10 September 2003 Invited tutorial at ICLP '03

A Tutorial on Proof Theoretic Foundations of Logic Programming

Paola Bruscoli and Alessio Guglielmi
Technische Universität Dresden
Hans-Grundig-Str. 25 - 01062 Dresden - Germany
Paola.Bruscoli@Inf.TU-Dresden.DE, Alessio.Guglielmi@Inf.TU-Dresden.DE

Abstract *Abstract logic programming is about designing logic programming languages via the proof theoretic notion of uniform provability. It allows the design of purely logical, very expressive logic programming languages, endowed with a rich meta theory. This tutorial intends to expose the main ideas of this discipline in the most direct and simple way.*

1 Introduction

Logic programming is traditionally introduced as an application of the resolution method. A limitation of this perspective is the difficulty of extending the pure language of Horn clauses to more expressive languages, without sacrificing logical purity.

After the work on abstract logic programming of Miller, Nadathur and other researchers (see especially [7, 8, 6]), we know that this limitation can largely be overcome by looking at logic programming from a proof theoretic perspective, through the idea of *uniform provability*. This way, one can make the declarative and operational meaning of logic programs coincide for large fragments of very expressive logics, like several flavours of higher order logics and linear logics. For these logics, proof theory provides a theoretical support, which is 'declarative' even in absence of a convenient semantics, like in the case of linear logic.

The essential tool is Gentzen's sequent calculus [2, 1], for which one always requires the cut elimination property. Deductive systems can be defined in absence of their semantics, but they automatically possess the properties usually requested by computer scientists: a rigorous definition independent of specific implementations and a rich metatheory allowing formal manipulation. An important advantage of defining logic programming languages in the sequent calculus is the universality of this proof theoretic formalism. This allows for a much easier designing and comparing different languages than the corresponding situation in which a common formalism is not adopted.

Moreover, given the modularity properties induced by cut elimination and its consequences, it is conceivable to design tools for logic programming languages in a modular fashion, which is an obvious advantage. For example, a compiler can be produced by starting from some core sublanguage and proceeding by successive additions corresponding to modularly enlarging the core language. A first large scale language designed according to this principle is λ-Prolog [3], in intuitionistic higher order logic, and several others are available, especially in logics derived from linear logic (see [4] for a survey).

The purpose of this tutorial is to give a concise, direct and simple introduction to logic programming via uniform provability. This is a two phase operation: firstly, the sequent calculus of classical logic is introduced, then uniform provability and the related concept of *abstract logic programming*. The idea is that anybody familiar with classical logic in its basic, common exposition, and who also knows Prolog, should be able to find enough information for understanding how Prolog could be designed, and extended, using this proof theoretic methodology. The reader who is already acquainted with sequent calculus can skip the first section.

2 The Sequent Calculus

We deal here with the sequent calculus of first order classical logic; a good reference is [1]. This is enough to establish all the main ideas of abstract logic programming. For the higher order case, a great source of trouble is dealing with unification, but this issue does not directly affect the notion of uniform provability. The paper [8] studies the sequent calculus of the higher order logic at the basis of λ-Prolog.

2.1 Classical Logic

A sequent is an expression of the form $\Gamma \vdash \Delta$, where Γ and Δ are multisets of formulae. We can think of Γ and Δ as, respectively, a conjunction and a disjunction of formulae. The symbol $\vdash$ is called 'entailment' and corresponds to logical implication. Therefore 'Γ entails Δ' means that from all formulae in Γ some of the formulae in Δ follow.

A rule is a relation among sequents $S_1, \ldots, S_n$ and S

$$\rho \, \frac{S_1 \quad \ldots \quad S_n}{S} \,,$$

where ρ is the name of the rule, S_1, ..., S_n are its premises and S is its conclusion. If $n = 0$ the rule has no premises and we call it an axiom.

A system in the sequent calculus is a finite set of rules, that we can partition as follows: the axiom, the set of left rules and the set of right rules. Rules come in pairs for every logical connective, giving meaning to it depending on whether the connective appears at the left or at the right of the entailment symbol. Some more rules, called 'structural', model basic algebraic properties of sequents, like idempotency; they also belong to the sets of left and right rules. A further rule, called cut, may be present as well.

A derivation for a formula is obtained by successively applying instances of the given rules; if all branches of the derivation tree reach axioms then the derivation is a proof. We take a bottom-up perspective, i.e., we think of proofs as built from the bottom sequent up.

We focus on classical logic, assuming that the reader is acquainted with some basic notions related to substitutions, instantiations and the like. We

Axiom

$$A \vdash A$$

Cut

$$\mathsf{cut}\ \frac{\Gamma \vdash \Delta, A \quad A, \Lambda \vdash \Theta}{\Gamma, \Lambda \vdash \Delta, \Theta}$$

Left Rules

$$\mathsf{c_L}\ \frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta} \qquad \mathsf{w_L}\ \frac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}$$

$$\supset_{\mathsf{L}}\ \frac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Delta}{A \supset B, \Gamma \vdash \Delta}$$

$$\wedge_{\mathsf{LL}}\ \frac{A, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta} \qquad \wedge_{\mathsf{LR}}\ \frac{B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}$$

$$\vee_{\mathsf{L}}\ \frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}$$

$$\neg_{\mathsf{L}}\ \frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}$$

$$\forall_{\mathsf{L}}\ \frac{A[x/t], \Gamma \vdash \Delta}{\forall x.A, \Gamma \vdash \Delta}$$

$$\exists_{\mathsf{L}}\ \frac{A, \Gamma \vdash \Delta}{\exists x.A, \Gamma \vdash \Delta}$$

Right Rules

$$\mathsf{c_R}\ \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A} \qquad \mathsf{w_R}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}$$

$$\supset_{\mathsf{R}}\ \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \supset B}$$

$$\wedge_{\mathsf{R}}\ \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

$$\vee_{\mathsf{RL}}\ \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} \qquad \vee_{\mathsf{RR}}\ \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B}$$

$$\neg_{\mathsf{R}}\ \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}$$

$$\forall_{\mathsf{R}}\ \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall x.A}$$

$$\exists_{\mathsf{R}}\ \frac{\Gamma \vdash \Delta, A[x/t]}{\Gamma \vdash \Delta, \exists x.A}$$

where in $\forall_{\mathsf{R}}$ and $\exists_{\mathsf{L}}$ the variable x is not free in the conclusion

Fig. 1 *First Order Classical Sequent Calculus with Multiplicative Cut* LK

introduce a system for first-order classical logic, called LK, which is a slight modification of the system bearing the same name proposed by Gentzen in 1934. By means of examples we will see how the proof construction process works in the system and we will shortly discuss cut elimination.

