A Proof-Theoretic Approach to Logic Programming. I. Clauses as Rules

LARS HALLNÄS* AND PETER SCHROEDER-HEISTER,† * *Programming Methodology Group, Department of Computer Science, Chalmers University of Technology and University of Göteborg,* 412 96 *Göteborg, Sweden*; † *Seminar für natürlich-sprachliche Systeme, Universität Tübingen,* 7400 *Tübingen, FRG*

Abstract

In this paper definite Horn clause programs are investigated within a proof-theoretic framework; program clauses being considered rules of a formal system. Based on this approach, the soundness and completeness of SLD-resolution is established by purely proof-theoretic methods. Extended Horn clauses are defined as rules of higher levels and related to an approach based on implication formulae in the bodies of clauses. In a further extension, which is treated in Part II of this series, program clauses are viewed as clauses in inductive definitions of atoms, justifying an additional inference schema: a reflection principle that roughly corresponds to interpreting the program clauses as introduction rules in the sense of natural deduction. The evaluation procedures for queries with respect to the defined extensions of definite Horn clause programs are shown to be sound and complete. The sequent calculus with the general elimination schema even permits the introduction of a genuine notion of falsity which is not defined via a meta-rule.

Keywords: Logic programming; proof theory; rule; inductive definition.

1. Introduction

The aim of this paper is to advance the view that definite Horn clause programs are sets of inference rules for the derivation of atoms rather than sets of logically compound formulae. This means that the theory of logic programming belongs to proof theory, in so far as proof theory not only studies complex logical and mathematical formalisms but also the general structure of proofs or derivations, independently of the internal logical structure of formulae being derived.

In the following we shall in particular deal with the proof-theoretic idea of rules of higher levels. Such rules result from rules in the ordinary sense by finite iteration of the process of rule formation. In derivations with higher level rules, one is allowed to introduce and discharge assumptions which are themselves rules. The concept of higher level rules was introduced by Schroeder-Heister [7] in the context of natural deduction systems in order to yield a uniform description of the relationship between introduction and elimination inferences for logical constants. However, the idea behind this

notion is completely independent of this application and can equally well be applied to derivations of atomic formulae. This leads to a natural extension of definite Horn clause programs, when we consider rules of higher levels to be generalized Horn clauses which may occur in a program. With respect to such extended programs, a query does not simply ask whether a certain atom (or more generally: a set of atoms) or a substitution instance thereof is derivable, but whether it is derivable *from certain assumptions,* where these assumptions may include rules.

The idea of derivations from rules as assumptions will be formally captured by a sequent calculus $C_{\Rightarrow}(P)$ which is defined in relation to a given program P. This approach is related to a concept of clauses which may contain implication formulae in their bodies. This latter concept leads to a sequent calculus $C_{\rightarrow}(P)$ which is based on the idea of introduction rules for an implication constant $\rightarrow$ both to the right and left of the sequent sign. Although differently motivated, the systems $C_{\Rightarrow}(P)$ and $C_{\rightarrow}(P)$ are equivalent in the sense that they can be embedded into each other.

A further proof-theoretic conception that we shall apply to logic programming is the view that introduction inferences determine the meaning of a constant whereas elimination inferences only reflect this meaning in some sense. We think of the rules of a logic program whose head has A as a substitution instance as determining the meaning of A. Consequently, a general inference schema corresponding to the idea of elimination inferences in natural deduction is then added to $C_{\rightarrow}(P)$, yielding an extended system $D(P)$. By means of this additional inference schema, an atom can be introduced as an assumption in a way that depends on the meaning given to it by the program rules.

With respect to the sequent calculi $C_{\Rightarrow}(P)$, $C_{\rightarrow}(P)$ and $D(P)$ there are evaluation procedures for queries which are formally handled by proofs in certain 'linear' formal systems $LC_{\Rightarrow}(P)$, $LC_{\rightarrow}(P)$ and $LD(P)$. The soundness and completeness of these formal systems will be established by proving their equivalence with $C_{\Rightarrow}(P)$, $C_{\rightarrow}(P)$, and $D(P)$, respectively. Since a very restricted version of these linear calculi (namely, without inference schemas for handling assumptions, higher level rules or implication formulae) covers the usual procedure of SLD-resolution for definite Horn clause programs, the soundness and completeness of SLD-resolution is obtained as a limiting case.

To make things easier to follow, however, we shall not start with the most general case, but apply our methods first to standard definite Horn clause programs and prove the soundness and completeness of SLD-resolution independently of the generalizations considered. This is especially justified because our way of proceeding reveals the proof-theoretic content which is only implicit in the use of the least-Herbrand-model property in standard proofs.

Our calculus $D(P)$ with the additional inference schema corresponding to elimination rules has the advantage that it contains a natural notion of falsity and thus of negation. This is due to the fact that from any zero-place predicate $\perp$, which is not head of a program rule (i.e. P does not define $\perp$), everything follows. In other words, in $D(P)$ we have a built-in '*ex falso quodlibet*', so that the notion of falsity in $D(P)$ resembles the intuitionistic concept of absurdity. To assume that $\perp$ holds according to P is absurd since what this should mean is not defined. Thus negation need not be introduced via a meta-rule (like negation as finite failure) but is so-to-speak 'intrinsic' in the system. The rule of 'negation as finite failure' can be looked upon as a derived inference schema of $D(P)$. An analogous treatment of falsity is not possible within the framework of usual definite Horn clause programs, since derivations from assumptions, and correspondingly queries referring to assumptions, must be available for that purpose.

This paper is the first in a two-part series. It discusses the basic proof-theoretic view of logic programming, according to which clauses are to be treated as inference rules. Here Sections 2 and 3 deal with ordinary logic programming based on definite Horn clauses, and Sections 4 and 5 with the extensions by rules of higher levels and by implication formulae. Part II will be concerned with the definitional view of logic programs (Sections 1 and 2), including computational aspects connected with this view (Section 3) and the notion of negation which is then available (Section 4). A general discussion of this approach and its possible applications concludes the two papers (Section 5).

2. Clauses as formulae versus clauses as rules

Logic programs are finite sets of program clauses. A standard notation for program clauses is

$$A \leftarrow B_1, \ldots, B_n,$$

where $A, B_1, \ldots, B_n$ represent atomic formulae of first-order logic (and where as a limiting case the B_i may be missing).

Such clauses can be understood in two different ways: (i) as compound logical formulae; or (ii) as rules of a formal system for the derivation of atomic formulae (axioms being considered rules without premises). The clauses-as-formulae view is commonly adopted when giving logic programs a model-theoretic semantics. In this case $A \leftarrow B_1, \ldots, B_n$ is taken as shorthand for $\forall(B_1 \wedge \ldots \wedge B_n \supset A)$ or, what in classical logic amounts to the same, for $\forall(\neg B_1 \vee \ldots \vee \neg B_n \vee A)$. Here $\forall$ denotes universal closure, i.e. universal quantification over the variables free in $B_1 \wedge \ldots \wedge B_n \supset A$. A goal, which may be posed as a query with respect to a program, is then either written as a headless clause $\leftarrow B_1, \ldots, B_n$ and understood as $\forall(\neg B_1 \vee \ldots \vee \neg B_n)$ or

written as $B_1, \ldots, B_n$ and understood as $\exists(B_1 \wedge \ldots \wedge B_n)$. In the first case one interprets it as something one wants to refute by giving a certain substitution as a counterexample to the universal closure, in the second case as something one wants to prove on the basis of the given program by giving a substitution as a witness for the existential closure. In classical logic both readings are dual to each other. We shall follow the second reading since it both seems more natural and allows for a constructive interpretation.

