A Proof-Theoretic Approach to Logic Programming.

II. Programs as Definitions

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*

† *Fakultät für Informatik, Universität Tübingen,* 7400 *Tübingen, Germany*

Abstract

We introduce a definitional extension of logic programming by means of an inference schema ($P\vdash$), which, in a certain sense, is dual to the ($\vdash P$) schema of rule application discussed in Part I. In the operational semantics, this schema is dual to the resolution principle. We prove soundness and completeness for the extended system, discuss the computation of substitutions that this new schema gives rise to, and also consider the notion of negation intrinsic to the system and its relation to negation by failure.

1. The definitional reading of logic programs

When one writes a logic program, one intends to define somehow the predicates involved, i.e., to give them a meaning via those rules whose heads start with the predicates in question. This is reflected in the terminology used in several PROLOG textbooks and manuals, where the set of rules for a predicate p is called the 'definition' of p. The proof-theoretic approach can be used to obtain an extension of definite Horn clause programming, which makes this definitional reading of logic programs precise. Proof-theoretically, we can look at a program rule $X \Rightarrow A$ as an *introduction rule* for A or as a clause in an inductive definition. The corresponding analogy is that in ordinary natural deduction, the introduction rules for a logical constant α can be viewed as rules giving meaning to α, whereas the elimination rule for α is uniquely determined by the introduction rules for α in the following sense: the elimination rule for α states that everything that can be derived from the premises of each introduction rule for α can be derived from the conclusion of these introduction rules (which is the same for all introduction rules). For example, in the case of a disjunction $E_1 \vee E_2$,

This paper continues 'A Proof-Theoretic Approach to Logic Programming. I. Clauses as Rules', which appeared in Vol. 1 No. 2 of this journal. In the following this first part is referred to as 'Part I'.

J. Logic Computat., Vol. 1 No. 5, pp. 635–660, 1991

the elimination rule for $\vee$ states that a formula E follows from $E_1 \vee E_2$ if E follows both from E_1 and from E_2 (which are the premises of the two introduction rules for $\vee$ with conclusion $E_1 \vee E_2$). The motivation behind this principle is that to assume a formula E with main operator α means to assume that E has been derived according to its definition, i.e. according to at least one of the introduction rules for α (if there is one). Therefore, what follows from the premises of *each* introduction rule for α also follows from E itself. If there is no introduction rule for α, we have as a limiting case that from E everything follows.[1]

To make this idea more precise in the case of logic programming, where we deal with predicates and atoms rather than logical constants and logically compound formulae, and where different introduction rules for the same predicate may have different conclusions, we introduce the following notation. Let

$$\mathbf{D}(A) =_{def} \{Y\sigma \mid Y \Rightarrow B \text{ is a program rule such that } A = B\sigma\}$$

This means, $\mathbf{D}(A)$ is a set of sets, containing as its elements all sets of premises from which A can be inferred according to the given program rules. The set $(\mathbf{D}(A))\theta$ for a substitution θ is defined elementwise, i.e. it is $\{Y_1\theta, \ldots, Y_n\theta\}$ if $\mathbf{D}(A)$ is $\{Y_1, \ldots, Y_n\}$. To obtain a convenient notation, we use $X, \mathbf{D}(A) \vdash F$ as an abbreviation for the set of all sequents $X, Y \vdash F$ such that Y is in $\mathbf{D}(A)$. Relying on the above motivation we may then require the following: if some F is derivable from each set in $\mathbf{D}(A)$ (perhaps with additional assumptions X), it is also derivable from A itself.

We define $D(P)$ to be the sequent calculus which results from $C_{\rightarrow}(P)$ by the addition of the inference schema:

$$\frac{X, \mathbf{D}(A) \vdash F}{X, A \vdash F}\,(P\vdash)$$

$$\text{provided for all } \sigma: \mathbf{D}(A\sigma) = (\mathbf{D}(A))\sigma$$

The proviso is to ensure that $(P\vdash)$ is closed under substitution, i.e. that derivability of $X \vdash F$ in $D(P)$ implies that of $X\sigma \vdash F\sigma$ for any substitution σ (for $C_{\rightarrow}(P)$ this is obvious). At the same time, the proviso guarantees that $\mathbf{D}(A)$ is finite: $\mathbf{D}(A)$ can be infinite only if some program rule $Y \Rightarrow B$, for which $A = B\sigma$ for some σ, contains variables in the premises Y which do not occur in B. But then the proviso would not be fulfilled.

[1] This relationship between introduction and elimination rules for logical constants was first pointed out by Gentzen [7] and has been elaborated in the meaning-theoretical reinterpretation of general proof theory by Prawitz [16] and Martin-Löf [14]. An analogous inversion principle for atomic systems was first formulated by Lorenzen [13]. For detailed formulations of the uniform relationship between introduction and elimination rules for logical constants which is exploited e.g. in the logical framework [24] see [17, 18, 19, 21, 22]. A discussion of the logical and philosophical background of our application of these ideas to logic programming can be found in [20].