From now on we consider the language of classical logic built over $\supset$, $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, and we denote formulae by A, B, ..., and atoms by a, b, ..., and by $p(x)$, $q(y)$. The system LK of *first order classical sequent calculus with multiplicative cut* is defined in fig. 1. The rules $\mathsf{c_L}$ and $\mathsf{c_R}$ are called *left* and *right contraction*; $\mathsf{w_L}$ and $\mathsf{w_R}$ are called *left* and *right weakening*; the other rules, except for *cut*, get the name from the connective they work on.

Regarding the use of the rules $\forall_{\mathsf{R}}$ and $\exists_{\mathsf{L}}$, one should remember that bound variables can always be renamed, i.e., one can consider, say, $\forall x.p(x)$ the same as $\forall y.p(y)$.

The first example shows the behaviour of several rules, in particular it shows how several structural rules operate. To see $\mathsf{c_L}$ in action, please wait until the next section.

2.1.1 Example In LK, the tautologies $A \vee \neg A$ and $\forall x.(q(y) \supset p(x)) \supset (q(y) \supset \forall x.p(x))$ have these proofs:

$$\mathsf{c_R}\,\dfrac{\mathsf{\vee_{RL}}\,\dfrac{\mathsf{\vee_{RR}}\,\dfrac{\mathsf{\neg_R}\,\dfrac{A \vdash A}{\vdash A, \neg A}}{\vdash A, A \vee \neg A}}{\vdash A \vee \neg A, A \vee \neg A}}{\vdash A \vee \neg A} \qquad \text{and} \qquad \mathsf{\supset_R}\,\dfrac{\mathsf{\forall_R}\,\dfrac{\mathsf{\forall_L}\,\dfrac{\mathsf{\supset_L}\,\dfrac{\mathsf{w_R}\,\dfrac{q(y) \vdash q(y)}{q(y) \vdash q(y), p(x)} \quad \mathsf{w_L}\,\dfrac{p(x) \vdash p(x)}{p(x), q(y) \vdash p(x)}}{q(y) \supset p(x), q(y) \vdash p(x)}}{\forall x.(q(y) \supset p(x)), q(y) \vdash p(x)}}{\forall x.(q(y) \supset p(x)), q(y) \vdash \forall x.p(x)}}{\forall x.(q(y) \supset p(x)) \vdash q(y) \supset \forall x.p(x)}}{\vdash \forall x.(q(y) \supset p(x)) \supset (q(y) \supset \forall x.p(x))}\ .$$

The second tautology proven above is an instance of one of Hilbert axiom schemes: $\forall x.(A \supset B) \supset (A \supset \forall x.B)$, where x is not free in A.

In the next example, please pay special attention to the role played by the provisos of the $\forall_\mathsf{R}$ and $\exists_\mathsf{L}$ rules. In case more than one quantifier rule is applicable during the proof search, try first to apply a rule which has a proviso, while building a proof bottom-up, i.e., starting from its bottom sequent.

2.1.2 Example We prove in LK that $(\forall x.p(x) \supset q(y)) \equiv \exists x.(p(x) \supset q(y))$. We need to prove both implications:

$$\mathsf{\supset_R}\,\dfrac{\mathsf{\supset_L}\,\dfrac{\mathsf{\forall_R}\,\dfrac{\mathsf{\supset_R}\,\dfrac{\mathsf{\exists_R}\,\dfrac{\mathsf{w_R}\,\dfrac{p(x) \vdash p(x)}{p(x) \vdash p(x), q(y)}}{\vdash p(x), p(x) \supset q(y)}}{\vdash p(x), \exists x.(p(x) \supset q(y))}}{\vdash \forall x.p(x), \exists x.(p(x) \supset q(y))} \quad \mathsf{\exists_R}\,\dfrac{\mathsf{\supset_R}\,\dfrac{\mathsf{w_L}\,\dfrac{q(y) \vdash q(y)}{p(x), q(y) \vdash q(y)}}{q(y) \vdash p(x) \supset q(y)}}{q(y) \vdash \exists x.(p(x) \supset q(y))}}{\forall x.p(x) \supset q(y) \vdash \exists x.(p(x) \supset q(y))}}{\vdash (\forall x.p(x) \supset q(y)) \supset \exists x.(p(x) \supset q(y))}$$

and

$$\mathsf{\supset_R}\,\dfrac{\mathsf{\exists_L}\,\dfrac{\mathsf{\supset_R}\,\dfrac{\mathsf{\forall_L}\,\dfrac{\mathsf{\supset_L}\,\dfrac{\mathsf{w_R}\,\dfrac{p(x) \vdash p(x)}{p(x) \vdash p(x), q(y)} \quad \mathsf{w_L}\,\dfrac{q(y) \vdash q(y)}{q(y), p(x) \vdash q(y)}}{p(x) \supset q(y), p(x) \vdash q(y)}}{p(x) \supset q(y), \forall x.p(x) \vdash q(y)}}{p(x) \supset q(y) \vdash \forall x.p(x) \supset q(y)}}{\exists x.(p(x) \supset q(y)) \vdash \forall x.p(x) \supset q(y)}}{\vdash \exists x.(p(x) \supset q(y)) \supset (\forall x.p(x) \supset q(y))}\ .$$

To complete the exposition of applications of rules, let's consider the cut rule.

2.1.3 Example In LK the formula $\exists x.\forall y.(p(x) \supset p(y))$ can be proved this way:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(z) \vdash p(z)}{p(z) \vdash p(z), p(y)}\,\mathsf{w}_\mathsf{R}}{\vdash p(z), p(z) \supset p(y)}\,\supset_\mathsf{R}}{\vdash p(z), \forall y.(p(z) \supset p(y))}\,\forall_\mathsf{R}}{\vdash p(z), \exists x.\forall y.(p(x) \supset p(y))}\,\exists_\mathsf{R}}{\vdash \forall z.p(z), \exists x.\forall y.(p(x) \supset p(y))}\,\forall_\mathsf{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(y) \vdash p(y)}{p(x), p(y) \vdash p(y)}\,\mathsf{w}_\mathsf{L}}{p(y) \vdash p(x) \supset p(y)}\,\supset_\mathsf{R}}{\forall z.p(z) \vdash p(x) \supset p(y)}\,\forall_\mathsf{L}}{\forall z.p(z) \vdash \forall y.(p(x) \supset p(y))}\,\forall_\mathsf{R}}{\forall z.p(z) \vdash \exists x.\forall y.(p(x) \supset p(y))}\,\exists_\mathsf{R}}{\vdash \exists x.\forall y.(p(x) \supset p(y)), \exists x.\forall y.(p(x) \supset p(y))}\,\mathsf{cut}}{\vdash \exists x.\forall y.(p(x) \supset p(y))}\,\mathsf{c}_\mathsf{R}$$

The idea behind this proof is to use the cut in order to create two hypotheses, one for each branch the proof forks into. In the right branch we assume $\forall z.p(z)$ and in the left one we assume the contrary, i.e., $\neg\forall z.p(z)$ (note that this is the case when $\forall z.p(z)$ is at the right of the entailment symbol). The contraction rule at the bottom of the proof ensures that each branch is provided with the formula to be proved. Of course, it's easier to prove a theorem when further hypotheses are available (there is more information, in a sense): the cut rule creates such information in pairs of opposite formulae.