If we abbreviate 'for which θ' by '$(?\theta)$', $(B_1 \wedge \ldots \wedge B_n)$ by G and first-order logical consequence by '$\models$', a query may be symbolized as

$$(?\theta) \quad P \models G\theta$$

or

$$(?\theta) \quad (P \models B_1\theta \textit{ and } \ldots \textit{ and } P \models B_m\theta) \tag{1}$$

The substitution θ is then called a *correct answer substitution* of G with respect to P. Since clauses are understood to be universally quantified, free variables may occur only in $G\theta$, i.e. to the right of $\models$. Thus $P \models G\theta$ means the same as $P \models \forall(G\theta)$. It turns out that where $G\theta$ is a closed formula, we can restrict ourselves to Herbrand interpretations whose universe is based on the constants appearing in $P \cup \{G\theta\}$. This is to say that $P \models G\theta$ if and only if each Herbrand model of P is a Herbrand model of $G\theta$, i.e. $G\theta$ holds in the least Herbrand model of P.

The clauses-as-rules interpretation is favoured when one thinks of a logic program P as defining a continuous mapping on the lattice of all Herbrand interpretations of P, which has a certain Herbrand interpretation as its least fixpoint. Using another terminology this means that programs are formal systems with the ground instances of clauses as primitive rules of inference, determining a set of ground atoms (i.e. a certain Herbrand interpretation) as a set of derivable formulae. In other words, logic programs are inductive definitions of Herbrand interpretations.

Both views of logic programs are connected by the theorem that the least fixpoint according to the second interpretation (= set of derivable ground atoms = inductively defined set) is exactly the least Herbrand model of the program considered. (This is the way of matters are presented in [3], although explicit reference to rules and formal systems is lacking.) The fact that the least fixpoint interpretation is equivalent to the least-Herbrand-model interpretation shows that both approaches give an adequate semantics to definite Horn clause programs only for *ground* answer substitutions. Answer substitutions θ for a query G for which $G\theta$ is not ground are not necessarily correct even if all ground instances of $G\theta$ hold in the least Herbrand model of P.

However, a slight modification of the clauses-as-rules view leads to satisfactory results also for substitutions which are not necessarily ground. We just have to consider the clauses themselves to be rules, and not only

their ground instances. Such a rule, which we write as

$$B_1, \ldots, B_n \Rightarrow A$$

with limiting case

$$\Rightarrow A,$$

allows one to pass from (not necessarily ground) atoms $B_1\sigma, \ldots, B_n\sigma$ to the (not necessarily ground) atom $A\sigma$ for any substitution σ. This step may be written schematically as

$$\frac{B_1\sigma \ldots B_n\sigma}{A\sigma} B_1, \ldots, B_n \Rightarrow A$$

with

$$\frac{}{A\sigma} \Rightarrow A$$

as a limiting case.

According to this view, a program P defines a formal system which allows the derivation of (not necessarily closed) atomic formulae. We denote this calculus which is based on the program P by $C(P)$. A query of the form $B_1, \ldots, B_m$ then asks for the substitutions θ such that $B_i\theta$ is derivable in $C(P)$ for all i $(1 \leq i \leq n)$, or symbolically

$$(?\theta) \quad (\vdash_{C(P)} B_1\theta \textit{ and } \ldots \textit{ and } \vdash_{C(P)} B_m\theta). \tag{2}$$

In the following, P denotes programs as sets of rules, and P_f denotes the corresponding program on the clauses-as-formulae view. More precisely, P_f contains the formula $\forall(A)$ if the premise-free rule $\Rightarrow A$ is in P, and the formula $\forall(B_1 \wedge \ldots \wedge B_n \supset A)$ if the rule $B_1, \ldots, B_n \Rightarrow A$ is in P. Using this terminology, and replacing first-order consequence $\models$ by derivability in first-order logic $\vdash_L$, we can rewrite a query in the clauses-as-formulae view of the form (1) as

$$(?\theta) \quad (P_f \vdash_L B_1\theta \textit{ and } \ldots \textit{ and } P_f \vdash_L B_m\theta). \tag{3}$$

It can be shown that very little of the strength of classical first-order consequence (or derivability) is needed for defining the correctness of answer substitutions. We lose nothing if instead we base our theorizing on the clauses-as-rules approach, understanding queries in the sense of (2) rather than in the sense of (3). This is seen as follows. Suppose first-order logic L is formulated as a natural deduction system for classical logic as in [2] or [6]. Then multiple $\wedge$-introduction

$$\frac{A_1 \ldots A_n}{A_1 \wedge \ldots \wedge A_n} \quad (\wedge\wedge I)$$

and multiple $\forall$-elimination

$$\frac{\forall(B)}{B\sigma} \quad (\forall\forall E)$$

are derived inference schemata.

Now assume $\vdash_{C(P)} A$, i.e. let a derivation of A in $C(P)$ be given. We replace each step of the form

$$\frac{}{B\sigma} \Rightarrow B \tag{4}$$

by

$$\frac{\dfrac{}{\forall(B)}\,(\textit{assumption introduction})}{B\sigma}\,(\forall\forall E) \tag{5}$$

and each step of the form

$$\frac{A_1\sigma \ldots A_n\sigma}{B\sigma} A_1, \ldots, A_n \Rightarrow B \tag{6}$$

by

$$\frac{\dfrac{A_1\sigma \ldots A_n\sigma}{A_1\sigma \wedge \ldots \wedge A_n\sigma}\,(\wedge\wedge I) \quad \dfrac{\dfrac{}{\forall(A_1 \wedge \ldots \wedge A_n \supset B)}\,(\textit{assumption introduction})}{(A_1 \wedge \ldots \wedge A_n \supset B)\sigma}\,(\forall\forall E)}{B\sigma}\,(\textit{modus ponens}), \tag{7}$$

where if $n = 1$, the step $(\wedge\wedge)$ is omitted.

Since the formulae introduced in (5) and (7) belong to P_f, we have obtained a derivation in first-order logic of A from P_f which only uses assumption introduction, $(\wedge\wedge)$, $(\forall\forall)$ and modus ponens as inference steps. Conversely, given a derivation in L of A from P_f, by using proof-theoretic methods as described in [6], in particular normalization, this derivation can be transformed into one which only uses the patterns of inference (5) and (7) where the assumptions introduced belong to P_f. Therefore, by replacing these patterns by (4) and (6), respectively, we obtain a derivation of A in P.

This shows that the part of first-order logic needed within the clauses-as-formulae framework is even less than its positive fragment. And as no rule of negation is involved, the distinctions between, for example, classical, intuitionistic and minimal logic make no difference to the result. What is actually involved in the patterns (5) and (7) is just the logical reformulation of rule application as given through (4) and (6). The inferences (5) and (7) look more complicated since the instantiation of a rule, being implicit in rule application, is made explicit as a step of universal instantiation, and since combining several premises into one, which is necessary to apply modus ponens, requires the introduction of conjunction. Since only that modest fragment of first-order logic is used that exactly corresponds to rule application, it is clearer to work immediately in the framework of rules rather than using full logical consequence or derivability.

