write higher-order programs, such as the following (using λProlog syntax).

```
type foreach, forsome   (A -> o) -> list A -> o.
type mappred            (A -> B -> o) -> list A -> list B -> o.

foreach P [].
foreach P [X|L] :- P X, foreach P L.

forsome P [X|L] :- P X; forsome P L.

mappred P [] [].
mappred P [X|L] [Y|K] :- P X Y, mappred P L K.
```

In the first two lines above, the types of three higher-order predicates are declared. These type expressions follow a convention begun by Church (1940) in which `o` is used to denote the type of formulas. Thus, a symbol of type `nat -> o` denotes a predicate of one argument of type `nat`. Capital letters in type expressions denote type variables: thus, these type declarations are polymorphically typed in a sense similar to, say, ML (Milner et al. 1990; Nadathur and Pfenning 1992).

Actually, *hohc* contained more than is necessary to capture higher-order relational programming. For example, it includes quantification over non-primitive and non-relational types as well as the simply-typed λ-calculus with equality and unification modulo α, β, and η conversions. As a result, some sophisticated computations on syntactic expressions containing bindings are possible in *hohc*, including, for example, program transformations (Huet and Lang 1978) and natural language semantics (Miller and Nadathur 1986b). For those not interested in dealing with bindings in term structures, Wadge (1991) and Bezem (2001) have developed restrictions to *hohc* that seem to capture what is needed for higher-order relational programming (including those displayed above).

5.4 Hypothetical goals and modular structures

Consider the following definition for a larger class of definite (program) clauses and goal formulas that extends the corresponding definition for Horn clauses and their goals (given by lines (2) and (3) in Section 2) by adding implication and universal quantification to goals.

$$G := A \mid \top \mid G \wedge G \mid \bot \mid G \vee G \mid \exists x.G \mid D \supset G \mid \forall x.G \tag{4}$$

$$D := A \mid G \supset D \mid \top \mid D \wedge D \mid \forall x.D \tag{5}$$

These definitions of G and D-formulas are mutually recursive. This definition is large enough to contain the extended logic programming systems that were mentioned at the beginning of Section 5. Compare the definition for D above with the following definition of *Harrop* formulas (Harrop 1960):

$$H := A \mid B \supset H \mid \top \mid H \wedge H \mid \forall x.H,$$

where the syntactic variable B ranges over arbitrary first-order formulas. The D formulas in (5) are such that any negatively occurring subformula of a D formula is such that negatively occurring subformulas of them are again Harrop formulas. Motivated by this observation, such D formulas are called first-order *hereditary Harrop formulas* (*fohh*) (Miller et al. 1991).

Let $\mathcal{D}_2$ and $\mathcal{G}_2$ be collections of formulas described by lines (4) and (5). Using inductive arguments with sequent calculus proofs, it can be shown that the triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ is an abstract logic programming language. Unlike the case with Horn clauses, the triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_C \rangle$ is *not* an abstract logic programming language. To see this, consider the goal formula $(p \supset q) \vee p$, where p and q are two propositional constants. Note that this formula is classically provable since we have the following classical equivalences (using only the associativity and commutativity of $\vee$ and the equivalence $B \supset C \equiv \neg B \vee C$):

$$(p \supset q) \vee p \equiv (\neg p \vee q) \vee p \equiv q \vee (\neg p \vee p) \equiv q \vee (p \supset p).$$

Since the last of these formulas is true, they are all true and classically provable. However, the sequent $\vdash (p \supset q) \vee p$ does not have a uniform proof since uniformity requires that this sequent is the conclusion of a right introduction of $\lor$ in which case either $\vdash p \supset q$ or $\vdash p$ are provable: but, of course, neither of these sequents are provable.

Goal formulas in *fohh* allow *hypothetical* goals: if we attempt to find a uniform proof of the sequent $\mathcal{P} \vdash D \supset G$ then that attempt leads to attempting to prove $\mathcal{P}, D \vdash G$. Note that the left-hand side of sequents can get larger as one searches for a uniform proof. For example, an attempt to find a uniform proof of the sequent

$$\mathcal{P} \vdash (D_1 \supset (G_1 \land (D_2 \supset G_2))) \land G_3 \land (D_3 \supset G_4)$$

must lead to attempts to prove the four sequents

$$\mathcal{P}, D_1 \vdash G_1 \qquad \mathcal{P}, D_1, D_2 \vdash G_2 \qquad \mathcal{P} \vdash G_3 \qquad \mathcal{P}, D_3 \vdash G_4.$$

Hence, the logic program (the left-hand side) for these subgoals can all be different. Such an observation has been used to design a logic-based approach to modular programming within logic programming: in particular, the goal $D \supset G$ can be operationally interpreted to mean that an individual goal can request that the program clauses in D are loaded before attempting to prove G (Miller 1989).

The use of hypothetical goals to load code available for a certain scope is not supported by classical logic. In particular, the intuitionistic logic interpretation of the goal $(D \supset G) \lor H$ means that D is available during the search for the proof of G but not H. In classical logic, this scoping breaks down since, classically, this formula is equivalent to $G \lor (D \supset H)$ and to $(D \supset (G \lor H))$ (using the classical equivalences mentioned above). An overview of modularity proposals for logic programming based on proof theory, Kripke semantics, modal operators, and algebraic operators can be found in the survey by Bugliesi et al. (1994).

Several researchers have used uniform proofs to motivate new proof procedures and new logic programming designs. Nadathur (1993) presented a proof procedure for hereditary Harrop formulas in which the interplay of unification and eigenvariables is explicitly treated. Harland (1997) and Nadathur (1998; 2000) have also provided new proof systems for classical logic based on using uniform proofs. Recently, such proofs have also been used to describe a coinductive proof procedure for Horn clauses (Basold et al. 2019). As we shall see in Section 7, uniform proofs have also been used to design various *linear logic programming languages*.

6 Universal goals, binder mobility, and abstract data types

Logic programming based on hereditary Harrop formulas allows goals to be universally quantified. We now describe the operational significance of that quantifier in logic programming. Once again, Gentzen's sequent calculus provides an elegant treatment for such quantification using the notion of *eigenvariable*.

6.1 Eigenvariables as explicit bindings

The $\forall R$ inference rule in Figure 1 replaces the universally quantified binding for x in its conclusion with a free variable y in its premise. The 47^{th} epigram of Alan Perlis (1982) is worth repeating here: "As Will Rogers would have said, 'There is no such thing as a free variable.' " The wisdom of this epigram is that free variables are, in fact, bound (or declared) somewhere, and that place should be made explicit. To this end, consider replacing the sequent $\Gamma \vdash E$ with $\Sigma\colon\ \Gamma \vdash E$, where Σ is a list of distinct variables that are considered bound over both Γ and E. Thus, the eigenvariables for a sequent are explicitly bound over the sequent. The inference rules for the universal quantifier in Figure 1 are then changed as follows.

$$\frac{\Sigma \vdash t : \tau \qquad \Sigma\colon\ \Gamma, [t/x]B \vdash E}{\Sigma\colon\ \Gamma, \forall x_\tau.B \vdash E}\ \forall L \qquad\qquad \frac{\Sigma, y : \tau\colon\ \Gamma \vdash [y/x]B}{\Sigma\colon\ \Gamma \vdash \forall_\tau x.B}\ \forall R$$

In the $\forall R$ rule, the eigenvariable is now explicitly bound within the sequent. The $\forall L$ rule is also updated with the premise $\Sigma \vdash t : \tau$ that ensures that the term t is built from only the eigenvariable in the context Σ. If we limit ourselves to a simple single-sorted first-order logic, then the type variable τ is set to just that sort. If we are working with a multi-sorted logic, τ could range over the various sorts. Finally, if we are dealing with the *elementary type theory* (Andrews 1974) subset of Church's Simple Theory of Types (Church 1940), then τ could range over all simple types, including higher-order types.

The other inference rules are also given the Σ binder prefix, but there is no interaction between those rules and this binder.

6.2 The λ-tree approach to syntax encoding

Those logic programming languages based directly on the sequent calculus have an elegant and direct technique for specifying computations in which terms may include bindings. This technique uses the three levels of bindings available in sequents: term-level bindings (in, say, λ-terms), formula-level bindings ($\forall$ and $\exists$ quantifiers), and proof-level bindings (eigenvariables). Furthermore, logic specifications are capable of having such bindings *move* between these different levels.

To illustrate this approach to computing with binders, consider an encoding of untyped λ-terms into simply-typed terms. In particular, let type tm be the type of encoded untyped λ-terms and let app and abs be constants of types $tm \to tm \to tm$ and $(tm \to tm) \to tm$, respectively. The following three clauses define a function $\lceil\cdot\rceil$ that translates untyped λ-terms into terms of type tm.

$$\lceil(MN)\rceil = (app\ \lceil M\rceil\ \lceil N\rceil) \qquad \lceil\lambda x.B\rceil = (abs\ (\lambda x.\lceil B\rceil)). \qquad \lceil x\rceil = x \text{ for variable } x$$

Note that bound variables in the untyped λ-terms correspond to bound variables in terms of type tm.

Consider now the problem of deciding whether or not an untyped λ-term can be given a simple type. To represent simple types, we introduce the type ty and the constant $arrow : ty \to ty \to ty$ that represents the $\to$ in simple types. The following logic program specifies the $typeof\ M\ T$ predicate that should hold if and only if the untyped λ-term M has simple type T (the type of $typeof$ is $tm \to ty \to o$). Note that this specification uses a program clause that contains both a universal quantifier and an implication in its body.

$$\begin{array}{rl} \forall B\forall T\forall T'[typeof\ (abs\ B)\ (T \to T') & :\!-\ \forall x(typeof\ x\ T \supset typeof\ (B\ x)\ T')] \wedge \\ \forall M\forall N\forall T\forall T'[typeof\ (app\ M\ N)\ T & :\!-\ typeof\ M\ (T' \to T) \wedge typeof\ N\ T'] \end{array}$$

Now consider the following combination of inference rules that are built when type checking a λ-abstraction.

$$\cfrac{\cfrac{\Sigma, \underline{x} : \Delta, \mathit{typeof}\ x\ T \vdash \mathit{typeof}\ \lceil B \rceil\ T'}{\Sigma : \Delta \vdash \forall \underline{x}\ (\mathit{typeof}\ x\ T \supset \mathit{typeof}\ \lceil B \rceil\ T')}\ \forall R,\ \supset R}{\Sigma : \Delta \vdash \mathit{typeof}\ \lceil \lambda \underline{x}.B \rceil\ (T \to T')}\ \text{backward chaining}$$

The binding for x moves from the term-level, to the formula-level (as a quantifier), to the proof-level (as an eigenvariable): these occurrences are underlined to highlight them. It is in this sense that the sequent calculus supports the *mobility of binders* (Miller 2019): that is, bound variables do not become free, instead, their scopes move.

Higher-order Horn clauses do not support the movement of bindings since no universally quantified goals nor eigenvariables are part of proof search involving them. In light of this, λProlog, which originally started as an implementation of *hohc* was extended to include both hypothetical and universally quantified goals in order to support binder mobility.

The term *higher-order abstract syntax* (Pfenning and Elliott 1988) is often used to describe systems in which the bindings in data structures are implemented using bindings in a programming language. Unfortunately, this term is ambiguous since such identification in the functional programming setting has almost no relationship with the approach described above. For example, if one uses bindings in an ML-style language, then functions are used to encode the syntax of terms with bindings. Such an encoding has many shortcomings (Despeyroux et al. 1995; Hofmann 1999), and it does not generally support checking the equality of syntax. Thus, the approach described above—providing binder mobility and equality via (at least) α-conversion—has been named the *λ-tree syntax* approach to differentiate it from the functional programming approach (Miller 2019).

The αProlog system (Cheney and Urban 2004) is a logic programming language with a different approach to encoding and computing with syntax containing bindings. Instead of using eigenvariables and binder mobility, αProlog is based on the logic of Pitts (2003) which uses the Fraenkel-Mostowski permutation model of set theory to provide a mechanism for generating and permuting the names used to encode binders.

6.3 Unification under a mixed prefix

Traditional unification can be seen as a technique for proving formulas of the form

$$\exists x_1 \ldots \exists x_n [t_1 = s_1 \wedge \cdots \wedge t_m = s_m] \quad (n \geq 0),$$

where the quantifier prefix is purely existential. In principle, such unification problems are sufficient to consider when building an interpreter for first-order Horn clauses. There are at least two ways in which richer designs of logic programming languages force one to consider performing unification under a *mixed prefix*, i.e., where both existential and universal quantifiers have the conjunction of equations in their scope.

One such extended design involves replacing first-order terms with simply-typed λ-terms, as is the case of *hohc*. The equality theory of such typed terms is generally assumed to contain not only the α, β, and η rules but also the ξ-rule, which states that the two expressions $\lambda x.t = \lambda x.s$ and $\forall x.t = s$ are logically equivalent. Since the forward direction of this equivalence is easily proved, the force of the ξ-rule is the converse. Using this equivalence, we can show that the unification problem $\exists y[\lambda z.y = \lambda z.z]$ involving

λ-terms is equivalent to the mixed prefixed unification problem $\exists y \forall z[y = z]$, which is an entirely first-order formula (assuming that the variables y and z have the same primitive type). Furthermore, these formulas are, in fact, not provable (unless one has additional axioms stating that the domain of quantification is a singleton set). Thus, more generally, the unification of simply-typed λ-terms can be seen as having an $\exists\forall$ prefix.