One should note that the cut rule is unique in the preceding sense. In fact, all other rules do not create new information. They enjoy the so called *subformula property*, which essentially means that all other rules break formulae into pieces while going up in a proof. In other words, there is no creation of new formulae, only subformulae of available formulae are used (there are some technicalities involved in the $\exists_\mathsf{R}$ rule, but we can ignore them for now).

One should also observe that the cut rule is nothing more than a generalised modus ponens, which is the first inference rule ever appeared in logic. Systems in the sequent calculus are usually given with cut rules, because cut rules simplify matters when dealing with semantics and when relating several different systems together, especially those presented in other styles like natural deduction. In fact, one useful exercise is to try to prove that LK is indeed classical logic. No matter which way one chooses, chances are very good that the cut rule will play a major role. Of course, classical logic is a rather well settled matter, but this is not so for other logics, or for extensions of classical logic, which are potentially very interesting for automated deduction and logic programming. In all cases, the cut rule is a key element when trying to relate different viewpoints about the same logic, like for example syntax and semantics.

2.2 Cut Elimination

Gentzen's great achievement has been to show that the cut rule is not necessary in his calculus: we say that it is *admissible*. In other words, the other rules are sufficient for the completeness of the inference system. This is of course good news for automated deduction. In the example above, a logician who understands the formula to be proved can easily come up with the hypothesis

$\forall z.p(z)$, but how is a computer supposed to do so? The formula used is not in fact a subformula of the formula to be proved. Since it's the only rule not enjoying the subformula property, we could say that the cut rule concentrates in itself all the 'evil'.

Given its capital importance, we state below Gentzen's theorem. The theorem actually gives us more than just the existence of cut free proofs: we could also transform a proof with cuts into a cut free one by way of a procedure (so, the theorem is constructive). This turns out to be fundamental for functional programming, but we will not talk about that in this paper.

2.2.1 Theorem *For every proof in* LK *there exists a proof with the same conclusion in which there is no use of the* cut *rule.*

2.2.2 Example By eliminating from the previous example cut rules according to the cut elimination algorithm you find in [1], you should obtain a much simpler proof, probably less intuitive from the semantic viewpoint:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(y)\vdash p(y)}{p(y),p(x)\vdash p(y)}\,\mathsf{w_L}}{p(y)\vdash p(x)\supset p(y)}\,\supset_{\mathsf{R}}}{p(y)\vdash p(z),p(x)\supset p(y)}\,\mathsf{w_R}}{\vdash p(y)\supset p(z),p(x)\supset p(y)}\,\supset_{\mathsf{R}}}{\vdash \forall z.(p(y)\supset p(z)),p(x)\supset p(y)}\,\forall_{\mathsf{R}}}{\vdash \exists x.\forall y.(p(x)\supset p(y)),p(x)\supset p(y)}\,\exists_{\mathsf{R}}}{\vdash \exists x.\forall y.(p(x)\supset p(y)),\forall y.(p(x)\supset p(y))}\,\forall_{\mathsf{R}}}{\vdash \exists x.\forall y.(p(x)\supset p(y)),\exists x.\forall y.(p(x)\supset p(y))}\,\exists_{\mathsf{R}}}{\vdash \exists x.\forall y.(p(x)\supset p(y))}\,\mathsf{c_R}\ .$$

We should remark that what happened in the previous example is not very representative of cut and cut elimination, with respect to the size of the proofs. In our example, the cut free proof is smaller than the proof with cuts above. In general, cut free proofs can be immensely bigger than proofs with cuts (there is a hyperexponential factor between them).

There is a possibility of exploiting this situation to our advantage. In fact, as we will also see here, cut free proofs correspond essentially to computations. Since proofs with cuts could prove the same theorems in much shorter ways, one could think of using cuts in order to speed up computations, or in order to make them more manageable when analysing them. This would require using cuts as sort of oracles that could divine the right thing to do to shorten a computation at certain given states. This could work by asking a human, or a very sophisticated artificial intelligence mechanism, to make a guess. We will not be dealing with these aspects in this paper, but one should keep them in mind when assessing the merits of the proof theoretic point of view on logic programming and automated deduction, because they open exciting possibilities.

As a last remark, let us notice that cut elimination is not simply a necessary feature for automated deduction, or in general for all what is related to proof

construction: its use is broader. For example, a cut elimination theorem can be used to prove the consistency of a system.

2.2.3 Example Prove that LK is consistent. We shall reason by contradiction. Suppose that LK is inconsistent: we can then find two proofs Π_1 and Π_2 for both $\vdash A$ and $\vdash \neg A$. We can then also build the following proof for $\vdash$:

$$\mathsf{cut}\ \dfrac{\Pi_1 \quad \vdash A \qquad \mathsf{cut}\ \dfrac{\Pi_2 \quad \vdash \neg A \qquad \neg_{\mathsf{L}}\ \dfrac{A \vdash A}{\neg A, A \vdash}}{A \vdash}}{\vdash}\ .$$

By the cut elimination theorem a proof for $\vdash$ must exist in which the cut rule is not used. But this raises a contradiction, since $\vdash$ is not an axiom, nor is the conclusion of any rule of LK other than cut.

2.2.4 Example We can reduce the axiom to atomic form. In fact, every axiom $A \vdash A$ appearing in a proof can be substituted by a proof

$$\Pi_A \quad A \vdash A$$

which only uses axioms of the kind $a \vdash a$, where A is a generic formula and a is an atom. To prove this, consider a sequent of the kind $A \vdash A$ and reason by induction on the number of logical connectives appearing in A.

Base Case If A is an atom then $\Pi_A = A \vdash A$.

Inductive Cases

If $A = B \supset C$ then

$$\Pi_A = \supset_{\mathsf{R}}\ \dfrac{\supset_{\mathsf{L}}\ \dfrac{\mathsf{w}_{\mathsf{R}}\ \dfrac{\Pi_B \quad B \vdash B}{B \vdash B, C} \qquad \mathsf{w}_{\mathsf{L}}\ \dfrac{\Pi_C \quad C \vdash C}{C, B \vdash C}}{B \supset C, B \vdash C}}{B \supset C \vdash B \supset C}$$