3. Linear derivations: soundness and completeness

In this section we use A, B, C for atoms and X, Y, Z for finite sets of atoms, which we also call *goals*. Every letter may be primed or indexed. We write

$X\sigma$ to denote substitution by σ in each element of X. We consider rules to be of the form $X \Rightarrow A$ (where X may be empty), i.e. the premises of rules are taken to be sets rather than sequences of atoms. By P we denote the program which is assumed to be given. The rules in P are also called *program rules*. A *variant* of a program rule is any rule which results from a program rule by relabelling its variables in such a way that different variables are transformed into different variables. Derivations in the formal system $C(P)$ based on the program P are then defined by the inference schema

$$\sigma\frac{X\sigma}{A\sigma}X \Rightarrow A \quad (\vdash P),$$

where $X \Rightarrow A$ is a program rule or a variant thereof. This schema is to be read as follows: If for each element C of X a derivation of $C\sigma$ is given, then by applying $(\vdash P)$ we obtain a derivation of $A\sigma$. (This means that for different C_1, $C_2 \in X$, the derivations of $C_1\sigma$ and $C_2\sigma$ may be different, even if $C_1\sigma$ and $C_2\sigma$ are identical.) Allowing variants of program rules relieves us from explicitly considering renaming substitutions in many cases. In an application of $(\vdash P)$, σ is called the substitution *used* at this step. Likewise, $X \Rightarrow A$ is called the rule *used* at this application. The formulation of $(\vdash P)$ with sets X instead of sequences means in particular that in a derivation in $C(P)$ the order of branches from left to right is not important. By a derivation of a set Z we mean a set containing at least one derivation of each element of Z. The empty set is thus a derivation of the empty set. We write $\vdash_{C(P)} Z$ if there is a derivation of Z in P (so $\vdash_{C(P)} \varnothing$ is always true). The *length* of a derivation is the number of inference steps in it, where, when considering a derivation of a set, we have to take the sum of the lengths of the derivations of its elements. It is clear that if we have a derivation of Z we can easily obtain a derivation of $Z\theta$ for any substitution θ. We just have to replace every step $(\vdash P)$ by

$$\sigma\theta\frac{X\sigma\theta}{B\sigma\theta} \quad X \Rightarrow B.$$

Posing a goal Z as a query can now be written as

$$(?\theta) \quad (\vdash_{C(P)} Z\theta).$$

If a derivation of Z in $C(P)$ is given, the variables occurring in rules $X \Rightarrow A$ used at steps of the form $(\vdash P)$ are also called the *rule variables* of that derivation. A derivation is called *purified* if rule variables occurring in a rule $X \Rightarrow A$ only occur at one single step of the form $(\vdash P)$ where $X \Rightarrow A$ is used, and here only in the rule $X \Rightarrow A$ and not in $X\sigma$ or $A\sigma$. It is obvious that by relabelling rule variables and changing substitutions in steps of the form

($\vdash P$), every derivation can be purified, without the derived set Z being altered.

Logic programming systems based on definite Horn clause programs are in general not theorem provers. If we pose a goal X of atoms as a query, we do not usually want to know whether for each atom A in X there is a derivation of A by means of the rules of a program P. Rather we want to know whether *there is a substitution* θ such that for each A in X, $A\theta$ is derivable, and if so, *for which* θ this holds. Thus we want to find θ *and at the same time* a derivation of $X\theta$. Obviously, this cannot be achieved by simple backward reasoning along the rules of P. Such backward reasoning would start from some A in X and try to find a derivation of A by looking for rules which, when applied in the last step, would lead to A, and so on. Logic programming systems like PROLOG start instead from X alone, and want to construct a derivation for $X\theta$ for some substitution θ which is not given at the beginning. If we consider $X\theta$ to be composed of X and θ, then we may say that the system starts from only the one component X. Thus, if we want to consider its query evaluation procedure to be a kind of (sophisticated) backward reasoning, we must develop a concept of derivation in which these two components are kept apart. In order to enable backward reasoning from X to establish the derivability of $X\theta$ without previous knowledge of θ, we need a concept of forward reasoning in which substitutions are split off from formulae. For this purpose we define a calculus $LC(P)$ for the derivation of pairs of the form $\langle X, \theta\rangle$. Derivations in $LC(P)$ are also called *linear derivations* (with respect to P) because there will be no branching. Standard SLD-resolution can then be shown to be backward reasoning along linear derivations of pairs $\langle X, \theta\rangle$, starting from a selected element of the left component X and constructing θ step by step.

We use Lloyd's [3] terminology concerning substitutions and their composition, i.e. substitutions are finite sets of bindings x/t such that x is a variable and t a term different from x. We say that σ and τ agree on X if σ and τ become identical after deleting all bindings for variables not in X. If V is a set of variables, we say that σ *acts on* V if σ contains a binding for a variable in V. The sign $\cup$ denotes the union of sets which are assumed to be disjoint.

A *linear derivation* is a finite sequence of pairs whose left component is a goal and whose right component is a substitution, which is generated by the inference schemata

$$\delta \frac{}{\langle \varnothing, \delta\rangle} (\varnothing)_L$$

$$\sigma \frac{\langle Y\sigma \cup X\sigma, \theta\rangle}{\langle Y \cup \{B\}, \sigma\theta\rangle} X \Rightarrow A \quad (\vdash P)_L.$$

In applications of $(\vdash P)_L$ it is assumed that

(i) $X \Rightarrow A$ is a variant of a program rule
(ii) $A\sigma = B\sigma$.

We say that σ and $X \Rightarrow A$ are *used* at the application of $(\vdash P)_L$ considered. The variables occurring in $X \Rightarrow A$ are called *rule variables*. A linear derivation is called *purified* if the variables of a rule used at an application of $(\vdash P)_L$ only occur in $X \Rightarrow A$ and nowhere else in the derivation. The *length* of a linear derivation is its length as a finite sequence.

The inference schema $(\vdash P)_L$ for linear derivations may be motivated as follows. Suppose σ unifies A and B, i.e. $A\sigma = B\sigma$. Then the rule $X \Rightarrow A$ allows us to pass over from $X\sigma$ to $B\sigma$ in $C(P)$, where additional formulae $Y\sigma$ that have been proved remain unaffected. This substitution σ, which is used in order to obtain some instance of B, is split off from the goal on the left side and added to the substitution on the right side of the pair. The substitution θ already present on the right side contains the substitutions used at earlier stages. Thus it is the aim of linear derivations to split off at each step of a given derivation in $C(P)$ that part of the substitution which is used to perform this step.

A linear derivation is called *strict* if all unifiers used at applications of $(\vdash P)_L$ are most general unifiers (mgu's). The substitution δ of the initial step $(\varnothing)_L$ of a linear derivation is called the *initial substitution* of this linear derivation. If a linear derivation of $\langle X, \tau \rangle$ is given, then τ can obviously be written as $\theta\delta$ where δ is the initial substitution of the linear derivation. The substitution θ is then called the *computed substitution* of the linear derivation of $\langle X, \tau \rangle$.[1]

A strict linear derivation of $\langle X, \theta\delta \rangle$ with computed substitution θ and initial substitution δ can be conceived of as representing an SLD-refutation of X with computed answer substitution θ written upside down. Whereas in linear derivations substitutions are split up into parts, in SLD-refutations unifiers are joined together in order to obtain a correct answer substitution. This becomes immediately obvious if in linear derivations we omit the right components of the pairs, since they can be constructed from the sequence of substitutions of the previous steps in the linear derivation. A linear