A second extended design arises in the simple use of *fohh*. For example, let Σ be some list of eigenvariables and let $\mathcal{P}$ be the following set of *fohh* formulas.

$$\{\forall x.\ r\ x\ x, \quad \forall y.[(\forall z.\ r\ y\ z) \supset q\ y]\}.$$

It should be clear that there is no proof of $\exists x.q(x)$ from Σ and $\mathcal{P}$. A proof attempt of this goal can be sketched using the following arrangement of sequents and pseudo-inference rules.

$$\dfrac{\Sigma \vdash X : i \qquad \dfrac{\dfrac{\dfrac{X = z}{z : i\colon\ \mathcal{P} \vdash r\ X\ z}\ \text{backward chaining}}{\Sigma\colon\ \mathcal{P} \vdash \forall z.\ r\ X\ z}\ \forall R}{\Sigma\colon\ \mathcal{P} \vdash q\ X}\ \text{backward chaining}}{\Sigma\colon\ \mathcal{P} \vdash \exists x.\ q\ x}\ \exists R$$

Here, i the a primitive type for quantification. We use X as a kind of logic variable: instead of instantiating the existential quantifier with a term (as is the requirement in Gentzen's inference rules), we enter X as a kind of hole that we plan to fill later, but we must remember that whatever fills that hole must be a term over the variables in Σ. Finally, moving upwards through the series of sequents, we can conclude that we have a proof if X is instantiated with z, which is an eigenvariable that is not a member of Σ. Thus, these two conditions are contradictory, and, as a result, there is no proof. Nadathur (1993) describes a unification procedure that works in the presence of the quantifier alternations that occur during proof search with hereditary Harrop formulas.

The general problem of unification of simply-typed λ-terms under a mixed prefix can be found in the work by the author (1992), which is itself an extension of the earlier work by Huet on unification for typed λ-calculus (1975). While Skolemization is often used in automated theorem provers to remove issues surrounding quantifier alternations, an alternative exists that works with binder mobility. It is possible to rotate a universal quantifier to the right over an existential quantifier: that is, $\forall y \exists x.B$ and $\exists h \forall y.[(hy)/x]B$ represent the same unification problem. In the first, the term t instantiating x can contain the eigenvariable associated with y, while in the second, h is instantiated with $\lambda y.t$, which, of course, does not contain y free. The type of the existentially quantified variable is raised in this process: in particular, if y has type τ and x has type τ' then h has type $\tau \to \tau'$. As a result, this operation is called *raising*, and it can be used to simplify all quantifier prenexes to the $\exists\forall$ kind (Miller 1992). (Raising is closely related to the $\forall$-lifting technique use to deal with eigenvariables in Isabelle (Paulson 1989).) To illustrate raising, consider the following unification problems where f is a function constant of two arguments.

$$\begin{array}{rl}
\forall x \exists y \forall z\ [(f\ z\ y) = (f\ z\ x)] & \\
\forall x \exists y\ [\lambda z(f\ z\ y) = \lambda z(f\ z\ x)] & (\xi) \\
\exists h \forall x\ [\lambda z(f\ z\ (h\ x)) = \lambda z(f\ z\ x)] & (\mathit{raising}) \\
\exists h\ [\lambda x \lambda z(f\ z\ (h\ x)) = \lambda x \lambda z(f\ z\ x)] & (\xi)
\end{array}$$

A *solution* is a substitution for the existentially quantified variables that makes the equated terms the same (modulo $\alpha\beta\eta$-convertibility). All of the above unification problems have their solutions in a one-to-one correspondence. In particular, the unique solution for the first problem is the substitution that maps y to x while the unique solution for the last problem is the substitution that maps h to $\lambda x.x$.

In the 1980s and earlier, there were many concerns that higher-order unification was too complex to allow within the logic programming setting. While some concern is justifiable, avoiding all forms of higher-order unification meant that the full story of unification in quantificational first-order was not told. At the same time, early implementations of higher-order unification in theorem provers indicated that it was not generally a bottleneck (Andrews et al. 1984; Paulson 1989). Part of the reason for the mild behavior of higher-order unification seems to be that many uses of higher-order unification tend to belong to the *higher-order pattern unification* fragment, which, like first-order unification, is a decidable and unary subset of higher-order unification (Miller 1991a; Nipkow 1993). In fact, systems such as Twelf (Pfenning and Schürmann 1999), Teyjus (Qi et al. 2015), Elpi (Dunchev et al. 2015), and Minlog (Schwichtenberg 2006) can encounter arbitrary higher-order unification problems but they only solve those unification problems that fall within this fragment: in most practical situations, this approach to higher-order unification is sufficient.

6.4 Abstract data types in logic programming

Similar to implications, universal quantifiers in goals can provide scope for term constructors within goal formulas: exploiting such a scoping mechanism for constructors provides a logic-based notion of abstract data type.

Judging from the name "eigenvariable", one expects that they *vary.* However, eigenvariables do not vary within a cut-free proof: they act more like constants given a particular scope. It is only during cut-elimination that eigenvariables can vary since they are then substituted by other terms. Thus, in the setting of proof search, it makes more sense to view eigenvariables as scoped constants.

Assume that the variable y is free in the formula D but not in G. The interpreter attempting to prove $\forall y(D \supset G)$ will then introduce a new eigenvariable for y, say k, and restrict all the current free variables so that they cannot be instantiated with terms containing k. The program code $[k/y]D$ can use the constant k to build data structures. Of course, if we are building an interpreter that uses unification, care must be taken to deal with the fact that some eigenvariables (constants) might be introduced before or after logic variables are introduced. We addressed this issue in Section 6.3. In the discussions above, the scope of y is, in a sense, only over D while we needed to use the universal quantifier $\forall y$ over the compound formula $D \supset G$, even though y is not free in G. To provide for a more natural scoping mechanism, note that $(\exists x\ D) \supset G$ and $\forall x(D \supset G)$ are equivalent (in intuitionistic logic) provided x is not free in G. Thus, we can use an existential quantifier over program clauses to limit the scope of constants used in those programs. Although $(\exists x\ D) \supset G$ is not a valid hereditary Harrop formula, it is equivalent to $\forall x(D \supset G)$, which is a valid such formula. To allow for the most interesting examples, we shall allow higher-order quantification for such locally scoped variables.

Consider the two existentially quantified conjunctions of Horn clauses displayed in

$$\begin{array}{ll}
\exists emp\ \exists stk.\ (& \exists qu.\ (\\
\quad empty\ emp \wedge & \quad [\forall l.\ empty\ (qu\ l\ l)] \wedge \\
\quad [\forall s \forall x.\ enter\ x\ s\ (stk\ x\ s)] \wedge & \quad [\forall x \forall l \forall k.\ enter\ x\ (qu\ l\ [x|k])\ (qu\ l\ k)] \wedge \\
\quad [\forall s \forall x.\ remove\ x\ (stk\ x\ s)\ s]) & \quad [\forall x \forall l \forall k.\ remove\ x\ (qu\ [x|l]\ k)\ (qu\ l\ k)])
\end{array}$$

Fig. 2. Two implementations of the predicates *empty*/*enter*/*remove*.

Figure 2. In both of those formulas, the only constants that appear free are the predicates *empty*, *enter*, and *remove*. The formula on the left is an implementation of a stack: here, *emp* denotes the empty stack, and *stk* denotes the non-empty stack constructor. In this case, the enter/remove predicates implement the last-in-first-out protocol. The formula on the right is an implementation of a queue: here *qu* forms a difference list in the usual style familiar to Prolog programmers (Clocksin and Mellish 1994). In this case, the enter/remove predicates implement the first-in-first-out protocol. Note that by hiding the internal implementation of the three predicates, it is possible to change one of these implementations with the other without the calling code becoming broken. Of course, the calling code might well have a different behavior when we swap implementations.

Hiding predicates is also possible using such higher-order quantification. For example, the usual way to specify the relationship between a list and its reverse is often defined using an auxiliary predicate, which can be hidden using a universal quantifier in a goal. Consider the following hereditary Harrop formula.

$$\begin{array}{l}
\forall L \forall K.\ reverse\ L\ K\ \texttt{:-}\ \forall rev.\ (\\
\qquad\qquad [\forall L.\ rev\ []\ L\ L] \supset \\
\qquad [\forall X \forall L \forall K \forall M.\ rev\ [X|L]\ K\ M\ \texttt{:-}\ rev\ L\ K\ [X|M]] \supset \\
\qquad\qquad\qquad rev\ L\ K\ [])
\end{array}$$

To prove an instance of the reverse relationship, this code instructs the proof search mechanism to create a new eigenvariable that plays the role of an auxiliary predicate *rev* and then loads two Horn clauses that define that auxiliary predicate before making a call to that auxiliary predicate. As a result, it is impossible to access this auxiliary predicate and its code from any other logic programming clauses that may be in the same context. More examples of this approach to abstract data types in logic programming can be found in (Miller 2003) and (Miller and Nadathur 2012).

7 Linear logic programming

All the previous developments in applying proof theory to logic programming took place within classical and intuitionistic logic. When Girard introduced linear logic in (1987), many researchers were eager to see if the story behind logic programming could be extended further using this new logic (see the encyclopedia article (Di Cosmo and Miller 2019) for an overview of linear logic). This new logic also seemed to be an extension to both classical and intuitionistic logic: as a result, there was the promise that linear logic programming could subsume and extend the various forms of logic programming we have already described. Also, the proof theory foundations of the logic programming paradigm had not provided any hints at how to account for either side-effects or concurrency: but there were hints that linear logic should provide for exactly these missing features. Since Girard gave a simple and clear presentation of linear logic using the sequent calculus, many researchers started working on new logic programming designs almost immediately.

Below is a list of several logic programming languages that incorporate elements of linear logic into their design. For more about linear logic programming, the reader is referred to the author's overview paper (2004).

- The LO (linear objects) language designed by Andreoli and Pareschi (1991) was the first of those languages. LO was a kind of Horn clause logic where atomic formulas were generalized to be more like a multiset of atomic formulas. The design provided a natural notion of an object-as-process that has a built-in notion of inheritance.
- Lolli is a simple extension to hereditary Harrop formulas (Hodas and Miller 1991; Hodas and Miller 1994). Essentially, the linear implication $\multimap$ is allowed to appear in the same way as the intuitionistic implication can appear: at the top-level of both definite clauses and goals. Lolli had the property that if a program never uses $\multimap$ as a goal formula, then proofs and proof search are essentially the same as when using intuitionistic logic. A new feature that Lolli provides over λProlog is a mechanism for describing state and state change, including database updates and retraction.
- The Lygon system of Harland and Pym (1996) was designed following a proof-theoretic analysis of goal-directed proof in linear logic (Pym and Harland 1994). The application areas of Lygon and Lolli overlap significantly.
- The language ACL by Kobayashi and Yonezawa (Kobayashi and Yonezawa 1993; Kobayashi and Yonezawa 1994) captures simple notions of asynchronous communication by identifying the send and read primitives with two complementary linear logic connectives.
- Lincoln and Saraswat developed a linear logic version of concurrent constraint programming (Lincoln and Saraswat 1993; Saraswat 1993), and Fages, Ruet, and Soliman have analyzed similar extensions to the concurrent constraint paradigm (Ruet and Fages 1997; Fages et al. 1998).
- The Forum language (Miller 1996; Bruscoli and Guglielmi 2006) is essentially a presentation of linear logic that allows for all of linear logic to be considered as an abstract logic programming language. The proof-theoretic analysis of Forum required lifting the notion of goal-directed proofs to deal with multiple-conclusion sequents. Forum can be seen as the result of merging LO and Lolli.

An early observation about linear logic is that it supports multiset rewriting in a rather direct fashion. Thus, linear logic programming can encode both Petri nets (Gunter and Gehlot 1989; Kanovich 1995) and the process calculi, such as the π-calculus (Miller 1993).

To illustrate how sequent calculus can be used to encode a small fragment of linear logic (the fragment that deals with $\supset$, $\multimap$, and $\forall$), we present the **LL** proof system in Figure 3. We continue to use $\supset$ to denote (intuitionistic) implication and introduce Girard's linear implication $\multimap$. Part of the informal meaning of linear implication is that a proof of $B \multimap C$ is a proof of C in which the assumption B is used exactly once. The corresponding informal meaning of the intuitionistic implication is that a proof of $B \supset C$ is a proof of C in which the assumption B is used any number of times, including zero. To permit these two different accounting methods for assumptions, the left-hand

Structural rules

$$\frac{\Gamma, B, B \,;\Delta \vdash E}{\Gamma, B \,;\Delta \vdash E}\; contr \qquad \frac{\Gamma \,;\Delta \vdash E}{\Gamma, B \,;\Delta \vdash E}\; weak \qquad \frac{\Gamma \,; B, \Delta \vdash E}{\Gamma, B \,;\Delta \vdash E}\; \text{dereliction}$$

Identity rules

$$\frac{}{\cdot \,; B \vdash B}\; init \qquad \frac{\Gamma_1 \,;\Delta_1 \vdash B \qquad \Gamma_2 \,;\Delta_2, B \vdash E}{\Gamma_1, \Gamma_2 \,;\Delta_1, \Delta_2 \vdash E}\; cut$$

Introduction rules

$$\frac{\Gamma_1 \,; \cdot \vdash B_1 \qquad \Gamma_2 \,;\Delta, B_2 \vdash E}{\Gamma_1, \Gamma_2 \,;\Delta, B_1 \supset B_2 \vdash E} \supset L \qquad \frac{\Gamma, B_1 \,;\Delta \vdash B_2}{\Gamma \,;\Delta \vdash B_1 \supset B_2} \supset R$$

$$\frac{\Gamma_1 \,;\Delta_1 \vdash B_1 \qquad \Gamma_2 \,;\Delta_2, B_2 \vdash E}{\Gamma_1, \Gamma_2 \,;\Delta_1, \Delta_2, B_1 \multimap B_2 \vdash E} \multimap L \qquad \frac{\Gamma \,;\Delta, B_1 \vdash B_2}{\Gamma \,;\Delta \vdash B_1 \multimap B_2} \multimap R$$

$$\frac{\Gamma \,;\Delta, [t/x]B \vdash E}{\Gamma \,;\Delta, \forall x.B \vdash E}\; \forall L \qquad \frac{\Gamma \,;\Delta \vdash [y/x]B}{\Gamma \,;\Delta \vdash \forall x.B}\; \forall R$$

Fig. 3. The **LL** proof system for $\supset$, $\multimap$, and $\forall$. In the $\forall R$ rule, the variable y is not free in the conclusion of that rule.

side of sequents is divided into two zones. In the sequent $\Gamma \,;\Delta \vdash E$, the context Γ holds the assumptions under the unbounded-use accounting, and the context Δ contains the assumptions under the single-use accounting: we refer to Γ as the *unbounded zone* and Δ as the *bounded zone*. Hodas and the author (1994) proved that **LL** (over the same connectives) is sound and (relatively) complete for linear logic.