All other cases are similar:

$$\wedge_{\mathsf{R}}\ \dfrac{\wedge_{\mathsf{LL}}\ \dfrac{\Pi_B \quad B \vdash B}{B \wedge C \vdash B} \qquad \wedge_{\mathsf{LR}}\ \dfrac{\Pi_C \quad C \vdash C}{B \wedge C \vdash C}}{B \wedge C \vdash B \wedge C}\ , \qquad \vee_{\mathsf{L}}\ \dfrac{\vee_{\mathsf{RL}}\ \dfrac{\Pi_B \quad B \vdash B}{B \vdash B \vee C} \qquad \vee_{\mathsf{RR}}\ \dfrac{\Pi_C \quad C \vdash C}{C \vdash B \vee C}}{B \vee C \vdash B \vee C}\ ,$$

$$\neg_{\mathsf{L}}\dfrac{\neg_{\mathsf{R}}\dfrac{\overset{\Pi_B}{\vartriangle}\ B\vdash B}{\neg B, B\vdash}}{\neg B\vdash\neg B}\,,\quad \forall_{\mathsf{R}}\dfrac{\forall_{\mathsf{L}}\dfrac{\overset{\Pi_B}{\vartriangle}\ B\vdash B}{\forall x.B\vdash B}}{\forall x.B\vdash\forall x.B}\,,\quad \exists_{\mathsf{L}}\dfrac{\exists_{\mathsf{R}}\dfrac{\overset{\Pi_B}{\vartriangle}\ B\vdash B}{B\vdash\exists x.B}}{\exists x.B\vdash\exists x.B}\,.$$

This example vividly illustrates the internal symmetries of system LK. These same symmetries are exploited in the cut elimination procedure, but their use is much more complicated there.

From now on, we will be working with cut free systems, so, with systems where all the rules enjoy the subformula property. Normally, proof theorists deal with the problem of showing a cut elimination theorem for a given logic, so providing the logic programming language designer with a good basis to start with.

As we will see later on, Horn clauses (as well as hereditary Harrop formulas) are usually dealt with as a fragment of intuitionistic logic, rather than classical logic. We do not discuss intuitionistic logic here. We just note that intuitionistic logic is a weaker logic than classical logic, because the famous *tertium non datur* classical logic tautology $A\vee\neg A$ does not hold, together with all its consequences, of course. The reasons for restricting ourselves to intuitionistic logic are technical. For all our purposes it is enough to say that intuitionistic logic is the logic whose theorems are proved by system LK with the restriction that at most one formula always appears at the right of the entailment symbol.

We preferred not to start from intuitionistic logic for two reasons: 1) classical logic is more familiar to most people, and 2) abstract logic programming can be defined also for logics for which the one-formula-at-the-right restriction is not enforced, like linear logic. Sticking to intuitionistic logic allows for a technical simplification in the operational reading of a deductive system, as we will see later. And, of course, intuitionistic logic, although weaker than classical logic, is still more powerful than Horn clauses and hereditary Harrop logics.

3 Abstract Logic Programming

We now informally introduce the main ideas of abstract logic programming by means of an example, and move to a more precise presentation later on.

Consider *clauses* of the following form:

$$\forall\vec{x}.((b_1\wedge\cdots\wedge b_h)\supset a),$$

where $\vec{x}$ is a sequence of variables, h can be 0, and in this case a clause is a *fact* $\forall\vec{x}.a$. A *logic programming problem* may be defined as the problem of finding a substitution σ such that

$$\mathsf{P}\vdash(a_1\wedge\cdots\wedge a_k)\sigma$$

is provable in some formal system, where $k > 0$, P is a given collection of clauses and $a_1, \ldots, a_k$ are atoms. P is called the *program*, $(a_1 \wedge \cdots \wedge a_k)$ is the *goal* and σ is the *answer substitution*.

Let us restrict ourselves to provability in LK. Suppose that P is a multiset of clauses which contains the clause

$$\forall \vec{x}.((b_1 \wedge \cdots \wedge b_h) \supset a);$$

consider the goal $(a_1 \wedge \cdots \wedge a_k)$ and suppose that σ is a unifier of a and a_l, for some l such that $1 \leqslant l \leqslant k$. Consider the following derivation Δ, where we suppose that $h > 0$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{P} \vdash (b_1 \wedge \cdots \wedge b_h)\sigma}\,\mathsf{w_R}}{\mathsf{P} \vdash (b_1 \wedge \cdots \wedge b_h)\sigma, a_l\sigma}\,\supset_{\mathsf{L}} \quad \dfrac{\dfrac{a\sigma \vdash a_l\sigma}{\vdots}\,\mathsf{w_L}}{\mathsf{P}, a\sigma \vdash a_l\sigma}\,\mathsf{w_L}}{\mathsf{P}, ((b_1 \wedge \cdots \wedge b_h) \supset a)\sigma \vdash a_l\sigma}\,\forall_{\mathsf{L}}}{\vdots}}{\mathsf{P}, \forall \vec{x}.((b_1 \wedge \cdots \wedge b_h) \supset a) \vdash a_l\sigma}\,\mathsf{c_L}}{\mathsf{P} \vdash (a_1 \wedge \cdots \wedge a_k)\sigma}\,\wedge_{\mathsf{R}}$$

where the premises are $\mathsf{P} \vdash a_1\sigma$, $\ldots$, $\mathsf{P} \vdash a_l\sigma$, $\ldots$, $\mathsf{P} \vdash a_k\sigma$.

Δ has one leaf which is an axiom $(a\sigma \vdash a_l\sigma)$; one leaf which stands for the logic programming problem $\mathsf{P} \vdash (b_1 \wedge \cdots \wedge b_h)\sigma$ and zero or more leaves $\mathsf{P} \vdash a_i\sigma$, for $1 \leqslant i \leqslant k$ and $i \neq l$. In the special case where $h = 0$ we consider the derivation Δ':

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdots}\,\mathsf{w_L}}{a\sigma \vdash a_l\sigma}}{\vdots}\,\mathsf{w_L}}{\mathsf{P}, a\sigma \vdash a_l\sigma}\,\forall_{\mathsf{L}}}{\vdots}}{\mathsf{P}, \forall \vec{x}.a \vdash a_l\sigma}\,\mathsf{c_L}}{\mathsf{P} \vdash (a_1 \wedge \cdots \wedge a_k)\sigma}\,\wedge_{\mathsf{R}}$$

where the premises are $\mathsf{P} \vdash a_1\sigma$, $\ldots$, $\mathsf{P} \vdash a_l\sigma$, $\ldots$, $\mathsf{P} \vdash a_k\sigma$.

For both cases we can consider a special inference rule b, which we call *backchain* and which stands for Δ or Δ':

$$\mathsf{b}\,\frac{\mathsf{P} \vdash a_1\sigma \quad \cdots \quad \mathsf{P} \vdash a_{l-1}\sigma \quad \mathsf{P} \vdash (b_1 \wedge \cdots \wedge b_h)\sigma \quad \mathsf{P} \vdash a_{l+1}\sigma \quad \cdots \quad \mathsf{P} \vdash a_k\sigma}{\mathsf{P} \vdash (a_1 \wedge \cdots \wedge a_k)\sigma},$$

where $\mathsf{P} \vdash (b_1 \wedge \cdots \wedge b_h)\sigma$ may be absent if $h = 0$. The rule degenerates to an axiom when $h = 0$ and $k = 1$.