[1] More precisely, the computed substitution of a linear derivation can be inductively defined as follows:
$\varnothing$ *is the computed substitution of a linear derivation of the form*

$$\theta \frac{}{\langle \varnothing, \theta \rangle} (\varnothing)_L.$$

If ρ is the computed substitution of a linear derivation of

$$\langle Y\sigma \cup X\sigma, \theta \rangle,$$

then $\sigma\rho$ is the computed substitution of the linear derivation ending with the step

$$\sigma \frac{\langle Y\sigma \cup X\sigma, \theta \rangle}{\langle Y \cup \{B\}, \sigma\theta \rangle} X \Rightarrow A \qquad (\vdash P)_L.$$

derivation then takes the form

$$\begin{array}{l} \delta \dfrac{}{Z_0} \\ \sigma_1 \dfrac{}{Z_1} X_1 \Rightarrow A_1 \\ \quad \vdots \\ \sigma_n \dfrac{}{Z_n} X_n \Rightarrow A_n, \end{array}$$

where Z_0 is the empty set,

$$\delta \frac{}{Z_0}$$

represents

$$\delta \frac{}{\langle \varnothing, \delta \rangle}$$

and, for each i $(1 \leq i \leq n)$,

$$\sigma_i \frac{Z_{i-1}}{Z_i} X_i \Rightarrow A_i$$

represents

$$\sigma_i \frac{\langle Y_i\sigma_i \cup X_i\sigma_i, \sigma_{i-1}, \ldots, \sigma_1\delta \rangle}{\langle Y_i \cup \{B_i\}, \sigma_i, \ldots, \sigma_1\delta \rangle} X_i \Rightarrow A_i,$$

where $Z_i = Y_i \cup \{B_i\}$ and $A_i\sigma_i = B_i\sigma_i$.

If the linear derivation is strict, then, read from the bottom to the top, this is nothing but a successful SLD-derivation (i.e. an SLD-refutation) for Z_n with sequence of mgu's $\sigma_n, \ldots, \sigma_1$. Here the δ represents an arbitrary substitution by which the substitution $\sigma_n \ldots \sigma_1$ can be specialized, yielding a substitution $\sigma_n \ldots \sigma_1\delta$ which is obviously a correct answer substitution if $\sigma_n \ldots \sigma_1$ is one.[2] Resolution can thus be conceived of as backward reasoning with the aim of constructing a linear derivation.

Therefore the following results have immediate consequences for the characterization of SLD-resolution.

THEOREM 1.1
Suppose a linear derivation of $\langle Z, \theta\delta \rangle$ with computed substitution θ and initial substitution δ is given. Then $Z\theta$ is derivable in $C(P)$.

PROOF. By induction on the length of the linear derivation considered. If it only consists of the initial step

$$\delta \frac{}{\langle \varnothing, \delta \rangle} (\varnothing)_L$$

the assertion is trivial, since $\varnothing$ is a derivation of $\varnothing$ in $C(P)$.

[2] Unlike Lloyd [3], we do not assume that the bindings in correct answer substitutions for X are restricted to variables actually occurring in X.

Suppose the schema $(\vdash P)_L$ is used in the last step:

$$\sigma \frac{\langle Y\sigma \cup X\sigma, \theta \rangle}{\langle Y \cup \{B\}, \sigma\theta \rangle} X \Rightarrow A \quad (\vdash P)_L.$$

Let θ_1 be the computed substitution of the linear derivation of the premise of this inference. Then we know by induction hypothesis that there is a derivation in $C(P)$ of $Y\sigma\theta_1 \cup X\sigma\theta_1$. By applying $(\vdash P)$ we obtain a derivation in $C(P)$ of $Y\sigma\theta_1 \cup \{B\sigma\theta_1\}$. Obviously, $\sigma\theta_1$ is the computed substitution of the linear derivation of $\langle Y \cup \{B\}, \sigma\theta \rangle$. □

Since derivations in $C(P)$ are closed under substitutions, we have as a corollary that $Z\tau$ is derivable in $C(P)$ if $\langle Z, \tau \rangle$ is derivable in $LC(P)$. Since linear derivations may be read as successful SLD-derivations, this theorem establishes the soundness of SLD-resolution. To establish completeness, we proceed by proving three lemmas. The first two of them correspond roughly (though not exactly) to the 'lifting lemma' and the 'mgu lemma', respectively (see [3], Section 8).

LEMMA 1.1
A linear derivation of $\langle Z\tau, \theta \rangle$ with computed substitution γ can be transformed into one of $\langle Z, \tau\theta \rangle$ with computed substitution $\tau\gamma$ provided τ does not act on rule variables in the given linear derivation. If the linear derivation of $\langle Z\tau, \theta \rangle$ is purified, then so is that of $\langle Z, \tau\theta \rangle$. The length of the linear derivation remains unchanged by this transformation.

PROOF. An initial step

$$\frac{}{\langle \varnothing\tau, \gamma \rangle}$$

is transformed into an initial step

$$\frac{}{\langle \varnothing, \tau\gamma \rangle}.$$

Suppose the inference schema $(\vdash P)_L$ is used in the last step:

$$\sigma \frac{\langle Y\tau\sigma \cup X\sigma, \delta \rangle}{\langle Y\tau \cup \{B\tau\}, \sigma\delta \rangle} X \Rightarrow A \quad (\vdash P)_L,$$

where $B\tau\sigma = A\sigma$.

Since τ does not act on any variables in $X \Rightarrow A$, we have that $X\tau\sigma = X\sigma$ and $A\tau\sigma = A\sigma = B\tau\sigma$. Therefore we may replace this step by

$$\tau\sigma \frac{\langle Y\tau\sigma \cup X\sigma, \delta \rangle}{\langle Y \cup \{B\}, \tau\sigma\delta \rangle} X \Rightarrow A \quad (\vdash P)_L.$$

If γ is the computed substitution of the linear derivation of $\langle Y\tau \cup \{B\tau\}, \sigma\delta \rangle$, then $\tau\gamma$ is the computed substitution of the linear derivation of $\langle Y \cup \{B\}, \tau\sigma\delta \rangle$. Obviously, the property of being purified is not changed by the transformation performed, nor is the length of the linear derivation considered. □

LEMMA 1.2
A linear derivation of $\langle Z, \tau \rangle$ with computed substitution γ can be transformed into a strict linear derivation of $\langle Z, \tau \rangle$ with computed substitution γ' such that $\gamma = \gamma'\rho$ for some ρ. If the linear derivation of $\langle Z, \tau \rangle$ is purified, then so is the strict linear derivation of $\langle Z, \tau \rangle$. The length of the linear derivation remains unchanged by this transformation.

PROOF. By induction on the length of the given linear derivation. Applications of $(\varnothing)_L$ remain unchanged, the computed substitution being $\varnothing$.

Suppose the schema $(\vdash P)_L$ is used in the last step:

$$\sigma \frac{\langle Y\sigma \cup X\sigma, \theta \rangle}{\langle Y \cup \{B\}, \sigma\theta \rangle} X \Rightarrow A \qquad (\vdash P)_L$$

where $B\sigma = A\sigma$.