Girard's original presentation of linear logic (1987) did not rely on using the two implications $\multimap$ and $\supset$. Instead, the implication $B \supset C$ was defined as $!B \multimap C$, where ! is one of linear logic's *exponentials*. A formula marked by a ! can be contracted and weakened when it appears on the left side of a sequent arrow. Dually, a formula marked by the other exponential ? can be contracted and weakened when it appears on the right. With these exponentials, linear logic can encode both classical and intuitionistic logics. We have chosen not to use the exponentials of linear logic here, but if we did introduce it, then the sequent $B_1, \ldots, B_n \,;\Delta \vdash E$ could be rewritten as $!B_1, \ldots, !B_n, \Delta \vdash E$.

When comparing the subset of the **LJ** proof system in Figure 1 with the **LL** proof system in Figure 3, we see that the contraction and weakening rules are available only in the unbounded zone, that the $\supset R$ rule adds its hypothesis to the unbounded zone, and that the $\multimap R$ rule adds its hypothesis to the bounded zone. Finally, also note that the two left-introduction rules for implication treat their unbounded zones *multiplicatively*, meaning that every side-formula occurrence in the bounded context of the conclusion occurs in a bounded zone of exactly one premise. Furthermore, in the $\supset L$ rule, the bounded zone of the left premise must be empty. Also, note that the only formula occurrences that are introduced on the left occur in the bounded zone. The dereliction rule is responsible for moving a formula in the unbounded zone to the bounded zone.

Figure 4 contains the simplification $\mathbf{LL}'$ of **LL** in which we remove the cut-rule (since we are generally interested here in cut-free proofs) and in which we fold the weakening and contraction rules into other rules so these rules are never explicitly invoked. They are still present in this simplified proof system, however. In particular, the *init* rule allows

$$\frac{}{\Gamma\,;B\vdash B}\;init \qquad \frac{\Gamma,B\,;B,\Delta\vdash E}{\Gamma,B\,;\Delta\vdash E}\;absorb$$

$$\frac{\Gamma\,;\cdot\vdash B_1 \qquad \Gamma\,;\Delta,B_2\vdash E}{\Gamma\,;\Delta,B_1\supset B_2\vdash E}\supset L \qquad \frac{\Gamma,B_1\,;\Delta\vdash B_2}{\Gamma\,;\Delta\vdash B_1\supset B_2}\supset R$$

$$\frac{\Gamma\,;\Delta_1\vdash B_1 \qquad \Gamma\,;\Delta_2,B_2\vdash E}{\Gamma\,;\Delta_1,\Delta_2,B_1\multimap B_2\vdash E}\multimap L \qquad \frac{\Gamma\,;\Delta,B_1\vdash B_2}{\Gamma\,;\Delta\vdash B_1\multimap B_2}\multimap R$$

$$\frac{\Gamma\,;\Delta,[t/x]B\vdash E}{\Gamma\,;\Delta,\forall x.B\vdash E}\;\forall L \qquad \frac{\Gamma\,;\Delta\vdash[y/x]B}{\Gamma\,;\Delta\vdash\forall x.B}\;\forall R$$

Fig. 4. The $\mathbf{LL}'$ proof system results from building into the $\mathbf{LL}$ proof system the structural rules of weak and contr.

the unbounded zone to be non-empty (since weakenings can be used to empty that zone) and the two implication-left rules keep the the unbounded zone the same in the premises and the conclusion. Also, the *absorb* rule links contraction with the dereliction rule.

To illustrate how linear logic provides for new logic programs with new dynamics, consider the following two linear logic formulas.

$$\forall G.(sw\ on \multimap G)\multimap(sw\ off \multimap toggle\ G)$$
$$\forall G.(sw\ off \multimap G)\multimap(sw\ on \multimap toggle\ G)$$

Linear logic contains the conjunction $\otimes$ (pronounced “tensor”) for which the equivalence $(A \multimap B \multimap C)\equiv((A\otimes B)\multimap C)$ holds. Following the Lolli language conventions (Hodas 1994; Hodas and Miller 1994), we write the $\otimes$ as a comma and the converse of $\multimap$ as `:-`. As a result, these two formulas can be written in the following Prolog-like style.

```
toggle(G) :- sw off, (sw on  -o G).
toggle(G) :- sw on,  (sw off -o G).
```

Using the proof system $\mathbf{LL}'$ (Figure 4), we can build the following partial proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\,;sw\ off,\Delta\vdash g}{\Gamma\,;\Delta\vdash sw\ off\multimap g}\multimap R \quad \dfrac{}{\Gamma\,;sw\ on\vdash sw\ on}\,init \quad \dfrac{}{\Gamma\,;toggle\ g\vdash toggle\ g}\,init}{\Gamma\,;(sw\ off\multimap g)\multimap(sw\ on\multimap toggle\ g),sw\ on,\Delta\vdash toggle\ g}\multimap L\times 2}{\Gamma\,;\forall G.(sw\ off\multimap G)\multimap(sw\ on\multimap toggle\ g),sw\ on,\Delta\vdash toggle\ g}\,\forall L}{\Gamma\,;sw\ on,\Delta\vdash toggle\ g}\,absorb$$

This derivation (and the analogous one using the other formula for `toggle`) essentially interprets these two clauses for `toggle` as the following two admissible rules.

$$\frac{\Gamma\,;sw\ off,\Delta\vdash g}{\Gamma\,;sw\ on,\Delta\vdash toggle\ g} \quad \text{and} \quad \frac{\Gamma\,;sw\ on,\Delta\vdash g}{\Gamma\,;sw\ off,\Delta\vdash toggle\ g}$$

Thus, the process of reducing the goal *toggle g* to *g* will flip the switch's value stored as the argument of (the presumably unique) *sw*-atom and will affect no other formula in the bounded or unbounded zones. We can attempt something similar using intuitionistic logic and hereditary Harrop: for example, consider the following specification for toggle.

$$\forall G.(sw\ on\supset G)\supset(sw\ off\supset toggle\ G)$$
$$\forall G.(sw\ off\supset G)\supset(sw\ on\supset toggle\ G)$$

A proof fragment in intuitionistic logic starting with one of these formulas will look as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \mathit{sw\ on}, \mathit{sw\ off} \vdash g}{\Gamma, \mathit{sw\ on} \vdash \mathit{sw\ off} \supset g} \supset R \quad \dfrac{}{\Gamma, \mathit{sw\ on} \vdash \mathit{sw\ on}}\ \mathit{init} \quad \dfrac{}{\mathit{toggle}\ g \vdash \mathit{toggle}\ g}\ \mathit{init}}{\Gamma, \Gamma, (\mathit{sw\ off} \supset g) \supset (\mathit{sw\ on} \supset \mathit{toggle}\ g), \mathit{sw\ on}, \mathit{sw\ on} \vdash \mathit{toggle}\ g} \supset L \times 2}{\Gamma, (\mathit{sw\ off} \supset g) \supset (\mathit{sw\ on} \supset \mathit{toggle}\ g), \mathit{sw\ on} \vdash \mathit{toggle}\ g}\ \text{several}\ \mathit{contr}}{\dfrac{\Gamma, \forall G.(\mathit{sw\ off} \supset G) \supset (\mathit{sw\ on} \supset \mathit{toggle}\ g), \mathit{sw\ on} \vdash \mathit{toggle}\ g}{\Gamma, \mathit{sw\ on} \vdash \mathit{toggle}\ g}\ \mathit{absorb}}\ \forall L$$

In this setting, we get the admissible rules

$$\dfrac{\Gamma, \mathit{sw\ on}, \mathit{sw\ off} \vdash g}{\Gamma, \mathit{sw\ on} \vdash \mathit{toggle}\ g} \quad \text{and} \quad \dfrac{\Gamma, \mathit{sw\ on}, \mathit{sw\ off} \vdash g}{\Gamma, \mathit{sw\ off} \vdash \mathit{toggle}\ g},$$

which is not what we expect from a proper switch.

The mechanism behind this simple example can easily be expanded to perform multiset rewriting. Let H be the *multiset rewriting system* $\{\langle L_i, R_i\rangle \mid i \in I\}$ where for each $i \in I$ (a finite index set), L_i and R_i are finite multisets. Define the relation $M \Longrightarrow_H N$ on finite multisets to hold if there is some $i \in I$ and some multiset C such that M is $C \uplus L_i$ and N is $C \uplus R_i$. Let $\Longrightarrow_H^*$ be the reflexive and transitive closure of $\Longrightarrow_H$.

The H rewrite system can be encoded as a multiset of linear logic formulas as follows: If H contains the pair $\langle\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\}\rangle$ then this pair is encoded as the clause

```
loop :- item a1,   ...,   item an,
        (item b1 -o ... -o item bm -o loop).
```

If either n or m is zero, the appropriate portion of the formula is deleted. Here `item` is a predicate of one argument that is used to inject multiset items into atomic formulas. Operationally, this clause (destructively) reads the a_i's out of the bounded context, loads the b_i's into that context, and then attempts another rewrite. Let Γ_H be the set resulting from encoding each pair in H. For example, if $H = \{\langle\{a, b\}, \{b, c\}\rangle, \langle\{a, a\}, \{a\}\rangle\}$ then Γ_H is the set of clauses

```
loop  :-   item a, item b, (item b -o item c -o loop).
loop  :-   item a, item a, (item a -o loop).
```

The following holds about this encoding of multiset rewriting: the relation

$$\{a_1, \ldots, a_n\} \Longrightarrow_H^* \{b_1, \ldots, b_m\}$$

holds if and only if sequent $\Gamma ; \Delta, \mathit{item}(a_1), \ldots, \mathit{item}(a_n) \vdash \mathit{loop}$ can be derived from the sequent $\Gamma ; \Delta, \mathit{item}(b_1), \ldots, \mathit{item}(b_m) \vdash \mathit{loop}$.

As these examples illustrate, the existence of formulas with limited use increases the expressiveness of linear logic programs. Along with that increase in expressiveness comes an increase in the cost of doing proof search. In particular, consider the $\multimap L$ inference rule from Figure 4, namely,

$$\dfrac{\Gamma ; \Delta_1 \vdash B_1 \qquad \Gamma ; \Delta_2, B_2 \vdash E}{\Gamma ; \Delta_1, \Delta_2, B_1 \multimap B_2 \vdash E} \multimap L.$$

When reading this inference from bottom to top, one must decide to take the side-formulas to $B_1 \multimap B_2$ within the bounded context, say Δ, and split that multiset into

Δ_1 (the formulas given to the left-premise) and Δ_2 (the formulas given to the right-premise). If Δ contains n occurrences of distinct formulas, then there are 2^n possible splittings available. It is possible to implement interpreters for linear logic programming languages in which Δ is not split immediately into Δ_1 and Δ_2 but rather all of Δ is given to, say, the proof attempt on the left premise. If that proof attempt is successful, then Δ_1 is taken to be those formulas consumed from the bounded context during that attempt. The formulas that result from removing Δ_1 from Δ yield Δ_2. Such a lazy splitting approach has been used in various implementations of linear logic programming languages (Hodas and Miller 1994; Cervesato et al. 2000).

As we have seen, much of the novel expressiveness of linear logic programmings comes from their ability to express multiset rewriting. In the specifications we have presented above, there is, however, only one multiset that is subjected to rewriting, and that is the multiset that forms the zone Δ in sequents of the form $\Gamma ; \Delta \vdash E$. An even more expressive framework would allow the left-hand side to be divided into multiple zones representing multiple multisets. In that setting, different parts of a logic program could use different multisets for different purposes. It turns out that just such multiple-zone sequents are possible in linear logic by noting that the exponentials of linear logic are not *canonical* logical connectives. To explain what we mean by *canonical*, consider adding to linear logic the logical connective $\&'$ which has the same inference rules that exist for $\&$. In such an extended logic and proof system, it is easy to prove that $B \& C$ and $B \&' C$ are logically equivalent formulas. As a result, we say that $\&$ is a *canonical logical connective*. Nothing is gained by adding such a variant of $\&$.

It is easy to show that all the connectives of linear logic are canonical except for the exponentials ! and ?. That is, if we add a blue $!^b$ and a red $!^r$ to linear logic and give them each the same inference rules that exist for !, we then have a more expressive logic. Furthermore, it is possible to allow explicitly the contraction and weakening rules to be applicable for formulas explicitly marked by, say, $!^b$ but not for $!^r$. Danos *et al.* (1993) proposed a linear logic system with such non-canonical exponentials and illustrated their uses in the framework of the Curry-Howard correspondence. These non-canonical exponentials are now called *subexponentials* (Nigam and Miller 2009). As we have seen, the difference between the two zones on the left of sequents in $\Gamma ; \Delta \vdash E$ comes down to the fact that the formulas in Γ should be considered as having ! attached to them while the formulas in Δ do not have ! attached. Thus, the existence of n different subexponentials can now encode $n+1$ zones on the left-hand side of sequents, and some of these zones will allow weakening and contraction (such as the Γ zone), and other zones will allow neither of these structural rules (such as the Δ zone). Similarly, expressions of the form $!^b B \multimap C$ and $!^r B \multimap C$ would provide new kinds of implications. The additional expressiveness of subexponentials in the logic programming setting has been developed a great deal in recent years: see the papers (Nigam 2009; Nigam and Miller 2009; Chaudhuri 2010a; Nigam et al. 2011; Olarte et al. 2015; Despeyroux et al. 2016; Kanovich et al. 2019). Subexponentials have also been used to encode concurrent process calculi (Nigam et al. 2017) and aspects of Milner's bigraphs (Chaudhuri and Reis 2015).

$$\frac{}{\Sigma\colon\ \Gamma\,;\cdot \Downarrow A \vdash A}\ init \qquad \frac{\Sigma\colon\ \Gamma\,;\Delta \Downarrow B \vdash A}{\Sigma\colon\ \Gamma\,;\Delta, B \vdash A}\ \text{decide} \qquad \frac{\Sigma\colon\ \Gamma, B\,;\Delta \Downarrow B \vdash A}{\Sigma\colon\ \Gamma, B\,;\Delta \vdash A}\ \text{decide!}$$

$$\frac{\Sigma\colon\ \Gamma\,;\cdot \vdash B_1 \qquad \Sigma\colon\ \Gamma\,;\Delta \Downarrow B_2 \vdash A}{\Sigma\colon\ \Gamma\,;\Delta \Downarrow B_1 \supset B_2 \vdash A}\ \supset L \qquad \frac{\Sigma\colon\ \Gamma, B_1\,;\Delta \vdash B_2}{\Sigma\colon\ \Gamma\,;\Delta \vdash B_1 \supset B_2}\ \supset R$$

$$\frac{\Sigma\colon\ \Gamma\,;\Delta_1 \vdash B_1 \qquad \Sigma\colon\ \Gamma\,;\Delta_2 \Downarrow B_2 \vdash A}{\Sigma\colon\ \Gamma\,;\Delta_1, \Delta_2 \Downarrow B_1 \multimap B_2 \vdash A}\ \multimap L \qquad \frac{\Sigma\colon\ \Gamma\,;\Delta, B_1 \vdash B_2}{\Sigma\colon\ \Gamma\,;\Delta \vdash B_1 \multimap B_2}\ \multimap R$$

$$\frac{\Sigma \vdash t : \tau \qquad \Sigma\colon\ \Gamma\,;\Delta \Downarrow [t/x]B \vdash A}{\Sigma\colon\ \Gamma\,;\Delta \Downarrow \forall_\tau x.B \vdash A}\ \forall L \qquad \frac{\Sigma, y : \tau\colon\ \Gamma\,;\Delta \vdash [y/x]B}{\Sigma\colon\ \Gamma\,;\Delta \vdash \forall_\tau x.B}\ \forall R$$

Fig. 5. The $\mathbf{LL^f}$ proof system. In these sequents, A denotes a syntactic variable for atomic formulas.