A backchain rule, when read from bottom to top, reduces one logic programming problem to zero or more logic programming problems. Please notice that every application of a backchain rule is associated to a substitution (in the case above it is σ) which allows us to use an axiom leaf. We can consider each such substitution as the answer substitution produced by the particular instance of the b rule. It is not necessarily an mgu! Of course, the 'best' choice might be an mgu, which in the first-order case is unique up to variable renaming. The composition of all answer substitutions gives us the final answer substitution to a particular logic programming problem.

We can look for proofs solving a logic programming problem that are entirely built by instances of b rules. We speak of *proof search space* for a given logic programming problem, and we mean a tree whose nodes are derivations, whose root is the logic programming problem, and such that children nodes are obtained by their fathers by adding an instance of b; every time b is applied, the associated substitution is computed and applied to the whole derivation.

3.1 Example Consider the logic programming problem $\mathsf{P} \vdash p(x)$, where

$$\begin{aligned}\mathsf{P} = \{&\forall y.(q(y) \wedge r(y) \supset p(y)),\\ &q(1),\\ &r(1),\\ &r(2)\}.\end{aligned}$$

The proof search space of proofs for this problem is shown in fig. 2. Every branch corresponds to a possible solution, where the substitution applied to the conclusion is the answer substitution (it is shown explicitly, ϵ is the empty substitution). The left and centre computations are *successful*, because a proof is found at the end; the right computation instead *fails*: there is no proof for the logic programming problem $\mathsf{P} \vdash q(2)$.

Infinite computations may be obtained as well, as the following example shows.

3.2 Example Consider the logic programming problem

$$\mathsf{P} \vdash a,$$

where $\mathsf{P} = \{a \wedge b \supset a\}$. The search space of proofs for this problem is shown in fig. 3.

So far, we did not define what a backchaining rule is, we only said that the rule transforms one logic programming problem into several ones. This takes care of an obvious intuition about the recursivity of logic programming, but it's not enough. Of course, given a formula to prove and a deductive system, like LK, there are in general several ways of reducing one logic programming problem into several others. The notions we are going to introduce address specifically the problem of choosing which rules to use.

The next, crucial, intuition, is to consider a goal as a set of instructions for organising a computation. For example, the goal $A \wedge B$, to be proven by

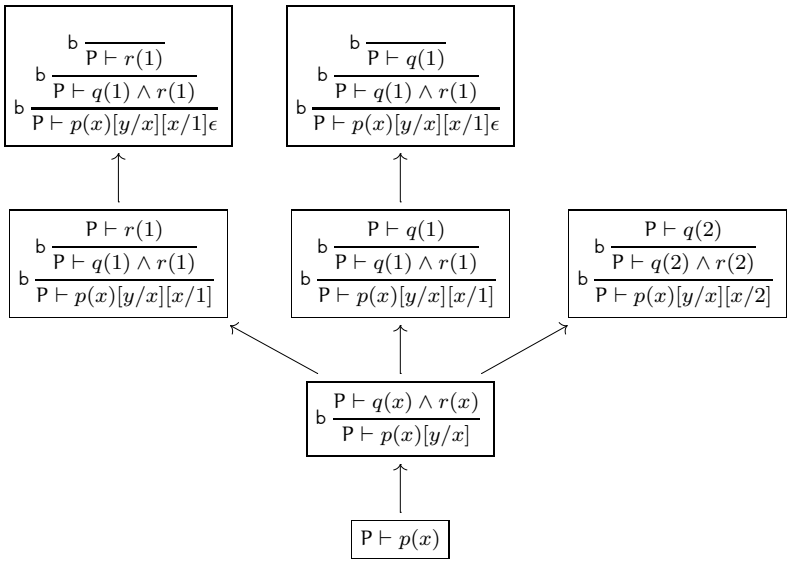

Fig. 2 *Proof search space for example 3.1*

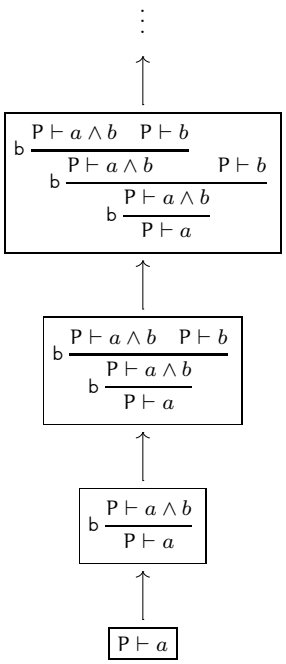

Fig. 3 *Proof search space for example 3.2*

using program P, tells the interpreter to organise the computation as the search for a proof of A and a search for a proof of B, both by using program P. This

situation is exactly what the intuitionistic version of the rule $\wedge_{\mathsf{R}}$ describes:

$$\wedge_{\mathsf{R}} \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,.$$

However, given a program P and a goal $A \wedge B$, i.e., given a sequent $\mathsf{P} \vdash A \wedge B$ to be proved, an interpreter based on LK has more options: it can use left rules by operating over the program P. Clearly, this does not correspond to our intuition as to how a logic programming interpreter should work.

In [7] the authors take a clear course of action:

1) They define what strategy for searching for proofs could correspond to a sensible, operational notion similar to the one adopted by Prolog; this leads to the idea of *uniform proof*, a uniform proof being a proof where the operational, sensible strategy is adopted.

2) They turn their attention to the question: when does uniform provability correspond to (normal, unrestricted) provability? For the languages and deductive systems where the two notions coincide, they speak of *abstract logic programming languages*.

We devote the next two subsections to uniform proofs and abstract logic programming languages.

3.1 Uniform Proofs

Let's now move to a more formal presentation of these ideas. We introduce a new sequent system, where structural rules have been eliminated, and their 'functionality' is included in the structure of sequents.

Sequent system MNPS, from the names of its authors Miller, Nadathur, Pfenning and Scedrov is built on sequents of the kind

$$\Gamma \vdash \Delta,$$

where Γ and Δ are sets of formulae (and not multisets as in system LK). $\top$ and $\bot$ are formulae different from atoms. Axioms and inference rules of MNPS are shown in fig. 4. In MNPS negation is obtained through $\neg A \equiv A \supset \bot$.

The reader should be able to relate the new system to LK, and then to any other presentation of classical logic. The exercise is not entirely trivial, and very useful for understanding the subtleties of the sequent calculus. However, it is not central to this tutorial.

3.1.1 Exercise Prove that MNPS and LK are equivalent (consider $\top \equiv A \vee \neg A$ and $\bot \equiv A \wedge \neg A$, for example).