Let γ be the computed substitution of this linear derivation: it can be written as $\sigma\gamma_1$ if γ_1 is the computed substitution of the linear derivation of the premise $\langle Y\sigma \cup X\sigma, \theta \rangle$ of the inference step considered. We can compute an mgu μ of B and A in such a way that $\sigma = \mu\delta$ for some δ which does not act on rule variables of the linear derivation ending with $\langle Y\sigma \cup X\sigma, \theta \rangle$.[3] By applying Lemma 1.1 to this linear derivation, we obtain a linear derivation of the same length of $\langle Y\mu \cup X\mu, \delta\theta \rangle$ with computed substitution $\delta\gamma_1$. The induction hypothesis gives us a strict linear derivation of $\langle Y\mu \cup X\mu, \delta\theta \rangle$ with computed substitution γ_2 such that $\delta\gamma_1 = \gamma_2\rho$ for some ρ. By applying the step

$$\mu \frac{\langle Y\mu \cup X\mu, \delta\theta \rangle}{\langle Y \cup \{B\}, \mu\delta\theta \rangle} X \Rightarrow A \qquad (\vdash P)_L$$

we obtain a strict linear derivation of $\langle Y \cup \{B\}, \sigma\theta \rangle$ with computed substitution $\mu\gamma_2$. Since $\sigma\gamma_1 = \mu\delta\gamma_1 = \mu\gamma_2\rho$, we obtain $\gamma = \gamma'\rho$ by taking γ' to be $\mu\gamma_2$. Obviously, the property of being purified is not changed by this transformation, nor is the length of the linear derivation. □

LEMMA 1.3
Suppose a purified derivation of Z in $C(P)$ is given. Then there is a linear derivation of $\langle Z, \tau \rangle$ for some τ such that τ only acts on V, where V is the set of rule variables of the derivation of Z.

PROOF. By induction on the length of purified derivations in $C(P)$. If Z is empty,

$$\overline{\langle \varnothing, \varnothing \rangle}$$

is the linear derivation we are looking for.

[3] If μ_1 is any mgu of A and B such that $\sigma = \mu_1\delta_1$ for some δ_1, then $A\mu_1\rho\rho^{-1}\delta_1 = B\mu_1\rho\rho^{-1}\delta_1$ for any renaming substitution ρ of $\{A\mu_1, B\mu_1\}$. By choosing an appropriate ρ, we may take μ to be $\mu_1\rho$ and δ to be $\rho^{-1}\delta_1$.

Let Z be $Y \cup \{B\}$. Then the derivation of Z contains derivations of Y and of B. If there is more than one derivation of B, choose one of them. Suppose it proceeds by

$$\sigma \frac{X\sigma}{A\sigma} X \Rightarrow A \qquad (\vdash P)$$

in the last step, where $A\sigma = B$. Since the derivation of Z is assumed to be purified, we have that $B = B\sigma$ and $Y = Y\sigma$. In particular, σ is a unifier of A and B. By induction hypothesis we obtain a linear derivation of $\langle Y\sigma \cup X\sigma, \tau_1 \rangle$ such that τ_1 only acts on V. By applying the step

$$\sigma \frac{\langle Y\sigma \cup X\sigma, \tau_1 \rangle}{\langle Y \cup \{B\}, \sigma\tau_1 \rangle} X \Rightarrow A \qquad (\vdash P)_L$$

we obtain a linear derivation of $\langle Z, \sigma\tau_1 \rangle$. Since both σ and τ_1 only act on V, also $\sigma\tau_1$ only acts on V. It is obvious that the resulting derivation is purified. □

THEOREM 1.2
From a derivation of $Z\tau$ in $C(P)$ a purified strict linear derivation of $\langle Z, \tau' \rangle$ can be obtained, such that τ and τ' agree on Z.

PROOF. First we rename rule variables in the derivation of $Z\tau$ in $C(P)$ in such a way that the resulting derivation of $Z\tau$ is purified and all rule variables are distinct from variables on which τ acts. Lemma 1.3 then gives a purified linear derivation of $\langle Z\tau, \theta \rangle$ for θ only acting on these rule variables. By Lemma 1.1 a purified linear derivation of $\langle Z, \tau\theta \rangle$, and by Lemma 1.2 a purified strict linear derivation of $\langle Z, \tau\theta \rangle$ is obtained. The substitutions τ and $\tau\theta$ agree on Z, since θ does not act on variables in Z or $Z\tau$. □

This theorem establishes completeness of SLD-resolution. It is also provable without the 'purified' and 'strict' requirements, i.e. derivations in $C(P)$ can be transformed into linear derivations. But the result is, of course, stronger if a narrower class of linear derivations is considered.

Remarks on the concepts of soundness and completeness

In the terminology used in this section, to prove soundness and completeness means roughly to prove the equivalence of two formal systems: the calculus $C(P)$ based on the program P and the calculus $LC(P)$ for linear derivations. How does this relate to the notions of soundness and completeness as used in the standard theory of logic programming?

Since derivations in $LC(P)$ can be read as upside-down versions of SLD-refutations in the usual sense, the only problem here is our definition of correct answer substitutions with respect to the calculus $C(P)$. One might argue that the notions of soundness and completeness should relate some

syntactic concept such as derivability in $LC(P)$ to some genuinely semantic concept, and not to another syntactic concept such as derivability in $C(P)$.

This objection, however, misses the central point of our clauses-as-rules approach. Whereas formulae *can* be given a semantics in terms of models (although it seems quite questionable to us whether this would be an appropriate semantics in the context of programming), rules have by their very nature an inferential reading: the meaning of a clause considered as a rule is specified by saying how this rule is to be applied, which is exactly what the inference schema ($\vdash P$) does. Therefore the definition of the calculus $C(P)$ actually represents the semantics of clauses-as-rules. So the calculus $C(P)$ is not a purely syntactic device, but has a semantic function.

It is, of course, possible to consider the system $C(P)$ just as a technical means to prove soundness and completeness of SLD-resolution with respect to the clauses-as-formulae approach. But even then there is nothing strange about our defining correct answer substitutions with reference to the calculus $C(P)$. As shown in Section 2, an atom A is derivable in $C(P)$ iff A is derivable in first-order logic from P_f (where P_f is the program viewed as consisting of clauses-as-formulae). Since first-order logic is complete, A is derivable in $C(P)$ iff A is a logical consequence of P_f. Therefore the notion of a correct answer substitution as defined with respect to the semantic concept of logical consequence from P_f is equivalent to the one defined with respect to the syntactic concept of derivability in $C(P)$. It is actually the advantage of defining correct answer substitutions proof-theoretically that one can reduce the proofs of soundness and completeness to a comparison between calculi and therefore stay at the proof-theoretic level.

A further remark is appropriate with respect to the notion of computation involved in linear derivations. We have shown *completeness* in the following *abstract* sense: If a derivation of $X\tau$ in $C(P)$ is given, *then* we can construct a linear derivation of $\langle X, \tau' \rangle$ such that τ and τ' agree on X. In particular, when choosing unifiers in the linear derivation, we can take over substitutions from the derivation in $C(P)$ rather than compute them with some particular algorithm. The completeness we have proved (which may be called *abstract completeness*) does not imply *computational completeness* in the sense that, when faced with some goal X, any correct answer substitution τ' can eventually be computed according to some particular evaluation procedure (or at least some correct answer substitution which is more general than τ'), without previous knowledge of a derivation of $X\tau'$ in $C(P)$. So abstract completeness is completeness of a formal system (namely, $LC(P)$) rather than completeness of a computational procedure. It is well known that evaluation procedures based on the unification algorithm including the occur-check and using breadth-first search are computationally complete in this sense, but this issue involves topics such as search strategies which go beyond abstract completeness. In the terminology of complexity

theory: abstract completeness is non-deterministic whereas computational completeness is not.

It should be pointed out that abstract completeness does not hinge on the fact that a certain unification algorithm is available, but just on the fact that given a unifier σ, *there is* a most general unifier μ such that $\sigma = \mu\delta$ (compare Lemma 1.2), no matter how μ is actually computed and whether μ can be computed without giving the unifier σ as a previous piece of information. These are questions belonging to the computational aspects of completeness.