8 Focusing and polarities

8.1 Extending two phases to linear logic

Once Girard introduced linear logic in (1987), Andreoli generalized the two-phase structure of uniform proofs (see Section 5.2) with the design of a *focused proof system* for linear logic (Andreoli 1990; Andreoli 1992). Two important insights distinguish focused proofs from uniform proofs. First, Andreoli's original focused proof system was defined for linear logic, which is more expressive than intuitionistic logic and contains an involutive negation. Second, and more importantly, Andreoli's phases were based on the notion of *invertibility* and *non-invertibility*. From the proof search point-of-view, invertible rules can be applied in a *don't-care-nondeterministic* fashion, whereas the non-invertible rules can be applied in a *don't-know-nondeterministic* fashion. As it turns out, the distinction between invertible and non-invertible inference rules is more fundamental than the distinction between left-hand side and right-hand side, especially in linear logic where the systematic use of negation means that all sequents can be assumed to be one-sided.

Figure 5 presents a focused version of the $\mathbf{LL'}$ proof system of Figure 4. This proof system, which is a subset of the $\mathcal{F}$ proof system in (Miller 1996), contains two kinds of sequents. Sequents of the form $\Sigma\colon\ \Gamma\,;\Delta \vdash B$ are essentially the sequents that appear in $\mathbf{LL'}$ but in the $\mathbf{LL^f}$ sequents of this style can only be the conclusion of right-introduction rules or the decide or decide! rules. The second kind of sequent is of the form

$$\Sigma\colon\ \Gamma\,;\Delta \Downarrow B \vdash A$$

where A is an atomic formula. Here, the $\Downarrow$ provides the left-hand side of a sequent with an additional zone between $\Downarrow$ and $\vdash$: this new zone always contains exactly one formula. Sequents containing a $\Downarrow$ are called *focused sequents* and they can only be the conclusion of left-introduction rules or the *init* rule. Thus, we can see two phases in focused proof construction. One phase involves only sequents containing $\Downarrow$ and having an atomic right-hand side. The other phase involves only sequents that do not contain $\Downarrow$. If we revisit the derivation in Section 5.2 containing underlined formulas, it is easy to rewrite that derivation in the $\mathbf{LL^f}$ proof system in such a way that the underlined formulas correspond to the formulas next to the $\Downarrow$. The $\Downarrow$-phase corresponds to backward chaining and the phase without the $\Downarrow$ corresponds to the goal-reduction phase of uniform proofs.

Andreoli's focused proof system was for a version of linear logic that did not include the $\multimap$ and $\supset$ implications and, as a result, that proof system was one-sided (all formulas are placed on the right of the sequent arrow). However, it is possible to revise Andreoli's proof system to include both implications and identify the $\Downarrow$-phase with left-introduction rules and the $\Downarrow$-free phase with right-introduction rules. This reorganization of focused linear logic proofs is called the Forum logic programming language (Miller 1994; Miller 1996). This presentation of linear logic allows one to view logic programming using Horn clauses (Section 5.2), hereditary Harrop formulas (Section 5.4), and Lolli (Section 7) all as subsets of just the one, large logic programming language. The Forum presentation of linear logic allows us to conclude that all of linear logic is an abstract logic programming language (Miller 1996).

8.2 The dynamics of an abstract logic programming language

Recall that sequents are used to capture the state of a logic programming computation: that is, the sequent $\Sigma\colon\ \mathcal{P} \vdash G$ represents a configuration where the current logic program is $\mathcal{P}$, the current goal is G, and the current signature of eigenvariables (scoped constants) is Σ. A natural and high-level characterization of logic programming languages is captured by the question: How richly can these configurations change during the search for a proof? In other words, if $\Sigma\colon\ \mathcal{P} \vdash G$ is the root of a derivation and if $\Sigma'\colon\ \mathcal{P}' \vdash G'$ is a sequent occurring above the root in that derivation, what is the relationship between Σ and Σ', between $\mathcal{P}$ and $\mathcal{P}'$, and between G and G'? Focused proof systems, such as the $\mathbf{LL^f}$ proof system of Figure 5, provides a natural and simple way to answer this question. In particular, call a sequent of the form $\Sigma\colon\ \Gamma\,;\Delta \vdash A$, for atomic A, a *border sequent*. Such sequents occur between the goal reduction phase and the backward-chaining phase. We can limit our questions about dynamics to just such border sequents:

> if $\Sigma\colon\ \Gamma\,;\Delta \vdash A$ is the root of a derivation and if $\Sigma'\colon\ \Gamma'\,;\Delta' \vdash A'$ is a border sequent occurring above the root in that derivation, what is the relations between Σ and Σ', between Γ and Γ', between Δ and Δ', and between A and A'?

If Γ is a multiset of Horn clauses, then we can immediately say that $\Sigma = \Sigma'$, $\Gamma = \Gamma'$, $\Delta = \Delta'$ and Δ must be, in fact, the empty multiset. Only the relationship between A and A' can be rich. That is, the left-hand side of the sequent is constant and global during the entire computation. The only dynamics of computation must take place within atomic formulas.

If Γ is a multiset of hereditary Harrop formulas, then we can immediately say that $\Sigma \subseteq \Sigma'$, $\Gamma \subseteq \Gamma'$, $\Delta = \Delta'$ and Δ must be, in fact, the empty multiset. Thus, slightly richer dynamics can take place in this setting since both the signature and the logic program can grow as proof search progresses.

If Γ is a multiset of any formulas using $\forall$, $\supset$, and $\multimap$, then we can immediately say again that $\Sigma \subseteq \Sigma'$ and $\Gamma \subseteq \Gamma'$ hold but that there is no simple relationship between Δ and Δ'. The relationship between these two multiset sets can be, essentially, arbitrary and depends on the nature of the logic programs available in that sequent.

8.3 Polarization applied to classical and intuitionistic logics

A standard presentation of linear logic does not involve implications (neither $\supset$ nor $\multimap$) and, as such, a sequent calculus for it can use one-sided sequents. In that setting, it turns

$$\frac{\Gamma, B_1 \vdash B_2}{\Gamma \vdash B_1 \supset B_2} \qquad \frac{\Gamma \vdash [y/x]B}{\Gamma \vdash \forall x.B}\ y \text{ not free in conclusion}$$

$$\frac{\Gamma \Downarrow [t/x]B \vdash A}{\Gamma \Downarrow \forall x.B \vdash A} \qquad \frac{\Gamma \vdash B_1 \Downarrow \qquad \Gamma \Downarrow B_2 \vdash A}{\Gamma \Downarrow B_1 \supset B_2 \vdash A}$$

$$\text{Decide:} \qquad \frac{\Gamma, N \Downarrow N \vdash A}{\Gamma, N \vdash A}\ D_l \qquad \frac{\Gamma \vdash P \Downarrow}{\Gamma \vdash P}\ D_r$$

$$\text{Release:} \qquad \frac{\Gamma, P \vdash A}{\Gamma \Downarrow P \vdash A}\ R_l \qquad \frac{\Gamma \vdash N}{\Gamma \vdash N \Downarrow}\ R_r$$

$$\text{Initial:} \qquad \frac{N \text{ atomic}}{\Gamma \Downarrow N \vdash N}\ I_l \qquad \frac{P \text{ atomic}}{\Gamma, P \vdash P \Downarrow}\ I_r$$

Here, A is an atomic formula (of either polarity), P is a positive (atomic) formula, and N is a negative formula.

Fig. 6. The $\mathbf{LJF}'$ proof system: **LJF** restricted to only $\supset$ and $\forall$.

out that the right-introduction rules for a given connective are invertible if and only if the right-introduction rules for the De Morgan dual of that connective are non-invertible. Such a property suggests introducing the notion of the *polarities* of a logic connective (Girard 1991; Andreoli 1992). In particular, a logical connective is *negative* if its right-introduction rule is invertible, and a logical connective is *positive* if it is the De Morgan dual of a negative connective.

Once polarity and focusing are described in terms of invertibility of inference rules, it is possible to apply them to proof systems in other logics. For example, Girard (1991), Danos et al. (1995), Curien and Munch-Maccagnoni (2010), and Wadler (2003) applied these concepts to classical logic in order to develop well-structured notions of functional programming in classical logic (via the Curry-Howard correspondence).

These concepts have also been applied in intuitionistic logic. For example, Herbelin (1995) and Dyckhoff and Lengrand (2006) developed focused proof systems for intuitionistic logic while the Ph.D. theses of Howe (1998) and Chaudhuri (2006) explored more variations on focused proof systems for both linear and intuitionistic logics. Liang and the author have developed focused proof systems for classical and intuitionistic first-order logics—called **LKF** and **LJF**, respectively—which can account for these various, earlier focused proof systems (Liang and Miller 2007; Liang and Miller 2009).

To illustrate the use of **LJF** in our setting, consider the $\mathbf{LJF}'$ proof system given in Figure 6. This proof system is a subset of **LJF** and resembles the $\mathbf{LL^f}$. There are three kinds of sequents.

1. unfocused sequents: $\Gamma \vdash B$
2. left focused sequents: $\Gamma \Downarrow B \vdash A$, with focus B
3. right focused sequents: $\Gamma \vdash A \Downarrow$, with focus A

Replacing $\Downarrow$ on the left with a comma and dropping $\Downarrow$ on the right yields a regular sequent.

The formulas of $\mathbf{LJF}'$ are given polarity as follows. Since the right rules for $\supset$ and $\forall$ are invertible, formulas of the form $B_1 \supset B_2$ and $\forall x.B$ are *negative*. We shall assign a polarity also to atomic formulas by allowing them to have an arbitrary (but fixed) polarity. Thus, atomic formulas can be either positive or negative. In the more general setting, **LJF** has more positive formulas (including disjunctions and existential quantifiers), but in this setting where we only consider implications and universal quantifiers, only atoms can be positive formulas.

Uniform proofs, when restricted to the logical connectives for implication and universal quantification, correspond to focused proofs where all atomic formulas are polarized negatively.

The following result about **LJF′** follows from the more general results for **LJF** given by Liang and the author (2009). Let B be a first-order formula built from atomic formulas, $\forall$, and $\supset$.

- If $\Gamma \vdash B$ is provable in **LJ** then for every polarization of atomic formulas, the sequent $\Gamma \vdash B$ is provable in **LJF′**.
- If atoms are given some polarization and $\Gamma \vdash B$ is provable in **LJF′**, then $\Gamma \vdash B$ is provable in **LJ**.

An immediate conclusion of this result is that the choice of the polarity of atoms does not affect provability. As we shall see next, that choice can have a big impact on the structure of proofs.

8.4 Characterizing forward and backward chaining

In the Curry-Howard correspondence, different control regimes for evaluation (e.g., call-by-value and call-by-name) can be explained by different choices in polarizations in intuitionistic logic formulas (Brock-Nannestad et al. 2015; Espírito Santo 2016). In the proof search setting, two familiar control strategies—top-down and bottom-up—can similarly be explained by using two different polarizations of atomic formulas with the **LJF** proof system. For example, consider the following partial derivation within **LJF′**.

$$\dfrac{\dfrac{\Xi_1}{\Gamma \vdash r\ a\ b \Downarrow} \quad \dfrac{\dfrac{\Xi_2}{\Gamma \vdash r\ b\ c \Downarrow} \quad \dfrac{\Xi_3}{\Gamma \Downarrow r\ a\ c \vdash A}}{\Gamma \Downarrow r\ b\ c \supset r\ a\ c \vdash A} \supset L}{\dfrac{\Gamma \Downarrow r\ a\ b \supset r\ b\ c \supset r\ a\ c \vdash A}{\Gamma \Downarrow \forall x \forall y \forall z (r\ x\ y \supset r\ y\ z \supset r\ x\ z) \vdash A} \forall L \times 3} \supset L$$

Here, A is some atomic formula, a, b, c are three terms, and the formula under focus in the concluding sequent states that the r relation is transitive. To complete the construction of this focused proof, we need to know the polarity of the atomic formulas $r\ a\ b$, $r\ b\ c$, and $r\ a\ c$. If these atoms have been assigned the negative polarity, then Ξ_3 is the initial rule, and A is $r\ a\ c$. Also, Ξ_1 and Ξ_2 must end with the Release rule. As a result, the inference rule constructed here is the following *backward-chaining* rule:

$$\frac{\Gamma \vdash r\ a\ b \qquad \Gamma \vdash r\ b\ c}{\Gamma \vdash r\ a\ c}$$

On the other hand, if these atoms have been assigned the positive polarity then Ξ_3 must end in the Release rule, and Ξ_1 and Ξ_2 must be the initial rule, which implies that Γ can be written as $\Gamma', r\ a\ b, r\ b\ c$. As a result, the inference rule constructed here is the following *forward-chaining* rule:

$$\frac{\Gamma',\ r\ a\ b,\ r\ b\ c,\ r\ a\ c \vdash A}{\Gamma',\ r\ a\ b,\ r\ b\ c \vdash A}$$

The fact that these two choices of polarity for atoms yield these two styles of inference rules was first published in the papers by Chaudhuri (2006) and Chaudhuri et al. (2008).