As we said at the end of Section 2, we are mostly interested in intuitionistic logic, i.e., the logic produced by the deductive systems we have seen so far, in

Axioms

$$a, \Gamma \vdash \Delta, a \qquad \bot, \Gamma \vdash \Delta, \bot \qquad \Gamma \vdash \Delta, \top$$

Left Rules	**Right Rules**
$\supset_{\mathsf{L}} \dfrac{\Gamma \vdash \Delta, A \quad B, \Gamma \vdash \Theta}{A \supset B, \Gamma \vdash \Delta, \Theta}$	$\supset_{\mathsf{R}} \dfrac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \supset B}$
$\wedge_{\mathsf{L}} \dfrac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}$	$\wedge_{\mathsf{R}} \dfrac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$
$\vee_{\mathsf{L}} \dfrac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}$	$\vee_{\mathsf{RL}} \dfrac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} \qquad \vee_{\mathsf{RR}} \dfrac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B}$
$\forall_{\mathsf{L}} \dfrac{A[x/t], \Gamma \vdash \Delta}{\forall x.A, \Gamma \vdash \Delta}$	$\forall_{\mathsf{R}} \dfrac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall x.A}$
$\exists_{\mathsf{L}} \dfrac{A, \Gamma \vdash \Delta}{\exists x.A, \Gamma \vdash \Delta}$	$\exists_{\mathsf{R}} \dfrac{\Gamma \vdash \Delta, A[x/t]}{\Gamma \vdash \Delta, \exists x.A}$
	$\bot_{\mathsf{R}} \dfrac{\Gamma \vdash \Delta, \bot}{\Gamma \vdash \Delta, A}$ where $A \neq \bot$

where in $\forall_{\mathsf{R}}$ and $\exists_{\mathsf{L}}$ the variable x is not free in the conclusion

Fig. 4 *Sequent system* MNPS

the special case when every sequent contains at most one formula at the right of the entailment symbol $\vdash$. From the operational viewpoint, this makes immediate sense, because it corresponds to always having one goal to prove.

Actually, the whole truth is more complicated than this, because, as we said, several formulae at the right of entailment just correspond to their disjunction. In the case of a disjunction, the intuitionistic restriction forces an immediate choice about which of the disjuncts the interpreter has to prove (check the rule $\vee_{\mathsf{R}}$, and keep in mind that c_{R} is not available, neither explicitly nor implicitly). Restricting oneself to the intuitionistic case is the key to a technical simplification in the notion of uniform provability. In cases where this simplification is not desirable, like for linear logic, one can introduce a more refined notion of uniform provability [5, 6].

A proof in MNPS, where each sequent has a singleton set as its right-hand side, is called an I-*proof* (intuitionistic proof).

A *uniform proof* is an I-proof in which each occurrence of a sequent whose right-hand side contains a non-atomic formula is the lower sequent of the inference rule that introduces its top-level connective.

In other words: until a goal is not reduced to atomic form, keep reducing it; when a goal is in atomic form, then you can use left inference rules.

3.1.2 Example In MNPS, the formula $(a \vee b) \supset (((a \vee b) \supset \bot) \supset \bot)$ admits the

following uniform proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a \vdash a}{a \vdash a \vee b}\,\vee_{\mathsf{RL}} \quad a, \bot \vdash \bot}{a, (a \vee b) \supset \bot \vdash \bot}\,\supset_{\mathsf{L}} \quad \dfrac{\dfrac{b \vdash b}{b \vdash a \vee b}\,\vee_{\mathsf{RR}} \quad b, \bot \vdash \bot}{b, (a \vee b) \supset \bot \vdash \bot}\,\supset_{\mathsf{L}}}{a \vee b, (a \vee b) \supset \bot \vdash \bot}\,\vee_{\mathsf{L}}}{a \vee b \vdash ((a \vee b) \supset \bot) \supset \bot}\,\supset_{\mathsf{R}}}{\vdash (a \vee b) \supset (((a \vee b) \supset \bot) \supset \bot)}\,\supset_{\mathsf{R}}.$$

We can get back to our informal introduction of backchaining, and we can note that not only does it reduce logic programming problems to further logic programming problems, but it does so by only producing (segments of) uniform proofs.

Now that we have the notion of uniform provability, it is time to check its limits, namely, we should try to answer the question: Is uniform provability enough to prove everything we can prove in unrestricted provability? The answer is, both for classical and intuitionistic logic, no.

3.1.3 Example There is no uniform proof for $\neg\forall x.p(x) \supset \exists x.\neg p(x)$. In fact, the following derivation is compulsory, if we require uniformity:

$$\dfrac{\dfrac{\dfrac{\dfrac{p(t) \vdash \forall x.p(x) \quad \bot, p(t) \vdash \bot}{\neg\forall x.p(x), p(t) \vdash \bot}\,\supset_{\mathsf{L}}}{\neg\forall x.p(x) \vdash \neg p(t)}\,\supset_{\mathsf{R}}}{\neg\forall x.p(x) \vdash \exists x.\neg p(x)}\,\exists_{\mathsf{R}}}{\vdash \neg\forall x.p(x) \supset \exists x.\neg p(x)}\,\supset_{\mathsf{R}}.$$

There is no proof for its left premise, being t different from x, by the proviso of the $\forall_{\mathsf{R}}$ rule.

3.1.4 Example There is no uniform proof for $(a \vee b) \supset (b \vee a)$. In fact the following derivation is compulsory if we require uniformity:

$$\dfrac{\dfrac{\dfrac{a \vdash b \quad b \vdash b}{a \vee b \vdash b}\,\vee_{\mathsf{L}}}{a \vee b \vdash b \vee a}\,\vee_{\mathsf{RR}}}{\vdash (a \vee b) \supset (b \vee a)}\,\supset_{\mathsf{R}}.$$

Another derivation exists, similar to this one, where the rule $\vee_{\mathsf{RL}}$ is applied instead of $\vee_{\mathsf{RR}}$.

3.1.5 Example Let A be a generic formula; the formula $A \supset ((A \supset \bot) \supset \bot)$ doesn't always admit a uniform proof in MNPS, it may or may not have one, depending on the formula A. Example 3.1.2 shows a case where a uniform proof exists, for $A = a \vee b$. If one considers $A = \forall x.(p(x) \vee q(x))$ instead, there is no way of building a uniform proof. The following derivation does not lead to a

uniform proof, because the premise is not provable (and the same happens if we apply $\vee_{\mathsf{RR}}$ instead of $\vee_{\mathsf{RL}}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x.(p(x) \vee q(x)) \vdash p(x)}{\forall x.(p(x) \vee q(x)) \vdash p(x) \vee q(x)}\vee_{\mathsf{RL}}}{\forall x.(p(x) \vee q(x)) \vdash \forall x.(p(x) \vee q(x))}\forall_{\mathsf{R}} \quad \forall x.(p(x) \vee q(x)), \bot \vdash \bot}{\forall x.(p(x) \vee q(x)), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\supset_{\mathsf{L}}}{\forall x.(p(x) \vee q(x)) \vdash (\forall x.(p(x) \vee q(x)) \supset \bot) \supset \bot}\supset_{\mathsf{R}}}{\vdash \forall x.(p(x) \vee q(x)) \supset ((\forall x.(p(x) \vee q(x)) \supset \bot) \supset \bot)}\supset_{\mathsf{R}}.$$