This situation is not different in the standard proofs of completeness of SLD-resolution such as those of Lloyd [3], which are seemingly more semantic. In considering the least-Herbrand-model as generated inductively by means of program clauses (compare the proof of Theorem 8.3 in [3]), one considers a kind of derivations of the elements of the least-Herbrand-model which corresponds to our derivations in $C(P)$. Furthermore, the completeness theorems do not rely on the particular computation procedures for most general unifiers, but just on the existence of mgu's if unifiers are given (compare the mgu Lemma 8.1 in [3]). So what one usually calls completeness is abstract completeness in our sense.

4. Logic programming with higher level rules

If we have a formal system K, we can ask not only whether some formula A is derivable in K ($\vdash_K A$), but also whether there is a derivation of A in K from a set of assumption formulae X ($X \vdash_K A$). In such a derivation we may introduce assumptions according to the schema

$$\overline{A}$$

for any formula A, the resulting derivation then being dependent on all formulae introduced in that way. Since logic programs can be viewed as formal systems, the concept of a derivation from assumptions is also defined for such programs. In particular, given a program P, queries of the form

$$(?\theta) \quad X\theta \vdash_{C(P)} A\theta \tag{8}$$

have a well-defined sense: they ask for substitutions θ such that $A\theta$ is derivable from the set of assumptions $X\theta$ according to the rules of the program P. Such a query is different from

$$(?\theta) \quad \vdash_{C(P \cup X)} A\theta \tag{9}$$

which asks for substitutions θ such that $A\theta$ is derivable according to the rules of the program P enlarged by the elements of X as axioms. A query of the form (9) can easily be handled in the standard systems based on definite Horn clause programs: we just have to add X as a set of clauses without bodies to the given program P and consider a query to be posed with respect

to this extension of P. The difference between (8) and (9) is that in program rules, but not in assumptions, variables are understood as expressing generality, i.e. they may be substituted arbitrarily. Therefore we have that for any θ, $X \vdash_{C(P)} A$ implies $X\theta \vdash_{C(P)} A\theta$ but not necessarily $X \vdash_{C(P)} A\theta$, whereas $\vdash_{C(P \cup X)} A$ always implies $\vdash_{C(P \cup X)} A\theta$. Thus the extension of logic programming systems by queries of the form (8) is an extension which goes beyond mere database handling.

By permitting the introduction of assumptions, we have only widened the notion of a *derivation* and correspondingly the notion of a query as what may be asked about a derivation. The notion of a *rule* is extended when we allow for the discharging of assumptions, as in natural deduction. Here this will be carried out immediately for rules of all finite levels, where assumptions may themselves be rules. We define rules of any level $n \geq 0$ and correspondingly programs with rules of higher levels as follows:

Each atom is a rule of level 0. If X is a non-empty finite set of rules of maximum level n, then $X \Rightarrow A$ is a rule of level $n+1$. We also consider $\varnothing \Rightarrow A$, which is identified with A (and thus is a rule of level 0). A program P is a finite set of rules.

From now on we use A, B, C as syntactic variables for atoms, R for rules, and X, Y, Z for finite sets of rules (all letters with or without subscripts). If we write rules explicitly, we sometimes omit set brackets, i.e. when we write

$$R_1, \ldots, R_n \Rightarrow A$$

this is to be understood as

$$\{R_1, \ldots, R_n\} \Rightarrow A.$$

As before, we suppose a fixed program P to be given, whose rules are referred to as program rules.

A rule of level 0 asserts an atom. A rule of level 1 is a rule in the familiar sense, allowing one to pass from formulae to formulae. A rule of level 2 is a rule which permits discharge of assumption formulae. Its general form is

$$((B_{11}, \ldots, B_{1m_1}) \Rightarrow B_1), \ldots, ((B_{n1}, \ldots, B_{nm_n}) \Rightarrow B_n) \Rightarrow A$$

(where some, but not all, m_i may be 0). Its intended meaning is that one may pass from $B_1, \ldots, B_n$ to A, and for each B_i discharge dependencies on $B_{n1}, \ldots, B_{nm_i}$. For example, the rule of $\supset$-introduction in natural deduction can be written in the following way as a rule of level 2:

$$(t(x) \Rightarrow t(y)) \Rightarrow t(x \supset y)$$

(with t a unary predicate symbol, $\supset$ a two-place function symbol and x, y variables), and similarly the rule of $\vee$-elimination:

$$t(x \vee y), (t(x) \Rightarrow t(z)), (t(y)) \Rightarrow t(z)) \Rightarrow t(z).$$

A rule of level 3 is a rule by whose application assumption rules of level 1

(and perhaps also atoms) can be discharged, and so on. An example of a rule of level 3 is the generalized $\supset$-elimination rule of the form

$$t(x \supset y),\ ((t(x) \Rightarrow t(y)) \Rightarrow t(z)) \Rightarrow t(z),$$

by means of which one may pass over to any atom $t(c)$, provided one has derived both $t(a \supset b)$ and $t(c)$, where in the derivation of $t(c)$ one may have used the level 1 rule $t(a) \Rightarrow t(b)$ as an assumption. (This rule is equivalent to the level 1 rule of modus ponens

$$t(x \supset y),\ t(x) \Rightarrow t(y),$$

but follows a uniform pattern of elimination rules.)

In general, the intended meaning of a higher level rule

$$(X_1 \Rightarrow B_1), \ldots, (X_n \Rightarrow B_n) \Rightarrow A$$

can be stated as follows: if, for each i $(1 \leq i \leq n)$, B_i has been derived from X_i, then A may be asserted and X_i may be discharged. This is to be understood with respect to any substitution instance of the rule. Rules of level 0 or 1 are definite Horn clauses in the usual sense, whereas rules of higher levels iterate the formation of definite Horn clauses to the left of the rule arrow $\Rightarrow$. The precise meaning of applying, assuming and discharging a rule is given by the definition of how to derive an atom from a set of assumptions X (which may contain rules) with respect to a program P. This could be done in a natural deduction style framework in which discharging of assumptions is made explicit, e.g. by bracketing the assumptions concerned. However, for the purpose of this paper it is more useful to define the derivability of A from X with respect to P directly by means of a calculus of sequents, where a sequent is understood as an expression of the form $X \vdash A$. This sequent calculus is called $C_{\Rightarrow}(P)$. It is the advantage of a sequent calculus that the assumptions on which a formula depends are written down explicitly with every inference step. For a detailed treatment of rules of higher levels in a natural deduction style framework see [7], and in a sequent-style formulation see [8].

The turnstile $\vdash$ is now a symbol of the object language $C_{\Rightarrow}(P)$. We shall use the notation $X \vdash_{C_{\Rightarrow}(P)} A$ to express that the sequent $X \vdash A$ is derivable in $C_{\Rightarrow}(P)$. Furthermore, in the context of informal motivations we continue to speak of the derivability of A from X with respect to P, since this is the notion which the derivability of the sequent $X \vdash A$ in $C_{\Rightarrow}(P)$ is intended to capture. We use the following abbreviations: $X, Y \vdash A$ stands for $X \cup Y \vdash A$; $X, A \vdash B$ for $X \cup \{A\} \vdash B$; $X \vdash Y \Rightarrow A$ stands for $X, Y \vdash A$, and $X \vdash Y$ for the set $\{X \vdash R \mid R \in Y\}$. In particular, $X \vdash \varnothing$ is the empty set of sequents. So if a rule or a set of rules of level >0 appears to the right of the turnstile, this is always an abbreviation. Rules as well as sequents have only atoms on the right.