It is also possible for some atomic formulas to have positive polarity and some to have negative polarity. For example, if the atoms $r\ a\ b$ and $r\ a\ c$ have negative polarity and $r\ b\ c$ has positive polarity then the inference rule built (from the focused derivation above) is

$$\frac{\Gamma', r\ b\ c \vdash r\ a\ b}{\Gamma', r\ b\ c \vdash r\ a\ c}$$

The λRCC proof system (Jagadeesan et al. 2005) allows for mixing both forward chaining and backward chaining in a superset of the hereditary Harrop fragment of intuitionistic logic. In that proof system, forward chaining is used to encode constraint propagation as found in concurrent constraint programming, and backward chaining is used to encode goal-directed search as found in λProlog. While the λRCC proof system is not a focusing system explicitly, Liang and the author (2009) showed that it can be accounted for using **LJF** by polarizing the atomic formulas denoting constraints positive and polarizing the remaining atomic formulas negative. Chaudhuri (2010b) also used flexible polarity assignments to model magic set transformations.

Choosing between forward chaining and backward chaining can result in very different-sized proofs. Consider, for example, the following specification of the Fibonacci series as the set $\mathcal{P}$ of three Horn clauses.

$$\mathit{fib}\ 0\ 0, \quad \mathit{fib}\ 1\ 1, \quad \forall n \forall f \forall f' [\mathit{fib}\ n\ f \supset \mathit{fib}\ (n+1)\ f' \supset \mathit{fib}\ (n+2)\ (f+f')]$$

If f_n denotes the n^{th} Fibonacci number then it is easy to prove that $\mathit{fib}\ n\ m$ is provable if and only if $m = f_n$ (assuming a suitable implementation of natural number arithmetic). The impact of polarity assignment is on the structure of proofs. In particular, if all atomic formulas are made negative, then there exists only one focused proof of $\mathit{fib}\ n\ f_n$: this one uses backward chaining, and its size is exponential in n. On the other hand, if all atomic formulas are made positive, then there is an infinite number of focused proofs, all of which use forward chaining: the smallest such proof has size linear in n.

Consider now the paper by Kowalski (1979) where he proposed the equation

$$\mathit{Algorithm} = \mathit{Logic} + \mathit{Control}.$$

One component for controlling proof search with Horn clauses was identified in that paper as "Direction (e.g., top-down or bottom-up)". In the early literature on logic programming, the connection between top-down and bottom-up search in Horn clauses and resolution was known to be related to hyper-resolution (for bottom-up) and SLD-resolution (for top-down). (See (Warren 2018) for a description of how these two forms of search have been integrated into a Prolog system using a tabling mechanism.) As the discussion in this section makes clear, this particular component of control now has a rather elegant proof-theoretic explanation: within a focused proof system, choose negative polarization for atoms to specify top-down (backward chaining) or choose positive polarization for atoms to specify bottom-up (forward chaining). Choosing a mixture of positive and negative polarity for atoms yields a mixture of these two search strategies.

Other aspects of control (of which there are many) are not captured by focusing classical, intuitionistic, and linear logics. For example, the left-to-right ordering of conjunctive goals is not captured by focusing alone. For that, there have been some results surrounding non-commutative logic (Lambek 1958; Retoré 1997; Abrusci and Ruet 1999; Polakow and Pfenning 1999; Guglielmi 2007) and associated logic programming languages (Ruet and Fages 1997; Polakow 2001). Still another aspect of control in logic programming is to allow certain special goals to be treated as *constraints* that can be delayed and solved by external solvers (Jaffar and Lassez 1987). This approach to constraints has been effectively implemented in numerous Prolog systems, such as SWI-Prolog (Wielemaker et al. 2012) and in the Elpi implementation of λProlog (Guidi et al. 2019).

9 Advantages for connecting logic programming to proof theory

Using proof theory as a framework for describing and studying logic programming has at least the following benefits.

1. This framework has allowed researchers to extend the role of logic in logic programming beyond first-order Horn clauses to include much richer logics involving higher-order quantification, intuitionistic logic, and linear logic.
2. This framework also makes it possible to see the simpler logic programs as part of a richer logic (in the survey here, that logic is linear logic).
3. Proof theory has also made it possible to vividly compare the nature of functional programming (as proof normalization) and logic programming (as proof search).
4. Given the often close relationship between type theory and the proof theory of intuitionistic logic, there has been a strong flow of design principles and implementation techniques from logic programming to type systems: examples of such a flow can be found in (Pfenning 1988; Elliott 1989; Pfenning 1989; Felty and Miller 1990; Pfenning 1991; Cervesato and Pfenning 1996).
5. A satisfactory proof-theoretic treatment of Clark's program completion (Clark 1978) was developed in the early 1990s using inference rules that worked directly with equality and fixed points (Girard 1992; Schroeder-Heister 1993; McDowell and Miller 2000). Those innovations allow sequent calculus proof systems to capture not only negation-as-finite-failure but also a range of model checking problems (Heath and Miller 2019).

Proving that cut elimination holds for a given sequent calculus proof system is probably the most important meta-theoretical result for such a proof system. The cut-elimination theorem usually implies the consistency of the logical system described by the proof system, and it is usually the starting point for describing proof search strategies. It has also been used to help in reasoning about logic programs as well. For example, *collection analysis* of Horn clause logic programs can be done statically using linear logic and cut-elimination (Miller 2008). The Abella theorem prover (Baelde et al. 2014) encodes a two-level logic approach to reasoning about computation (Gacek et al. 2012). One of these logic levels is for the logic programs used to specify computation; the second logic level captures the first level's metatheory using induction and coinduction. Two of

Abella's tactics are based on the cut-elimination theorem for the logic specification level. In many meta-theoretic proofs, these cut-elimination-based tactics immediately provide proofs of key substitution lemmas, i.e., lemmas stating that if a certain predicate holds for a term, it also holds for all instances of that term (Gacek et al. 2012).

Another advantage of basing a programming language within proof theory is that complexity results regarding proof theory can be immediately applied to logic programs. In particular, since it is known that any Turing computable function can be computed using first-order Horn clauses (Tärnlund 1977), the first-order fragments of all the logic programming languages we have seen are undecidable, since they all include provability in Horn clauses. When we restrict to propositional logics, we have the following results: satisfiability and provability in propositional Horn clauses is linear time (Dowling and Gallier 1984), provability of propositional hereditary Harrop formulas is polynomial-space complete (Statman 1979), and propositional linear logic is undecidable, even when there are no propositional variables (Lincoln 1995).

In Section 2, we listed several shortcomings of Horn clause logic programming languages, such as Prolog. Some of these shortcomings are addressed, to some degree, by linking logic programming more closely to proof theory. Probably the most significant improvement to the logic programming paradigm is the inclusion of programming level *abstractions*: as we have seen, the sequent calculus supports higher-order programming (Section 5.3), modular program construction (Section 5.4), abstract syntax for data containing bindings (Section 6.2), and abstract data types (Section 6.4). Making a connection to linear logic also allows for certain forms of `assert` and `retract` to be provided (Section 7). The use of the proof-theoretic notion of polarization and focused proof has also provided descriptions of both bottom-up and top-down proof search as well as combinations of these two (Section 8.4).

10 Prospects for logic programming

Logic programs have often been and continue to be deployed to build various kinds of database systems, interpreters of other languages, parsers, and type inference engines: for such examples, see the popular texts (Maier and Warren 1988; O'Keefe 1990; Clocksin and Mellish 1994). Given the prominence of proof theory in this paper, the following comments on the prospects for logic programming are limited to those tasks that demand effective implementation of trustworthy logical deduction.

Traditionally, Prolog has not made a strong commitment to logical correctness given the large number of non-logical primitives in it, ranging from assert/retract, to univ, the cut control operator (!), negation-as-failure, and the absence of the occur-check. Fortunately, more recent logic programming systems have put much more focus on implementing sound logical reasoning. Systems such as Teyjus (Nadathur and Mitchell 1999; Nadathur 2005), Elpi (Dunchev et al. 2015), miniKanren (Friedman et al. 2018), and Makam (Stampoulis and Chlipala 2018), have made logical soundness a goal, at least for core aspects of their implementations.

There are many places in the analysis of software, logic, and proof where logic programming can be applied but where soundness is critical: we expand on several such topics in the rest of this section.

10.1 Theorem proving

Several of today's interactive theorem provers make use of LCF *tactics and tacticals* (Milner 1979; Gordon et al. 1979; Gordon 2000), which are themselves generally implemented using higher-order functional programs. With the advent of higher-order logic programming languages, such as λProlog, the argument has been made that logic programming would make for a more flexible and natural setting to implement such tactics and tacticals, especially since the applications of tactics can fail and require backtracking (Felty 1989; Felty 1993). More recently, the Elpi implementation of λProlog has been integrated into Coq as a plugin (Tassi 2018) and used to help automate aspects of the Coq prover (Tassi 2019).

Early papers, such as those by Stickel (1988) and Wos and McCune (1991), point out the rich connections between logic programming and automated deduction and the cross-fertilizations of implementation techniques between those two domains. Some years later, systems such as leanTAP (Beckert and Posegga 1995; Lisitsa 2003) and leanCop (Hodas and Tamura 2001; Otten and Bibel 2003) were built around the notion of *lean deduction* in which small Prolog programs were capable of capturing sound and complete theorem provers for first-order logic.

10.2 Proof checking

The logic programming paradigm is a natural candidate for performing proof checking (Miller 2017) for several reasons. First, there are many kinds of proof certificates in use these days. In almost all cases, those certificates do not contain all the required details to formally check a proof. Instead, many details are left implicit, and so the proof checker will, in general, need to perform some forms of proof reconstruction. Here, standard logic programming technology—unification and backtracking search—can be employed. For example, a certificate might not contain the actual substitution terms needed to instantiate a quantifier. Logic variables and unification can infer such substitution terms. Similarly, claiming that a goal formula is already present in the context requires an index into the context as the witness of that claim. Backtracking search can also be used to find such a witness. Second, quantificational logic formulas and their proofs often contain variable bindings to capture both quantifiers and eigenvariables. There are several logical frameworks, in particular, higher-order hereditary Harrop formulas (by virtue of being based on Church's STT (1940)) and the LF logical framework (Harper et al. 1993), that provide a purely logic-based representation of such binding structures. Implementations of such frameworks—λProlog and Twelf (Pfenning and Schürmann 1999)—treat binding structures via both unification and backtracking search.

Early use of logic programming to implement proof checkers was explored within the Proof Carrying Code project (Necula 1997; Appel and Felty 1999). In that context, logic programming allowed for compact, flexible, and easy to understand proof checkers. Logic programming was used as the core motivation of the *foundational proof certificate* (FPC) project (Chihani et al. 2017). In that project, a proof certificate can be seen as a data structure that incorporates control information for a simplistic sequent calculus theorem prover. The FPC framework can be used as both a kernel itself (assuming that one is willing to admit a logic programming implementation into the trusted base) or as part of a toolchain that allows for the flexible manipulation of proof certificates. In the latter setting, proof checking can be organized to transform a proof certificate with some details missing into a fully detailed proof structure that could be given to existing and trusted kernels, such as is found in Coq (Blanco et al. 2017).

10.3 Software systems

If we consider programming as merely the activity of "writing and shipping code with the hope that it does not do much harm"—which characterizes much about programming to date—then it seems unlikely that logic programming languages will impact the building of software systems. However, it seems clear that we should broaden the discipline of programming to include many other activities that can improve the quality and correctness of programming. Such activities can include automated testing, various kinds of static analyses, program transformation and refinement, and proving partial or full functional correctness of code. Once we add all of these activities to the programming discipline, then logic programming can play a sizable role since it has important uses in all of these additional activities.

Logic programming had early successful uses in the specification of the operational semantics of programming language using either *structural operational semantics* (Plotkin 1981; Plotkin 2004) or *natural semantics* (Despeyroux 1986; Kahn 1987). For example, the Typol subsystem (Despeyroux 1984; Clément et al. 1985; Despeyroux 1988) of the Mentor (Donzeau-Gouge et al. 1984) and Centaur (Borras et al. 1988) systems compiled both dynamic and static semantic definitions of various programming languages into Prolog in order to generate parsers, type checkers, compilers, interpreters, and debuggers.

Many of the early and most convincing logic programming applications in higher-order, intuitionistic logic involved the mechanization of the meta-theory of functional programming (Hannan 1990; Michaylov and Pfenning 1992; Hannan and Miller 1992; Hannan 1993). Verifiable compilers have been described and implemented in Elf (Hannan and Pfenning 1992) and in λProlog (Whalen 2005; Wang 2016; Wang and Nadathur 2016).

10.4 Reasoning directly with logic programs

Given that the logic programming specifications used to encode programming language semantics and inference rules are concise and based on logic itself, there should be rich ways to reason on such specifications directly. In the context of the Typol system, such reasoning could be done by treating provable atomic goals as belonging to an inductive data type (Despeyroux 1986). A more sophisticated approach to reasoning directly on logic programming has been developed within the Abella theorem prover (Baelde et al. 2014). That prover includes such innovations as the ∇-quantifier (Miller and Tiu 2005; Gacek et al. 2011) and the two-level logic approach to reasoning (Gacek et al. 2012). As we mentioned above, the cut-elimination result for the object-logic (the logic programming specification) is turned into a proof technique in Abella for reasoning about such logic specifications.