Choosing to proceed on a different formula on the left of the entailment symbol, while building up the proof, doesn't improve the situation. Modulo applicability of $\vee_{\mathsf{RR}}$ instead of $\vee_{\mathsf{RL}}$, we get the following two cases:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(t) \vee q(t) \vdash p(x)}{p(t) \vee q(t) \vdash p(x) \vee q(x)}\vee_{\mathsf{RL}}}{p(t) \vee q(t) \vdash \forall x.(p(x) \vee q(x))}\forall_{\mathsf{R}} \quad p(t) \vee q(t), \bot \vdash \bot}{p(t) \vee q(t), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\supset_{\mathsf{L}}}{\forall x.(p(x) \vee q(x)), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\forall_{\mathsf{L}}}{\forall x.(p(x) \vee q(x)) \vdash (\forall x.(p(x) \vee q(x)) \supset \bot) \supset \bot}\supset_{\mathsf{R}}}{\vdash \forall x.(p(x) \vee q(x)) \supset ((\forall x.(p(x) \vee q(x)) \supset \bot) \supset \bot)}\supset_{\mathsf{R}},$$

where the premise is not provable, no matter which term t we choose (it cannot be $t = x$, of course), and

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p(t) \vdash \forall x.(p(x) \vee q(x)) \quad p(t), \bot \vdash \bot}{p(t), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\supset_{\mathsf{L}} \quad \dfrac{q(t) \vdash \forall x.(p(x) \vee q(x)) \quad q(t), \bot \vdash \bot}{q(t), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\supset_{\mathsf{L}}}{p(t) \vee q(t), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\vee_{\mathsf{L}}}{\forall x.(p(x) \vee q(x)), \forall x.(p(x) \vee q(x)) \supset \bot \vdash \bot}\forall_{\mathsf{L}}}{\forall x.(p(x) \vee q(x)) \vdash (\forall x.(p(x) \vee q(x)) \supset \bot) \supset \bot}\supset_{\mathsf{R}}}{\vdash \forall x.(p(x) \vee q(x)) \supset ((\forall x.(p(x) \vee q(x)) \supset \bot) \supset \bot)}\supset_{\mathsf{R}},$$

where again some premises are not provable. This exhausts all possible cases for that specific A.

These examples tell us that the *language* matters as much as the deductive system as far as uniform provability goes.

3.2 Abstract Logic Programming Languages

The question is now for which languages uniform provability and (unrestricted) provability coincide. Several languages of goals and clauses can be defined (not necessarily with formulae in MNPS). Also, we could consider several intuitionistic sequent systems, different from MNPS, for which uniform provability can

be contemplated. Our objective now is to define sort of a *completeness* notion which allows us to say, for a particular language, that looking for uniform proofs is equivalent to looking for proofs. Since the search for uniform proofs is computationally motivated, we have a powerful guide to assist us in the syntax-driven definition of logic programming languages.

Consider a language D of *clauses* and a language G of *goals*, both of them subsets of a language for which an entailment relation $\vdash$ is defined; let P be a finite set of clauses, called *program*; we say that $\langle \mathsf{D}, \mathsf{G}, \vdash \rangle$ is an *abstract logic programming language* if, whenever $\mathsf{P} \vdash G$ is provable, then there is a uniform proof for it, for every program P and goal G.

The property of being an abstract logic programming language is usually proved by examining the *permutability relations* between pairs of inference rules in the sequent system which defines $\vdash$. For example, consider the following fragment of derivation:

$$\supset_{\mathsf{L}} \dfrac{\Gamma \vdash A \qquad \vee_{\mathsf{RL}} \dfrac{B, \Gamma \vdash C}{B, \Gamma \vdash C \vee D}}{A \supset B, \Gamma \vdash C \vee D};$$

it clearly cannot be part of a uniform proof. The problem can be fixed if we *permute* the $\supset_{\mathsf{L}}$ and $\vee_{\mathsf{RL}}$ rules, like this:

$$\begin{array}{c} \supset_{\mathsf{L}} \dfrac{\Gamma \vdash A \quad B, \Gamma \vdash C}{A \supset B, \Gamma \vdash C} \\ \vee_{\mathsf{RL}} \dfrac{}{A \supset B, \Gamma \vdash C \vee D} \end{array}.$$

3.2.1 Example Consider goals in the set G generated by the grammar:

$$\begin{aligned} \mathsf{G} &:= \mathsf{A} \mid \mathsf{D} \supset \mathsf{A} \mid \mathsf{G} \vee \mathsf{G}, \\ \mathsf{D} &:= \mathsf{A} \mid \mathsf{G} \supset \mathsf{A} \mid \forall x.\mathsf{D}, \end{aligned}$$

where A stands for any atom and D is the set of clauses. We want to define in MNPS a backchain rule for the language considered. Remember that a backchain rule has two properties: 1) read from bottom to top, it transforms one logic programming problem into several ones (or zero); 2) it only produces uniform proofs.
The most general case is the following:

$$\mathsf{b}\,\dfrac{\mathsf{P}, D \vdash G\sigma}{\mathsf{P} \vdash G_1 \vee \cdots \vee G_h} = \begin{array}{c} \supset_{\mathsf{L}} \dfrac{\mathsf{P}, D \vdash G\sigma \quad \mathsf{P}, a'\sigma, D \vdash a}{\forall_{\mathsf{L}} \dfrac{\mathsf{P}, (G \supset a')\sigma, D \vdash a}{\vdots}} \\ \forall_{\mathsf{L}} \dfrac{}{\mathsf{P}, \forall \vec{x}.(G \supset a'), D \vdash a} \\ \supset_{\mathsf{R}} \dfrac{}{\mathsf{P} \vdash D \supset a} \\ (\vee_{\mathsf{RL}} \text{ or } \vee_{\mathsf{RR}}) \dfrac{}{\vdots} \\ (\vee_{\mathsf{RL}} \text{ or } \vee_{\mathsf{RR}}) \dfrac{}{\mathsf{P} \vdash G_1 \vee \cdots \vee G_h} \end{array},$$

where $a'\sigma = a$. Special cases are

$$\mathsf{b}\,\frac{\mathsf{P} \vdash G\sigma}{\mathsf{P} \vdash G_1 \vee \cdots \vee G_h} = \begin{array}{r} \supset_{\mathsf{L}} \dfrac{\mathsf{P} \vdash G\sigma \quad \mathsf{P}, a'\sigma \vdash a}{\mathsf{P}, (G \supset a')\sigma \vdash a} \\ \forall_{\mathsf{L}} \dfrac{}{\vdots} \\ \forall_{\mathsf{L}} \dfrac{}{\mathsf{P}, \forall\vec{x}.(G \supset a') \vdash a} \\ (\vee_{\mathsf{RL}} \text{ or } \vee_{\mathsf{RR}}) \dfrac{}{\vdots} \\ (\vee_{\mathsf{RL}} \text{ or } \vee_{\mathsf{RR}}) \dfrac{}{\mathsf{P} \vdash G_1 \vee \cdots \vee G_h} \end{array}$$