The system $C_{\Rightarrow}(P)$ is then defined by the following inference schemata:

$$\frac{X \vdash Y}{X, (Y \Rightarrow A) \vdash A} \quad (\Rightarrow)$$

$$\sigma\frac{X \vdash Y\sigma}{X \vdash A\sigma} Y \Rightarrow A \quad (\vdash P),$$

where in applications of $(\vdash P)$, $Y \Rightarrow A$ is a program rule or a variant thereof. The schema $(\Rightarrow)$ says how to introduce a rule $Y \Rightarrow A$ as an assumption, namely, by using it to infer A from Y, whereas according to $(\vdash P)$, $Y \Rightarrow A$ is used as a program rule and thus does not appear to the left of the turnstile. As a limiting case, $(\Rightarrow)$ captures the schema

$$\frac{}{X, A \vdash A}$$

by just letting $Y \Rightarrow A$ be $\varnothing \Rightarrow A$. Since variables in program rules are understood universally, substitutions are involved in $(\vdash P)$, but not in $(\Rightarrow)$.

In the following, finite sets of sequents will be denoted by Γ and Σ. Similar to the conventions of Section 3, a derivation of a set Γ in $C_{\Rightarrow}(P)$ is defined to be a set containing a derivation in $C_{\Rightarrow}(P)$ of each element of Γ. We write $\vdash_{C_{\Rightarrow}(P)}\Gamma$ to indicate that there is a derivation of Γ in $C_{\Rightarrow}(P)$. Substitution $\Gamma\theta$ is defined elementwise. The length of a derivation is the number of inference steps used.

A query with respect to P poses a question of the form

$$(?\theta) \quad (X_1\theta \vdash_{C_{\Rightarrow}(P)} A_1\theta \textit{ and } \ldots \textit{ and } X_n\theta \vdash_{C_{\Rightarrow}(P)} A_n\theta),$$

which, using the terminology introduced, can be written as

$$(?\theta) \quad \vdash_{C_{\Rightarrow}(P)}\Gamma\theta$$

for some Γ. Thus a query can be represented by a finite set Γ of sequents (where, as a trivial case, the empty set is allowed). This generalizes the notion of a query in the standard theory of logic programming, where it is a finite set of atoms.

Evaluation of queries is defined by linear derivations of pairs $\langle \Sigma, \sigma \rangle$ in a system $LC_{\Rightarrow}(P)$, which are based on the following inference schemata:

$$\frac{}{\langle \varnothing, \delta \rangle}(\varnothing)_L$$

$$\sigma\frac{\langle \Sigma\sigma \cup \{X\sigma \vdash Y\sigma\}, \theta \rangle}{\langle \Sigma \cup \{X, (Y \Rightarrow A) \vdash B\}, \sigma\theta \rangle} \quad (\Rightarrow)_L,$$

where σ is a unifier of A and B; and

$$\sigma \frac{\langle \Sigma\sigma \cup \{X\sigma \vdash Y\sigma,\ \theta\}\rangle}{\langle \Sigma \cup \{X \vdash B\},\ \sigma\theta\rangle}\ Y \Rightarrow A \quad (\vdash P)_L,$$

where $Y \Rightarrow A$ is a program rule or a variant thereof and σ is a unifier of A and B.

As in Section 3, we can prove soundness and completeness of $LC_{\Rightarrow}(P)$ with respect to $C_{\Rightarrow}(P)$, in particular: if $\vdash_{LC_{\Rightarrow}(P)}\langle \Sigma, \tau\rangle$, then $\vdash_{C_{\Rightarrow}(P)}\Sigma\tau$ and if $\vdash_{C_{\Rightarrow}(P)}\Sigma\tau$, then $\vdash_{LC_{\Rightarrow}(P)}\langle \Sigma, \tau'\rangle$ for some τ' such that τ and τ' agree on Σ. The proofs of these assertions proceed exactly as those of Theorems 1.1 and 1.2. Only $(\Rightarrow)$ and $(\Rightarrow)_L$ have to be considered in addition. Since these inference schemata are construed in parallel to $(\vdash P)$ and $(\vdash P)_L$, we do not give detailed proofs here.

5. Rules and implications

Apart from the fact that program rules are primitive rules of inference whereas assumption rules are explicitly assumed and thus appear to the left of the turnstile, program rules and assumption rules are treated alike in $C_{\Rightarrow}(P)$. They are both considered as rules in the sense that they permit the derivation of atoms, provided other atoms have been derived (perhaps from assumptions). In particular, the rule arrow $\Rightarrow$ is only iterated to the left and not to the right.

Another possibility of extending definite Horn clause programs is to consider (iterated) implications in the bodies of program clauses. Syntactically, iterated implications differ from higher level rules in that iteration is now allowed both to the right and to the left, and in that there is no conjunction-like association corresponding to the comma (however, the latter would be easy to introduce). Semantically, the difference between higher level rules and iterated implications is that rules can be assumed by applying them [reflected in the inference schema $(\Rightarrow)$ of $C_{\Rightarrow}(P)$] but cannot be asserted, whereas iterated implications are formulae that can be both assumed and asserted. Correspondingly, in the following sequent system we will postulate introduction rules for iterated implications on the left of the turnstile, governing assumption, as well as on the right of the turnstile, governing assertion. To make the conceptual difference between implications and rules clear, we use the single arrow $\rightarrow$ for implication.

Let an *implication formula* be defined as follows: every atom is an implication formula. If F and G are implication formulae, then so is $(F \rightarrow G)$. Let in the following F, G, H (with and without subscripts) denote implication formulae, and let X, Y, Z denote sets of implication formulae. Let a program rule have the form $X \Rightarrow A$. Let P be a fixed program. Let a

sequent be of the form $X \vdash F$, and let $X \vdash Y$ denote $\{X \vdash F \mid F \in Y\}$. Then $C_{\rightarrow}(P)$ is defined by the following inference schemata:

$$\frac{}{X, A \vdash A}(I)$$

$$\frac{X, F \vdash G}{X \vdash F \rightarrow G}(\vdash\rightarrow)$$

$$\frac{X \vdash F \quad X, G \vdash H}{X, (F \rightarrow G) \vdash H}(\rightarrow\vdash)$$

$$\sigma\frac{X \vdash Y\sigma}{X \vdash A\sigma} Y \Rightarrow A \quad (\vdash P),$$

where $Y \Rightarrow A$ is a program rule or a variant thereof.

In $C_{\rightarrow}(P)$ we have not given up the clauses-as-rules view of logic programs which is the basis of our proof-theoretic approach. The difference between $C_{\Rightarrow}(P)$ and $C_{\rightarrow}(P)$ only concerns the bodies of program clauses. In both cases clauses have the form $Y \Rightarrow A$ and are therefore rules. This is expressed in the inference schema $(\vdash P)$ which is the same in both systems. The difference is that, in $C_{\Rightarrow}(P)$, Y may contain (higher level) rules, governed by a single inference schema $(\Rightarrow)$, whereas in $C_{\rightarrow}(P)$, Y may contain (iterated) implications governed by the two schemata $(\vdash\rightarrow)$ and $(\rightarrow\vdash)$.