10.5 A defense of declarative techniques

One advantage of having a proof theory for logic programming is that it sometimes makes it possible to write compact, high-level specifications for which correctness is easy to establish. At the same time, techniques such as partial evaluation (Lloyd and Shepherdson 1991), program transformation (Pettorossi and Proietti 1994), and various forms of static analysis can often be applied directly to specifications written using logical expressions. As a result, rich manipulations of specifications are possible.

As an example of how such manipulations can be applied to logic specifications in a rich programming language, consider the following example, taken from (Hannan and Miller 1992). The specification of call-by-name evaluation of the untyped λ-terms can be given as a binary relation using two higher-order Horn clauses and two constructors (encoding the untyped λ-terms). Given its simplicity, the correctness of that specification is easy to establish. Since that specification is written as logical formulas, a sequence of transformations can be applied to that specification until it results in the specification of an *abstract machine* in which an argument stack and De Bruijn numerals (De Bruijn 1972) are used encode λ-terms. This latter specification can be written using only first-order (binary) Horn clauses. Given the correctness of the initial specification and correctness of the transformations used, the correctness of the derived low-level specification—which has an effective implementation in Prolog—easily follows.

Similar examples can be found in (Cervesato 1998), where aspects of the Warren abstract machine were developed by the direct manipulation of higher-order logic specifications and in (Pientka 2002), where proof theory techniques helped to design a strategy for tabled evaluation of (higher-order) logic programs.

10.6 Further advances in proof theory

The relationship between logic programming and proof theory is not just in one direction. The author has documented in (Miller 2021) several influences of logic programming research on structural proof theory. One computational feature that is often desired in the logic programming world is *saturation*: that is, one would like to know that forward chaining from a given set of clauses will not yield new atomic facts being derived. Saturation was a key component of the Gamma multiset rewriting programming language (Banâtre and Métayer 1996) and the work on *logical algorithms* (Ganzinger and McAllester 2001; Ganzinger and McAllester 2002; Simmons and Pfenning 2008). Currently, structural proof theory does not appear to have any techniques that can account for saturation.

Answer set programming (ASP), as described in papers by Brewka et al. (2011) and Lifschitz (2008), is a form of declarative programming that describes computation as the construction of stable models (Gelfond and Lifschitz 1988). The operational semantics behind such search resembles Datalog's bottom-up inference along with saturation and the negation-as-failure approach to negation. While proof structures exist in this domain (see, for example, (Marek and Truszczyński 1993) and (Lifschitz 1996)), those structures are seldom related to the proof structures found in structural proof theory (which has been our focus here). Some of the proof-theoretic topics described in Sections 8.4 and 9 might also relate proof structures to ASP. Schubert and Urzyczyn (2018) have considered initial steps in that direction.

Finally, developing model-theoretic semantics for these rich proof-theory-inspired languages is interesting to considered. Lipton and Nieva (2018) have shown how to extend the Kripke λ-models of Mitchell and Moggi (1991) to treat an extension to higher-order hereditary Harrop with constraints. A Kripke-style model for Lolli was given in (Hodas and Miller 1994). While Girard (1987) considered various forms of model theory semantics for linear logic, his models have been hard to apply directly to logic programming: an exception is the paper by Fages et al. (2001).

11 Conclusion

Structural proof theory has played an essential role in understanding the nature and structure of logic programming languages. This role has been significant when one wants to have more expressive, dynamic, and modern versions of Prolog. The proof theory of first-order and higher-order versions of intuitionistic and linear logics have provided designs for logic programming languages that support higher-order and modular programming, abstract data-types, and state. Additionally, the theory of focused proofs provides a satisfying description of how to specify forward chaining and backward chaining during proof search.

In 1991, Peter Schroeder-Heister (1991) and the author (1991b) (independently) wrote opinion pieces in which they proposed that the sequent calculus was an appropriate framework for exploring the semantics of logic in philosophical and computational settings. The goal of those papers was to ensure that the term "semantics" was not just understood in terms of model theory and denotational semantics. This survey outlines the successes and methods that have arisen from using proof theory based on the sequent calculus as a semantic framework for logic programming.

Acknowledgments: I thank Miroslaw Truszczynski and several anonymous reviewers for their comments on an earlier version of this paper.

References

Abrusci, V. M. and Ruet, P. 1999. Non-commutative logic I: The multiplicative fragment. *Annals of Pure and Applied Logic 101,* 1, 29–64.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. 2007. *Compilers: Principles, Techniques, and Tools*, Second ed. Pearson, Addison Wesley.

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* Logic Programming Research Reports and Notes. MIT Press, Cambridge, MA.

Andreoli, J.-M. 1990. Proposal for a synthesis of logic and object-oriented programming paradigms. Ph.D. thesis, University of Paris VI.

Andreoli, J.-M. 1992. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation 2,* 3, 297–347.

Andreoli, J. M. and Pareschi, R. 1991. Linear objects: Logical processes with built-in inheritance. *New Generation Computing 9,* 3-4, 445–473.

Andrews, J. 1997. A logical semantics for depth-first Prolog with ground negation. *Theoretical Computer Science 184,* 1-2, 105–143.

Andrews, P. B. 1971. Resolution in type theory. *J. of Symbolic Logic 36*, 414–432.

Andrews, P. B. 1974. Provability in elementary type theory. *Zeitschrift fur Mathematische Logic und Grundlagen der Mathematik 20*, 411–418.

Andrews, P. B., Cohen, E. L., Miller, D., and Pfenning, F. 1984. Automating higher order logic. In *Automated Theorem Proving: After 25 Years*. American Mathematical Society, Providence, RI, 169–192.

Appel, A. W. and Felty, A. P. 1999. Lightweight lemmas in Lambda Prolog. In *16th International Conference on Logic Programming*. MIT Press, 411–425.

Apt, K. R. and Bol, R. 1994. Logic programming and negation: a survey. *J. of Logic Programming 19-20*, 9–71.

Apt, K. R. and Doets, K. 1994. A new definition of SLDNF-resolution. *The J. of Logic Programming 18,* 2 (Feb.), 177–190.

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

Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., and Wang, Y. 2014. Abella: A system for reasoning about relational specifications. *J. of Formalized Reasoning 7,* 2, 1–89.

Banâtre, J.-P. and Métayer, D. L. 1996. Gamma and the chemical reaction model: ten years after. In *Coordination programming: mechanisms, models and semantics*. World Scientific Publishing, IC Press, 3–41.

Barendregt, H. 1984. The Lambda Calculus: Its Syntax and Semantics. Volume 103 of Studies in Logic and the Foundations of Mathematics, Elsevier, New York.

Barendregt, H., Dekkers, W., and Statman, R. 2013. Lambda Calculus with Types. Perspectives in Logic, Cambridge University Press.

Basold, H., Komendantskaya, E., and Li, Y. 2019. Coinduction in uniform: Foundations for corecursive proof search with Horn clauses. In *28th European Symposium on Programming, ESOP 2019*, L. Caires, Ed. LNCS, vol. 11423. Springer, 783–813.

Beckert, B. and Posegga, J. 1995. leanTAP: Lean tableau-based deduction. *J. of Automated Reasoning 15,* 3 (Dec.), 339–358.

Bezem, M. 2001. An improved extensionality criterion for higher-order logic programs. In *CSL: 15th Workshop on Computer Science Logic*, L. Fribourg, Ed. LNCS, vol. 2142. Springer-Verlag, 203–216.

Blanco, R., Chihani, Z., and Miller, D. 2017. Translating between implicit and explicit versions of proof. In *26th International Conference on Automated Deduction (CADE)*, L. de Moura, Ed. LNCS, vol. 10395. Springer, 255–273.

Börger, E. and Rosenzweig, D. 1995a. A mathematical definition of full Prolog. *Sci. Comput. Program. 24,* 3, 249–286.

Börger, E. and Rosenzweig, D. 1995b. The WAM - definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications, Studies in Computer Science and Artificial Intelligence*, C. Beierle and L. Plümer, Eds. Elsevier Science B.V./North-Holland, 20–90.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. 1988. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*. ACM, Boston, 14–24.

Bowen, K. A. 1982. Programming with full first-order logic. In *Machine Intelligence 10*, Hayes, Michie, and Pao, Eds. Ellis Horwood and John Wiley, 421–440.

Brewka, G., Eiter, T., and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Brock-Nannestad, T., Guenot, N., and Gustafsson, D. 2015. Computation in focused intuitionistic logic. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, M. Falaschi and E. Albert, Eds. ACM, 43–54.

De Bruijn, N. G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae 34,* 5, 381–392.

Bruscoli, P. and Guglielmi, A. 2006. On structuring proof search for first order linear logic. *Theoretical Computer Science 360,* 1-3, 42–76.

Bugliesi, M., Lamma, E., and Mello, P. 1994. Modularity in logic programming. *J. of Logic Programming 19/20*, 443–502.

Buss, S. R. 1998. An Introduction to Proof Theory. In *Handbook of Proof Theory*, S. R. Buss, Ed. Elsevier Science Publishers, Amsterdam, 1–78.

Cellucci, C. 1987. Using full first order logic as a programming language. In *Proceedings of the conference on 'Logic and Computer Science: New Trends and Applications'*. Rendiconti del Seminario Matematico Universitae Politecnico di Torino. Fascicolo Speciale 1987. 115–152.

Cervesato, I. 1998. Proof-theoretic foundation of compilation in logic programming languages. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, J. Jaffar, Ed. 115–129.

Cervesato, I., Hodas, J. S., and Pfenning, F. 2000. Efficient resource management for linear logic proof search. *Theoretical Computer Science 232,* 1-2, 133–163.

Cervesato, I. and Pfenning, F. 1996. A linear logic framework. In *11th Symp. on Logic in Computer Science*. IEEE Computer Society Press, New Brunswick, New Jersey, 264–275.

Chaudhuri, K. 2006. The focused inverse method for linear logic. Ph.D. thesis, Carnegie Mellon University. Technical report CMU-CS-06-162.

Chaudhuri, K. 2010a. Classical and intuitionistic subexponential logics are equally expressive. In *CSL 2010: Computer Science Logic*, A. Dawar and H. Veith, Eds. LNCS, vol. 6247. Springer, Brno, Czech Republic, 185–199.

Chaudhuri, K. 2010b. Magically constraining the inverse method using dynamic polarity assignment. In *Proc. 17th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, C. Fermüller and A. Voronkov, Eds. LNCS, vol. 6397. Springer, Yogyakarta, Indonesia, 202–216.

Chaudhuri, K., Pfenning, F., and Price, G. 2008. A logical characterization of forward and backward chaining in the inverse method. *J. of Automated Reasoning 40,* 2-3 (Mar.), 133–177.

Chaudhuri, K. and Reis, G. 2015. An adequate compositional encoding of bigraph structure in linear logic with subexponentials. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, M. Davis, A. Fehnker, A. McIver, and A. Voronkov, Eds. LNCS, vol. 9450. Springer, 146–161.

Chen, W., Kifer, M., and Warren, D. S. 1993. HILOG: a foundation for higher-order logic programming. *J. of Logic Programming 15,* 3, 187–230.

Cheney, J. and Urban, C. 2004. Alpha-Prolog: A logic programming language with names, binding, and alpha-equivalence. In *Logic Programming, 20th International Conference*, B. Demoen and V. Lifschitz, Eds. LNCS, vol. 3132. Springer, 269–283.

Chihani, Z., Miller, D., and Renaud, F. 2017. A semantic framework for proof evidence. *J. of Automated Reasoning 59,* 3, 287–330.

Church, A. 1940. A formulation of the Simple Theory of Types. *J. of Symbolic Logic 5*, 56–68.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, J. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

Clément, D., Despeyroux, J., Despeyroux, T., Hascoët, L., and Kahn, G. 1985. Natural semantics on the computer. Research Report 416, INRIA, Rocquencourt, France. June.

Clocksin, W. F. and Mellish, C. S. 1994. *Programming in Prolog*, Fourth ed. Springer.

Colmerauer, A. and Roussel, P. 1993. The birth of Prolog. *SIGPLAN Notices 28,* 3 (Mar.), 37–52.

Curien, P. and Munch-Maccagnoni, G. 2010. The duality of computation under focus. In *Theoretical Computer Science - 6th IFIP TC 1/WG 2.2 International Conference*, C. S. Calude and V. Sassone, Eds. IFIP Advances in Information and Communication Technology, vol. 323. Springer, 165–181.

Curry, H. B. 1934. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences 20,* 11, 584–590.

Danos, V., Joinet, J.-B., and Schellinx, H. 1993. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In *Kurt Gödel Colloquium*, G. Gottlob, A. Leitsch, and D. Mundici, Eds. LNCS, vol. 713. Springer, 159–171.

Danos, V., Joinet, J.-B., and Schellinx, H. 1995. LKT and LKQ: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In *Advances in Linear Logic*, J.-Y. Girard, Y. Lafont, and L. Regnier, Eds. Number 222 in London Mathematical Society Lecture Note Series. Cambridge University Press, 211–224.

Despeyroux, J. 1986. Proof of translation in natural semantics. In *1st Symp. on Logic in Computer Science (LICS)*. IEEE, Cambridge, Mass, 193–205.

Despeyroux, J., Felty, A., and Hirschowitz, A. 1995. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*. Springer, 124–138.

Despeyroux, J., Olarte, C., and Pimentel, E. 2016. Hybrid and subexponential linear logics technical report. In *Proceedings of LSFA*. ENTCS, vol. 332. Elsevier.

Despeyroux, T. 1984. Executable specification of static semantics. In *Semantics of Data Types*. LNCS, vol. 173. Springer.

Despeyroux, T. 1988. TYPOL: A formalism to implement natural semantics. Research Report 94, INRIA, Rocquencourt, France. Mar.

Di Cosmo, R. and Miller, D. 2019. *Linear Logic*. The Stanford Encyclopedia of Philosophy (Summer 2019 Edition), Edward N. Zalta (ed.). https://plato.stanford.edu/archives/sum2019/entries/logic-linear/.

Donzeau-Gouge, V., Huet, G., and Kahn, G. 1984. Programming environments based on structured editors: the MENTOR experience. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. McGraw-Hill.