and

$$\mathsf{b}\,\frac{}{\mathsf{P} \vdash G_1 \vee \cdots \vee G_h} = \begin{array}{r} \forall_{\mathsf{L}} \dfrac{\mathsf{P}, a'\sigma, D \vdash a}{\vdots} \\ \forall_{\mathsf{L}} \dfrac{}{\mathsf{P}, \forall\vec{x}.a', D \vdash a} \\ \supset_{\mathsf{R}} \dfrac{}{\mathsf{P} \vdash D \supset a} \\ (\vee_{\mathsf{RL}} \text{ or } \vee_{\mathsf{RR}}) \dfrac{}{\vdots} \\ (\vee_{\mathsf{RL}} \text{ or } \vee_{\mathsf{RR}}) \dfrac{}{\mathsf{P} \vdash G_1 \vee \cdots \vee G_h} \end{array} ,$$

or a combination of the last two. Please observe that in the derivations above clauses $\forall\vec{x}.(G \supset a')$ at the left of the entailment can be singled out thanks to the implicit use of contraction (P is a set!).

3.2.2 Exercise Consider the following languages of goals and clauses:

$$\mathsf{G} := \mathsf{A} \mid \mathsf{D} \supset \mathsf{A} \mid \mathsf{G} \vee \mathsf{G},$$
$$\mathsf{D} := \mathsf{A} \mid \mathsf{G} \supset \mathsf{A} \mid \forall x.\mathsf{D},$$

where A are atoms. Prove that $\langle \mathsf{D}, \mathsf{G}, \vdash \rangle$ is an abstract logic programming language, where $\vdash$ is entailment as it is defined by (the intuitionistic fragment of) MNPS.

We now present two examples of well known languages, Horn clauses and hereditary Harrop formulas. It is proved in [7] that both admit uniform proofs and are, therefore, abstract logic programming languages with respect to uniform provability in MNPS.

Horn clauses are given by:

$$\mathsf{G} := \top \mid \mathsf{A} \mid \mathsf{G} \wedge \mathsf{G} \mid \mathsf{G} \vee \mathsf{G} \mid \exists x.\mathsf{G},$$
$$\mathsf{D} := \mathsf{A} \mid \mathsf{G} \supset \mathsf{A} \mid \mathsf{D} \wedge \mathsf{D} \mid \forall x.\mathsf{D}.$$

One important limitation of Horn clauses is the inability of dealing with modules. Assume that we want to evaluate a goal, say $G_1 \wedge G_2$, in a complex program composed of several parts, say D_0, D_1 and D_2. Then, an interpreter for Horn clauses, as defined before, will need to evaluate the sequent $D_0, D_1, D_2 \vdash G_1 \wedge G_2$, or, in other words it is necessary for the entire goal to have access to all pieces of the program since the very beginning.

This is often not desirable in practice, since we might know, as programmers, that specific chunks of the program will exclusively be necessary to evaluate distinct goals, for example D_1 for G_1 and D_2 for G_2.

The language of hereditary Harrop formulas extends that of Horn clauses, in particular by providing implication in goals, thus realising a notion of module. Universal quantification in goals creates private variables for these modules.

Hereditary Harrop formulas are given by:

$$\mathsf{G} := \top \mid \mathsf{A} \mid \mathsf{G} \wedge \mathsf{G} \mid \mathsf{G} \vee \mathsf{G} \mid \exists x.\mathsf{G} \mid \forall x.\mathsf{G} \mid \mathsf{D} \supset \mathsf{G},$$
$$\mathsf{D} := \mathsf{A} \mid \mathsf{G} \supset \mathsf{A} \mid \mathsf{D} \wedge \mathsf{D} \mid \forall x.\mathsf{D}.$$

In this language we can better structure the information in modules, as the following example shows.

3.2.3 Example An interpreter evaluates $D_0 \supset (\forall x.(D_1 \supset G_1) \wedge \forall y.(D_2 \supset G_2))$ by producing the derivation

$$\dfrac{\dfrac{\dfrac{\supset_{\mathsf{R}} \dfrac{D_0, D_1 \vdash G_1}{D_0 \vdash D_1 \supset G_1}}{\forall_{\mathsf{R}} \dfrac{}{D_0 \vdash \forall x.(D_1 \supset G_1)}} \quad \dfrac{\supset_{\mathsf{R}} \dfrac{D_0, D_2 \vdash G_2}{D_0 \vdash D_2 \supset G_2}}{\forall_{\mathsf{R}} \dfrac{}{D_0 \vdash \forall y.(D_2 \supset G_2)}}}{\wedge_{\mathsf{R}} \dfrac{}{D_0 \vdash \forall x.(D_1 \supset G_1) \wedge \forall y.(D_2 \supset G_2)}}}{\supset_{\mathsf{R}} \dfrac{}{\vdash D_0 \supset (\forall x.(D_1 \supset G_1) \wedge \forall y.(D_2 \supset G_2))}} \quad .$$

The programs D_1 and D_2 are now split in two different branches of the derivation, together with the goals that are supposed to access them. Moreover, the variables x and y are private to the modules D_1 and D_2 and their respective goals G_1 and G_2, thanks to the proviso of the $\forall_{\mathsf{R}}$ rule.

The theory of programming languages teaches us that these abstraction properties are very important. One can go much further than this by moving to higher orders. There we can quantify over functions and relations, thus allowing to define abstract data types in a very convenient way. It is beyond the scope of this tutorial to deal with the higher order case, it should suffice to say that the methodology shown above doesn't need any modification and carries through graciously. The reference paper is [8].

4 Conclusions

In this tutorial we have presented the proof theoretic perspective on logic programming introduced by [7]. We just stuck to the basic notions and we tried to come up with a less formal, and, hopefully, easier to understand exposition than what is already available.

Given its introductory nature, we have been forced to leave out several new exciting, recent developments, especially in connection to linear logic. Indeed, the methodology of uniform proofs and abstract logic programming lends itself naturally to applications in new logics, for new languages. Its grounds in proof theory make it especially useful for logics whose semantics are too complicated, or simply unavailable.

On the other hand, we believe that understanding this relatively new methodology in the old, familiar playground of classical logic is the best way of getting a feeling for it without having to invest too much time in its study.

References

[1] J. Gallier. Constructive logics. Part I: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110:249–339, 1993.

[2] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[3] D. Miller. Documentation for lambda prolog. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/docs.html.

[4] D. Miller. Overview of linear logic programming. Accepted as a chapter for a book on linear logic, edited by Thomas Ehrhard, Jean-Yves Girard, Paul Ruet, and Phil Scott. Cambridge University Press. URL: http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/llp.pdf.

[5] D. Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual IEEE Symp. on Logic in Computer Science*, pages 272–281, Paris, July 1994.

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

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

[8] G. Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the ACM*, 37(4):777–814, 1990.

Typeset with Xy-pic