Formally, a clauses-as-formulae view may of course be given for $C_{\rightarrow}(P)$ simply by interpreting both $\rightarrow$ and $\Rightarrow$ as intuitionistic implication. However, this does not capture our intentions. Not even the implication $\rightarrow$, which may appear in bodies of clauses, must be read as intuitionistic implication, although this reading might be suggested by our choice of the inference schemata $(\vdash\rightarrow)$ and $(\rightarrow\vdash)$ which are exactly those of the intuitionistic sequent calculus. We consider $\rightarrow$ to be a kind of implication in terms of which 'propositional' implications like intuitionistic implication can be defined. A program for defining intuitionistic implication $\supset$ as a two-place function constant occurring in the scope of a truth predicate t may, for example, be given by

$$(t(x) \rightarrow t(y)) \Rightarrow t(x \supset y)$$
$$t(x \supset y), t(x) \Rightarrow t(y)$$

(cf. the similar program in Section 4 with higher level rules). Thus one may perhaps look at $\rightarrow$ as a kind of 'meta-level' implication which may be used to define object-level implications within a truth-definition. (Note that 'meta-level' here still means 'on the basis of a program' and not 'in the meta-language'.)

Although the systems $C_{\Rightarrow}(P)$ and $C_{\rightarrow}(P)$ are motivated in different ways, they are equivalent in the sense that they can be embedded into each other.

Define the following translation of rules into implication formulae:

$$A^{\circ} = A$$

$$(R_1, \ldots, R_n \Rightarrow A)^{\circ} = R_1^{\circ} \rightarrow (\ldots (R_n^{\circ} \rightarrow A) \ldots).$$

If P is a program in the sense of $C_{\Rightarrow}(P)$, let P° be the result of replacing every program rule $R_1, \ldots, R_n \Rightarrow A$ by $R_1^{\circ}, \ldots, R_n^{\circ} \Rightarrow A$. Conversely, define the following translation of implicational formulae into rules:

$$A^{+} = A$$

$$(F_1 \rightarrow (F_2 \rightarrow \ldots \rightarrow (F_n \rightarrow A) \ldots))^{+} = (F_1^{+}, \ldots, F_n^{+}) \Rightarrow A.$$

If P is a program in the sense of $C_{\rightarrow}(P)$, let P^{+} be the result of replacing every program rule $G_1, \ldots, G_n \Rightarrow A$ by $G_1^{+}, \ldots, G_n^{+} \Rightarrow A$. Then it can be easily shown that

$$R_1, \ldots, R_n \vdash_{C_{\Rightarrow}(P)} A \textit{ iff } R_1^{\circ}, \ldots, R_n^{\circ} \vdash_{C_{\rightarrow}(P^{\circ})} A$$

and

$$F_1, \ldots, F_n \vdash_{C_{\rightarrow}(P)} G \textit{ iff } F_1^{+}, \ldots, F_n^{+} \vdash_{C_{\Rightarrow}(P^{+})} G^{+}.$$

Furthermore, it is easy to see that for $C_{\rightarrow}(P)$ the following proposition holds:

$$\text{if } X \vdash_{C_{\rightarrow}(P)} F \text{ and } Y, F \vdash_{C_{\rightarrow}(P)} G, \text{ then } X, Y \vdash_{C_{\rightarrow}(P)} G. \tag{10}$$

For $C_{\rightarrow}(P)$ again an evaluation procedure can be defined by means of a linear calculus $LC_{\rightarrow}(P)$, and soundness and completeness can be proved. This system $LC_{\rightarrow}(P)$ is a subsystem of the calculus $LD(P)$ considered in Part II. If one takes away from the proof given there everything that has to do with the rules $(P\vdash)$ and $(P\vdash)_L$, all results can be read as results for $C_{\rightarrow}(P)$ and $LC_{\rightarrow}(P)$.

Assumptions versus databases in sequent-style systems

A query is always a query with respect to a given program P. This program may be considered as a database, forming the general background of reasoning and expressing certain assumptions about the *world* the programmer is dealing with. In the case of generalized Horn clause programs, a query is a set of sequents, whose antecedents can be interpreted as assumptions. When we ask whether [a substitution instance of]

$$X \vdash A$$

is derivable in $C_{\Rightarrow}(P)$ or $C_{\rightarrow}(P)$, this can be read as expressing that [a substitution instance of] A is derivable *from* the [corresponding substitution instances of] assumptions X *with respect to* P. The set of assumptions X here

is not to be considered as an addition to the database P. We do not, when assuming X, change the program P by extending it with X. Thus 'assuming' a program is to be distinguished from assuming certain rules as antecedents of sequents with respect to the program. One may perhaps say that the program represents *global* assumptions whereas the assumptions in antecedents of sequents are *local*. It is the conceptual advantage of sequent-style systems that they make this distinction explicit. The local assumptions are always carried with the sequents as their antecedents whereas the global assumptions remain in the background as the database.

As mentioned above, mixing up databases with local assumptions may even lead to technical mistakes: if Y contains variables it does not only make a conceptual difference whether a sequent $X \cup Y \vdash A$ is derived with respect to a program (database) P, or whether $X \vdash A$ is derived with respect to the extended database $P \cup Y$, since within the database the variables in Y are understood universally whereas within the antecedent of a sequent they are not.

Miller [4], Miller *et al.* [5], Gabbay and Reyle [1] and others have discussed logical extensions of definite Horn clause programs by allowing implications to occur in the bodies of clauses. Apart from the fact that they work in the clauses-as-formulae framework, their treatment of queries with assumptions is different from ours, particularly with respect to the distinction between local assumptions and databases. Instead of working with a calculus of sequents, where this distinction is obvious and natural, Miller and Gabbay and Reyle work with ordinary derivability of formulae from a database (program) P. A hypothetical query which in our framework would be written as the sequent $X \vdash A$, is understood as a query of A with respect to the extended program $P \cup X$.

Acknowledgements

The authors started their collaboration when Peter Schroeder-Heister visited the University of Stockholm in the autumn of 1984, supported by a travel grant from the Deutsche Forschungsgemeinschaft. The present paper was written during a stay of Lars Hallnäs at the University of Konstanz in January 1987, which was financed by the Stifterverband für die Deutsche Wissenschaft. The final version was completed when P. S.-H. visited the University of Stockholm in the autumn of 1987, supported by Svenska Institutet. The results were presented in part at the 4th Japanese–Swedish Workshop on Fifth Generation Computer Systems, Skokloster (Sweden), July 1986, at the Computer Laboratory, University of Cambridge (England), at the Workshop on General Logic at the Laboratory for Foundations of Computer Science, University of Edinburgh (Scotland), February 1987, and at the Logic Colloquium, Granada (Spain), July 1987. We would like to thank Dale Miller, Michael Morreau, Tobias Nipkow and the reviewers for helpful comments.

References

[1] D. M. Gabbay and U. Reyle (1984). N-PROLOG: an extension of PROLOG with hypothetical implications: I., *Journal of Logic Programming*, **1**, 319–55.
[2] G. Gentzen (1985). Untersuchungen über das logische Schließen, *Mathematische Zeitschrift*, **39**, 176–210, 405–31.
[3] J. W. Lloyd (1984). *Foundations of Logic Programming*. Springer-Verlag, Berlin.
[4] D. Miller (1986). A theory of modules for logic programming. In *Proceedings of the 1986 Symposium on Logic Programming* (*Salt Lake City Utah*). IEEE Computer Society Press, Washington.
[5] D. Miller, G. Nadathur and A. Scedrov (1987). Hereditary marrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Washington.
[6] D. Prawitz (1965). *Natural Deduction: A Proof-Theoretical Study*. Almkvist and Wiksell, Stockholm.
[7] P. Schroeder-Heister (1984). A natural extension of natural deduction, *Journal of Symbolic Logic*, **49**, 1284–1300.
[8] P. Schroeder-Heister (1987). *Structural Frameworks with Higher-Level Rules: Proof-Theoretic Investigations*. Habilitationsschrift, Universität Konstanz.

Received 19 February 1990