Dowling, W. F. and Gallier, J. H. 1984. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. of Logic Programming* 3, 267–284.

Dunchev, C., Guidi, F., Coen, C. S., and Tassi, E. 2015. ELPI: fast, embeddable, λProlog interpreter. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20*, M. Davis, A. Fehnker, A. McIver, and A. Voronkov, Eds. LNCS, vol. 9450. Springer, 460–468.

Dyckhoff, R. and Lengrand, S. 2006. LJQ: a strongly focused calculus for intuitionistic logic. In *Computability in Europe 2006*, A. Beckmann and *et al.*, Eds. LNCS, vol. 3988. Springer, 173–185.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM 13,* 2, 94–102.

Elliott, C. 1989. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*. LNCS, vol. 355. Springer, 121–136.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *J. of the ACM 23,* 4, 733–742.

Espírito Santo, J. E. 2016. The polarized λ-calculus. In *11th Workshop on Logical and Semantic Frameworks with Applications, LSFA 2016*, V. Nigam and M. Florido, Eds. Electronic Notes in Theoretical Computer Science, vol. 332. Elsevier, 149–168.

Fages, F., Ruet, P., and Soliman, S. 1998. Phase semantics and verification of concurrent constraint programs. In *13th Symp. on Logic in Computer Science*, V. Pratt, Ed. IEEE.

Fages, F., Ruet, P., and Soliman, S. 2001. Linear concurrent constraint programming: Operational and phase semantics. *Information and Computation 165,* 1, 14–41.

Felty, A. 1989. Specifying and implementing theorem provers in a higher-order logic programming language. Ph.D. thesis, University of Pennsylvania.

Felty, A. 1993. Implementing tactics and tacticals in a higher-order logic programming language. *J. of Automated Reasoning 11,* 1, 43–81.

Felty, A. and Miller, A. 1990. Encoding a Dependent-Type λ-Calculus in a Logic Programming Language. In *10th Conf. on Automated Deduction (CADE)*, M. E. Stickel, Ed. LNAI, vol. 449. Springer, 221–235.

Friedman, D. P., Byrd, W. E., Kiselyov, O., and Hemann, J. 2018. *The Reasoned Schemer, Second Edition.* The MIT Press.

Gabbay, D. M. 1985. N-Prolog: An extension of Prolog with hypothetical implication II—logical foundations, and negation as failure. *J. of Logic Programming 2,* 4, 251–283.

Gabbay, D. M. and Reyle, U. 1984. N-Prolog: An extension of Prolog with hypothetical implications. I. *J. of Logic Programming 1*, 319–355.

Gacek, A., Miller, D., and Nadathur, G. 2011. Nominal abstraction. *Information and Computation 209,* 1, 48–73.

Gacek, A., Miller, D., and Nadathur, G. 2012. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning 49,* 2, 241–273.

Gallier, J. H. 1986. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* Harper & Row, 1986.

Ganzinger, H. and McAllester, D. 2002. Logical algorithms. In *Proc. International Conference on Logic Programming (ICLP) 2002.* LNCS, vol. 2401. Springer, 209–223.

Ganzinger, H. and McAllester, D. A. 2001. A new meta-complexity theorem for bottom-up logic programs. In *First International Joint Conference on Automated Reasoning (IJCAR)*, R. Goré, A. Leitsch, and T. Nipkow, Eds. LNCS, vol. 2083. Springer, 514–528.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, R. A. Kowalski and K. Bowen, Eds. The MIT Press, Cambridge, Massachusetts, 1070–1080.

Gentzen, G. 1935. Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, Ed. North-Holland, Amsterdam, 68–131. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969.

Girard, J.-Y. 1971. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symposium*, J. E. Fenstad, Ed. North-Holland, Amsterdam, 63–92.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50,* 1, 1–102.

Girard, J.-Y., Taylor, P., and Lafont, Y. 1989. *Proofs and Types.* Cambridge University Press, 1989.

Girard, J.-Y. 1991. A new constructive logic: classical logic. *Math. Structures in Comp. Science 1*, 255–296.

Girard, J.-Y. 1992. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu.

Gödel, K. 1930. Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatshefte für Mathematik 37,* 1, 349–360.

Gordon, M. 2000. From LCF to HOL: a short history. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. D. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 169–186.

Gordon, M. J., Milner, A. J., and Wadsworth, C. P. 1979. *Edinburgh LCF: A Mechanised Logic of Computation.* LNCS, vol. 78. Springer.

Guglielmi, A. 2007. A system of interaction and structure. *ACM Trans. on Computational Logic 8,* 1 (Jan.), 1–64.

Guidi, F., Coen, C. S., and Tassi, E. 2019. Implementing type theory in higher order constraint logic programming. *Mathematical Structures in Computer Science 29,* 8, 1125–1150.

Gunter, C. A. and Gehlot, V. 1989. Nets as tensor theories. Tech. Rep. MS-CIS-89-68, University of Pennsylvania, Computer and Information Science Department. Oct.

Hagiya, M. and Sakurai, T. 1984. Foundation of logic programming based on inductive definition. *New Generation Computing 2,* 1, 59–77.

Hallnäs, L. and Schroeder-Heister, P. 1990. A proof-theoretic approach to logic programming. I. Clauses as rules. *J. of Logic and Computation 1,* 2, 261–283.

Hallnäs, L. and Schroeder-Heister, P. 1991. A proof-theoretic approach to logic programming. II. Programs as definitions. *J. of Logic and Computation 1,* 5, 635–660.

Hannan, J. 1993. Extended natural semantics. *J. of Functional Programming 3,* 2, 123–152.

Hannan, J. and Miller, D. 1992. From operational semantics to abstract machines. *Mathematical Structures in Computer Science 2,* 4, 415–459.

Hannan, J. and Pfenning, F. 1992. Compiler verification in LF. In *7th Symp. on Logic in Computer Science*. IEEE Computer Society Press, Santa Cruz, California.

Hannan, J. J. 1990. Investigating a proof-theoretic meta-language for functional programs. Ph.D. thesis, University of Pennsylvania.

Haridi, S. and Sahlin, D. 1983. Evaluation of logic programs based on natural deduction (draft). In *Logic Programming Workshop*, L. M. Pereira, A. Porto, L. Monteiro, and M. Figueiras, Eds. Núcleo de Intelligência Artificial, Universidade Nova De Lisboa, Portugal, 560–574.

Harland, J. 1997. On goal-directed provability in classical logic. *Comput. Lang 23,* 2-4, 161–178.

Harland, J., Pym, D., and Winikoff, M. 1996. Programming in Lygon: An overview. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, Eds. M. Wirsing and M. Nivat. LNCS, vol. 1101. Springer, 391–405.

Harper, R., Honsell, F., and Plotkin, G. 1993. A framework for defining logics. *J. of the ACM 40,* 1, 143–184.

Harrop, R. 1960. Concerning formulas of the types $A \rightarrow B \vee C,\ A \rightarrow (Ex)B(x)$ in intuitionistic formal systems. *J. of Symbolic Logic 25*, 27–32.

Heath, Q. and Miller, D. 2019. A proof theory for model checking. *J. of Automated Reasoning 63,* 4, 857–885.

Henkin, L. 1949. The completeness of the first-order functional calculus. *J. of Symbolic Logic 14,* 3, 159–166.

Herbelin, H. 1995. Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes. Ph.D. thesis, Université Paris 7.

Hodas, J. and Miller, D. 1991. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In *6th Symp. on Logic in Computer Science*, G. Kahn, Ed. IEEE, Amsterdam, 32–42.

Hodas, J. and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation 110,* 2, 327–365.

Hodas, J. S. 1994. Logic programming in intuitionistic linear logic: Theory, design, and implementation. Ph.D. thesis, University of Pennsylvania, Department of Computer and Information Science.

Hodas, J. S. and Tamura, N. 2001. lolliCop — A linear logic implementation of a lean connection-method theorem prover for first-order classical logic. In *Proceedings of IJCAR:*

International Joint Conference on Automated Reasoning, R. Goré, A. Leitsch, and T. Nipkow, Eds. LNCS, vol. 2083. Springer, 670–684.

Hofmann, M. 1999. Semantical analysis of higher-order abstract syntax. In *14th Symp. on Logic in Computer Science*. IEEE Computer Society Press, 204–213.

Howard, W. A. 1980. The formulae-as-type notion of construction, 1969. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and R. Hindley, Eds. Academic Press, New York, 479–490.

Howe, J. M. 1998. Proof Search Issues in Some Non-Classical Logics. Ph.D. thesis, University of St Andrews. Available as University of St Andrews Research Report CS/99/1.

Huet, G. and Lang, B. 1978. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica 11*, 31–55.

Huet, G. P. 1973. A mechanization of type theory. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. William Kaufmann, 139–146.

Huet, G. P. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science 1*, 27–57.

ISO.org. 2000. ISO/IEC 13211-2:2000: Information technology — programming languages — Prolog — part 2: Modules.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages*. ACM.

Jagadeesan, R., Nadathur, G., and Saraswat, V. 2005. Testing concurrent systems: An interpretation of intuitionistic logic. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*. LNCS, vol. 3821. Springer, Hyderabad, India, 517–528.

Kahn, G. 1987. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. LNCS, vol. 247. Springer, 22–39.

Kanovich, M., Kuznetsov, S., Nigam, V., and Scedrov, A. 2019. Subexponentials in non-commutative linear logic. *Mathematical Structures in Computer Science 29,* 8, 1217–1249.

Kanovich, M. I. 1995. Petri nets, Horn programs, Linear Logic and vector games. *Annals of Pure and Applied Logic 75,* 1–2, 107–135.

Kobayashi, N. and Yonezawa, A. 1993. ACL - A concurrent linear logic programming paradigm. In *Proceedings of the 1993 International Symposium on Logic Programming*, D. Miller, Ed. MIT Press, 279–294.

Kobayashi, N. and Yonezawa, A. 1994. Asynchronous communication model based on linear logic. *Formal Aspects of Computing 3*, 279–294.

Kowalski, R. A. 1979. Algorithm = Logic + Control. *Communications of the Association for Computing Machinery 22*, 424–436.

Kowalski, R. 1974. Predicate logic as a programming language. *Information Processing 74*, 569–574.

Lambek, J. 1958. The mathematics of sentence structure. *American Mathematical Monthly 65*, 154–169.

Li, B. Z. 1994. A π-calculus specification of Prolog. In *Proc. of European Symposium on Programming (ESOP 1994)*. LNCS, vol. 788. Springer, 379–393.

Liang, C. and Miller, D. 2007. Focusing and polarization in intuitionistic logic. In *CSL 2007: Computer Science Logic*, J. Duparc and T. A. Henzinger, Eds. LNCS, vol. 4646. Springer, 451–465.

Liang, C. and Miller, D. 2009. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science 410,* 46, 4747–4768.

Lifschitz, V. 1996. Foundations of logic programming. In *Principles of Knowledge Representation*, G. Brewka, Ed. CSLI Publications, Stanford, California, 69–127.

Lifschitz, V. 2008. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence*. Vol. 3. AAAI Press, 1594–1597.

Lincoln, P. 1995. Deciding provability of linear logic formulas. In *Proceedings of the workshop on Advances in linear logic*, J.-Y. Girard, Y. Lafont, and L. Regnier, Eds. Cambridge University Press, New York, NY, USA, 197–210.

Lincoln, P. and Saraswat, V. 1993. Higher-order, linear, concurrent constraint programming. ftp://parcftp.xerox.com/pub/ccp/lcc/hlcc.dvi.

Lipton, J. and Nieva, S. 2018. Kripke semantics for higher-order type theory applied to constraint logic programming languages. *Theoretical Computer Science 712*, 1–37.

Lisitsa, A. 2003. λleanTAP: Lean deduction in λProlog. Tech. Rep. ULCS-03-017, University of Liverpool, Department of Computer Science.

Lloyd, J. and Topor, R. 1984. Making Prolog more expressive. *J. of Logic Programming 1,* 3 (Oct.), 225–240.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *The J. of Logic Programming 11,* 3, 217–242.

Loveland, D. W. 1970. A linear format for resolution. In *Symposium on Automatic Demonstration*. Number 125 in Lecture Notes in Mathematics. Springer Verlag, Berlin, 147–162.

Loveland, D. W. 1987. Near-Horn PROLOG. In *Proceedings of the Fourth International Conference on Logic Programming*, J.-L. Lassez, Ed. MIT Press Series in Logic Programming. The MIT Press, Melbourne, 456–469.

Loveland, D. W. and Reed, D. W. 1991. A near-Horn Prolog for compilation. In *Computational Logic - Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, 542–564.

Luckham, D. 1970. Refinement theorems in resolution theory. In *Symposium on Automatic Demonstration*. Number 125 in Lecture Notes in Mathematics. Springer Verlag, Berlin, 163–190.

Maier, D. and Warren, D. S. 1988. *Computing with Logic: Logic Programming with Prolog.* Addison-Wesley, Reading, MA.

Marek, W. and Truszczyński, M. 1993. *Nonmonotonic logics: context-dependent reasoning.* Berlin: Springer-Verlag.

Martin-Löf, P. 1971. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In *Proceedings of the Second Scandinavian Logic Symposium*, J. E. Fenstad, Ed. Studies in Logic and the Foundations of Mathematics, vol. 63. North-Holland, 179–216.

Martin-Löf, P. 1982. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*. North-Holland, Amsterdam, 153–175.

McCarty, L. T. 1988a. Clausal intuitionistic logic I. Fixed point semantics. *J. of Logic Programming 5*, 1–31.

McCarty, L. T. 1988b. Clausal intuitionistic logic II. Tableau proof procedure. *J. of Logic Programming 5*, 93–132.

McDowell, R. and Miller, D. 2000. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science 232*, 91–119.

Michaylov, S. and Pfenning, F. 1992. Natural semantics and some of its meta-theory in Elf. In *Extensions of Logic Programming*, L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, Eds. LNCS, vol. 596. Springer, 299–344.

Miller, D. 1989. A logical analysis of modules in logic programming. *J. of Logic Programming 6,* 1-2, 79–108.

Miller, D. 1991a. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation 1,* 4, 497–536.

Miller, D. 1991b. Proof theory as an alternative to model theory. *Newsletter of the Association for Logic Programming 4,* 3 (Aug.). Guest editorial.

Miller, D. 1992. Unification under a mixed prefix. *J. of Symbolic Computation 14,* 4, 321–358.

Miller, D. 1993. The π-calculus as a theory in linear logic: Preliminary results. In *3rd Workshop on Extensions to Logic Programming*, E. Lamma and P. Mello, Eds. LNCS, vol. 660. Springer, Bologna, Italy, 242–265.

Miller, D. 1994. A multiple-conclusion meta-logic. In *9th Symp. on Logic in Computer Science*, S. Abramsky, Ed. IEEE Computer Society, Paris, 272–281.

Miller, D. 1996. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science 165,* 1, 201–232.

Miller, D. 2003. Encryption as an abstract data-type: An extended abstract. In *Proceedings of FCS'03: Foundations of Computer Security*, I. Cervesato, Ed. ENTCS, vol. 84. Elsevier, 18–29.

Miller, D. 2004. Overview of linear logic programming. In *Linear Logic in Computer Science*, T. Ehrhard, J.-Y. Girard, P. Ruet, and P. Scott, Eds. London Mathematical Society Lecture Note, vol. 316. Cambridge University Press, 119–150.

Miller, D. 2006. Representing and reasoning with operational semantics. In *Proceedings of IJCAR: International Joint Conference on Automated Reasoning*, U. Furbach and N. Shankar, Eds. LNAI, vol. 4130. Springer, 4–20.

Miller, D. 2008. A proof-theoretic approach to the static analysis of logic programs. In *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday.* Number 17 in Studies in Logic. College Publications, 423–442.

Miller, D. 2017. Proof checking and logic programming. *Formal Aspects of Computing 29,* 3, 383–399.

Miller, D. 2019. Mechanized metatheory revisited. *J. of Automated Reasoning 63,* 3, 625–665.

Miller, D. 2021. Reciprocal influences between logic programming and proof theory. *Philosophy & Technology 34,* 1 (Mar.), 75–104.

Miller, D. and Nadathur, G. 1986a. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, E. Shapiro, Ed. LNCS, vol. 225. Springer, London, 448–462.

Miller, D. and Nadathur, G. 1986b. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, Morristown, New Jersey, 247–255.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic.* Cambridge University Press.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51,* 1–2, 125–157.

Miller, D. and Tiu, A. 2005. A proof theory for generic judgments. *ACM Trans. on Computational Logic 6,* 4 (Oct.), 749–783.

Milner, R. 1979. LCF: A way of doing proofs with a machine. In *International Symposium on Mathematical Foundations of Computer Science*. Springer, 146–159.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML.* MIT Press.

Minker, J. and Rajasekar, A. 1990. A fixpoint semantics for disjunctive logic programs. *J. of Logic Programming 9,* 1, 45–74.

Mints, G. and Tyugu, E. 1990. Propositional logic programming and Priz system. *J. Logic Programming 9,* 2&3, 179–193.

Mitchell, J. C. and Moggi, E. 1991. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic 51,* 1-2, 99–124.

Nadathur, G. 1987. A higher-order logic as the basis for logic programming. Ph.D. thesis, University of Pennsylvania.

Nadathur, G. 1993. A proof procedure for the logic of hereditary Harrop formulas. *J. of Automated Reasoning 11,* 1, 115–145.

Nadathur, G. 1998. Uniform provability in classical logic. *J. of Logic and Computation 8,* 2, 209–229.

Nadathur, G. 2000. Correspondences between classical, intuitionistic and uniform provability. *Theoretical Computer Science 232*, 273–298.

Nadathur, G. 2005. A treatment of higher-order features in logic programming. *Theory and Practice of Logic Programming 5,* 3, 305–354.

Nadathur, G. and Loveland, D. 1995. Uniform proofs and disjunctive logic programming. In *10th Symp. on Logic in Computer Science*, D. Kozen, Ed. IEEE Computer Society Press, San Diego, California, 148–155.

Nadathur, G. and Loveland, D. W. 1998. Proof procedures for logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, Eds. Vol. 5. Clarendon Press, Oxford, 163–234.

Nadathur, G. and Miller, D. 1988. An Overview of λProlog. In *Fifth International Logic Programming Conference.* MIT Press, Seattle, 810–827.

Nadathur, G. and Miller, D. 1990. Higher-order Horn clauses. *J. of the ACM 37,* 4, 777–814.

Nadathur, G. and Mitchell, D. J. 1999. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In *16th Conf. on Automated Deduction (CADE)*, H. Ganzinger, Ed. LNAI, vol. 1632. Springer, Trento, 287–291.

Nadathur, G. and Pfenning, F. 1992. The type system of a higher-order logic programming language. In *Types in Logic Programming*, F. Pfenning, Ed. MIT Press, 245–283.

Necula, G. C. 1997. Proof-carrying code. In *Conference Record of the 24th Symposium on Principles of Programming Languages 97.* ACM, Paris, France, 106–119.

Negri, S. and von Plato, J. 2001. *Structural Proof Theory.* Cambridge University Press.

Nigam, V. 2009. Exploiting non-canonicity in the sequent calculus. Ph.D. thesis, Ecole Polytechnique.

Nigam, V. and Miller, D. 2009. Algorithmic specifications in linear logic with subexponentials. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP).* ACM, 129–140.

Nigam, V., Olarte, C., and Pimentel, E. 2017. On subexponentials, focusing and modalities in concurrent systems. *Theoretical Computer Science 693*, 35–58.

Nigam, V., Pimentel, E., and Reis, G. 2011. Specifying proof systems in linear logic with subexponentials. In *Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop (LSFA 2010).* ENTCS, vol. 269. Elsevier, 109–123.

Nipkow, T. 1993. Functional unification of higher-order patterns. In *8th Symp. on Logic in Computer Science*, M. Vardi, Ed. IEEE, 64–74.

O'Keefe, R. A. 1990. *The Craft of Prolog.* MIT Press.

Olarte, C., Nigam, V., and Pimentel, E. 2015. Subexponential concurrent constraint programming. *Theoretical Computer Science 606*, 98–120.

Ono, H. 1998. Proof-theoretic methods in nonclassical logic. an introduction. In *Theories of Types and Proofs*, M. Takahashi, M. Okada, and M. Dezani-Ciancaglini, Eds. Mathematical Society of Japan Memoirs, vol. 2. Mathematical Society of Japan, 207–254.

Otten, J. and Bibel, W. 2003. `leanCoP`: lean connection-based theorem proving. *J. of Symbolic Computation 36,* 1–2, 139–161.

Paulson, L. C. 1989. The foundation of a generic theorem prover. *J. of Automated Reasoning 5*, 363–397.

Perlis, A. J. 1982. Epigrams on programming. *ACM SIGPLAN Notices 17,* 9 (Sept.), 7–13.

Pettorossi, A. and Proietti, M. 1994. Transformation of logic programs: Foundations and techniques. *J. of Logic Programming 19/20*, 261–320.

Pfenning, F. 1988. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Lisp and Functional Programming Conference*. ACM.

Pfenning, F. 1989. Elf: A language for logic definition and verified metaprogramming. In *4th Symp. on Logic in Computer Science*. IEEE, Monterey, CA, 313–321.

Pfenning, F. 1991. Logic programming in the LF logical framework. In *Logical Frameworks*, G. Huet and G. Plotkin, Eds. Cambridge University Press, 149–181.

Pfenning, F. and Elliott, C. 1988. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 199–208.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf — A meta-logical framework for deductive systems. In *16th Conf. on Automated Deduction (CADE)*, H. Ganzinger, Ed. LNAI, vol. 1632. Springer, Trento, 202–206.

Pientka, B. 2002. A proof-theoretic foundation for tabled higher-order logic programming. In *18th International Conference on Logic Programming (ICLP 2002)*, P. J. Stuckey, Ed. LNCS, vol. 2401. Springer, 271–286.

Pitts, A. M. 2003. Nominal logic, A first order theory of names and binding. *Information and Computation 186,* 2, 165–193.

Plotkin, G. D. 1981. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark. Sept.

Plotkin, G. D. 2004. A structural approach to operational semantics. *J. of Logic and Algebraic Programming 60-61*, 17–139.

Polakow, J. 2001. Ordered linear logic and applications. Ph.D. thesis, Department of Computer Science, Caregnie Mellon.

Polakow, J. and Pfenning, F. 1999. Natural deduction for intuitionistic non-commutative linear logic. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, J.-Y. Girard, Ed. LNCS, vol. 1581. Springer, 295–309.

Pym, D. and Harland, J. 1994. The uniform proof-theoretic foundation of linear logic programming. *J. of Logic and Computation 4,* 2, 175–207.

Qi, X., Gacek, A., Holte, S., Nadathur, G., and Snow, Z. 2015. The Teyjus system – version 2. http://teyjus.cs.umn.edu/.

Retoré, C. 1997. Pomset logic: a non-commutative extension of classical linear logic. In *Proceedings of TLCA*. Vol. 1210. Springer, 300–318.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. of the ACM 12*, 23–41.

Ruet, P. and Fages, F. 1997. Concurrent constraint programming and non-commutative logic. In *Proceedings of the* 11^{th} *Conference on Computer Science Logic*, M. Nielsen and W. Thomas, Eds. LNCS, vol. 1414. Springer.

Saraswat, V. 1993. A brief introduction to linear concurrent constraint programming. Tech. rep., Xerox Palo Alto Research Center.

Schroeder-Heister, P. 1991. Uniform proof-theoretic semantics for logical constants (abstract). *J. of Symbolic Logic 56*, 1142.

Schroeder-Heister, P. 1993. Rules of definitional reflection. In *8th Symp. on Logic in Computer Science*, M. Vardi, Ed. IEEE Computer Society Press, IEEE, 222–232.

Schubert, A. and Urzyczyn, P. 2018. First-order answer set programming as constructive proof search. *Theory Pract. Log. Program. 18,* 3-4, 673–690.

Schwichtenberg, H. 2006. Minlog. In *The Seventeen Provers of the World*, F. Wiedijk, Ed. LNCS, vol. 3600. Springer, 151–157.

Scott, D. 1970. Outline of a mathematical theory of computation. In *Proceedings, 4th Annual*

Princeton Conference on Information Sciences and Systems. Princeton University, 169–176. Also, Programming Research Group Technical Monograph PRG–2, Oxford University.

Simmons, R. J. and Pfenning, F. 2008. Linear logical algorithms. In *35th International Colloquium Automata, Languages and Programming (ICALP 2008)*, L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, Eds. LNCS, vol. 5126. Springer, 336–347.

Sørensen, M. H. and Urzyczyn, P. 2006. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic, vol. 149. Elsevier.

Stampoulis, A. and Chlipala, A. 2018. Prototyping a functional language using higher-order logic programming: a functional pearl on learning the ways of λProlog/Makam. *Proc. ACM Program. Lang. 2,* ICFP, 93:1–93:30.

Statman, R. 1979. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science 9*, 67–72.

Stickel, M. 1988. A PROLOG technology theorem prover: Implementation by an extended PROLOG compiler. *J. of Automated Reasoning 4,* 4, 353–380.

Stoy, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA.

Takahashi, M. 1967. A proof of cut-elimination theorem in simple type theory. *J. of the Mathematical Society of Japan 19*, 399–410.

Takeuti, G. 1953. On a generalized logic calculus. *Japanese Journal of Mathematics 23*, 39–96. Errata: ibid, vol. 24 (1954), 149–156.

Tärnlund, S.-A. 1977. Horn Clause Computability. *BIT 17*, 215–226.

Tassi, E. 2018. Elpi: an extension language for Coq. CoqPL 2018: The Fourth International Workshop on Coq for Programming Languages.

Tassi, E. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, J. Harrison, J. O'Leary, and A. Tolmach, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 29:1–29:18.

Van Gelder, A., Ross, K. A., and Schilpf, J. S. 1991. The well-founded semantics for general logic programs. *J. of the ACM 38,* 3, 620–662.

Wadge, W. W. 1991. Higher-order Horn logic programming. In *Proceedings of the 1991 International Symposium on Logic Programming*. MIT Press, San Diego, California, USA, 289–303.

Wadler, P. 2003. Call-by-value is dual to call-by-name. In *8th Int. Conf. on Functional Programming*. ACM, New York, NY, 189–201.

Wang, Y. 2016. A higher-order abstract syntax approach to the verified compilation of functional programs. Ph.D. thesis, University of Minnesota.

Wang, Y. and Nadathur, G. 2016. A higher-order abstract syntax approach to verified transformations on functional programs. In *Programming Languages and Systems. ESOP 2016*, P. Thiemann, Ed. LNCS, vol. 9632. Springer, 752–779.

Wansing, H. 2002. Sequent systems for modal logics. In *Handbook of Philosophical Logic, Volume VIII*, 2 ed., D. M. Gabbay and F. Guenthner, Eds. Kluwer Academic Publishers, Amsterdam, 61–146.

Warren, D. H. D. 1982. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence 10*. Halsted Press, Chicester, England, 441–454.

Warren, D. H. D. 1983. An abstract Prolog instruction set. Tech. Rep. 309, SRI International.

Warren, D. S. 2018. Top-down and Bottom-up Evaluation Procedurally Integrated. *Theory Pract. Log. Program 18,* 3-4, 706–721.

WHALEN, M. W. 2005. Trustworthy translation for the requirements state machine language without events. Ph.D. thesis, University of Minnesota.

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory Pract. Log. Program 12,* 1-2, 67–96.

WOS, L. AND MCCUNE, W. 1991. Automated theorem proving and logic programming: A natural symbiosis. *J. of Logic Programming 11*, 1–53.

ZAMOV, N. K. AND SHARONOV, V. I. 1971. On a class of strategies for the resolution method. In *Studies in Constructive Mathematics and Mathematical Logic, Part III*, A. O. Slisenko, Ed. Number 16 in Seminars in Mathematics, V.A. Steklov Mathematical Institute, Leningrad. Consultants Bureau, New York, 26–31. Translated from Russian by J. S. Wood.