To demonstrate how powerful the inference schema $(P\vdash)$ is, consider as an example the program with the single rule

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

Then a sequent expressing modus ponens can be derived by using the inference schema $(P\vdash)$, which was not possible with the example program given in Section 5 of Part I where a rule for modus ponens had to be stated explicitly:

$$\dfrac{\dfrac{\dfrac{}{t(x)\vdash t(x)}(I) \qquad \dfrac{}{t(x),t(y)\vdash t(y)}(I)}{t(x),(t(x)\rightarrow t(y))\vdash t(y)}(\rightarrow\vdash)}{t(x),t(x\supset y)\vdash t(y)}(P\vdash)$$

Another example is the derivation of $\vdash \bot\rightarrow t(x)$ in the same program, where $\bot$ is a zero-place predicate. Since $\bot$ does not occur in the program, $\mathbf{D}(\bot)$ is empty:

$$\dfrac{\dfrac{}{\bot\vdash t(x)}(P\vdash)}{\vdash \bot\rightarrow t(x)}(\bot\rightarrow)$$

For some more general comments on the definition of object logics in our system, cf. [20].

To see the necessity of the proviso, consider a program containing the rule $p(x)\Rightarrow q$. Here, x is an extra variable in the premise of the rule which does not occur in its conclusion. Then $p(x)$ is in $\mathbf{D}(q)$, hence in $\mathbf{D}(q\{x/y\})$, but not in $(\mathbf{D}(q))\{x/y\}$. This means that according to the proviso, the schema $(P\vdash)$ cannot be applied with conclusion $X, q\vdash F$. However, it must be emphasized that this restriction for the applicability of $(P\vdash)$ is not due to the proviso's being too strong—it is a natural proviso for the interpretation of P as a definition (see below). Rather, it has to do with the fact that in program rules with extra variables in the premises these variables are *understood* existentially but are not quantified *explicitly*. In other words, it is intrinsic to the usual way variables are handled in program rules. If we could write a program rule such as $p(x)\Rightarrow q$ in the form $\Sigma_x p(x)\Rightarrow q$ where Σ_x expresses a kind of existential quantification in the premise of the rule, the problem mentioned would not arise. Another possibility would be to require in the definition of $\mathbf{D}(A)$ that extra variables in Y be replaced by new constants. We do not deal with this family of problems here since it is not specific for the approach we propose. There are techniques in automated theorem proving to handle quantification that would be appropriate for our context (see, e.g., [11]).

From the standpoint of the theory of definitions, as treated in logic textbooks (see, e.g., Suppes [23], ch. 8), the proviso is perfectly well

motivated. One does not usually allow for extra variables in the definiens which do not occur in the definiendum. In this sense we may interpret the proviso as telling that A has a *definite* meaning according to the definition (program) P. However, the proviso is not equivalent to the no-extra-variable condition. If one has $p \Rightarrow q(x)$ and $r \Rightarrow q(t)$ as program rules, then there are no variables at all in the premises, but $\mathbf{D}(q(x))$ is $\{p\}$, whereas $\mathbf{D}(q(x)\{x/t\})$ is $\{p, r\}$. So the proviso also expresses that A is instantiated far enough to have a definite meaning with respect to P.

Note that the proviso is a decidable property of A: The proviso holds iff $\mathbf{D}(A)$ contains no variables beyond those occurring in A and furthermore for all program rules $Y \Rightarrow B$ such that $A\theta = B\theta$, we have that $Y\theta = Z\theta$ for some Z in $\mathbf{D}(A)$, which are decidable properties.

In addition to $C_{\rightarrow}(P)$, $D(P)$ gives us new means to assume an atom. Whereas in $C_{\rightarrow}(P)$, by assuming A, we could gain nothing more than A itself (via (I)), in $D(P)$ we can assume A (via $(P\vdash)$) by, so to speak, 'reflecting' on the meaning of A. If $\mathbf{D}(A)$ is empty, $(P\vdash)$ permits to infer $X, A \vdash F$ for any X and F. In particular, we can prove $X, \bot \vdash F$ for any zero-place predicate $\bot$ which is not head of a program rule (see above). This means that $D(P)$ has a genuine notion of falsity and thus of negation (see Section 4).

In $D(P)$ the conceptual difference between implication formulae and program rules remarked earlier (Section 5 of Part I) becomes extremely important. According to the reading of a logic program which underlies the calculus $D(P)$, to add a rule to a program means to change the definition of an atom by giving a new introduction rule for it, whereas by assuming an implication formula the meaning of predicates as given by the program is not changed. One of the most important consequences of this fact is that for $D(P)$ transitivity in the sense of (10) [Section 5 of Part I] is lost. Take for example the program P consisting of the single rule $\{p \rightarrow q\} \Rightarrow p$. The following derivation shows that we have both $p \vdash_{D(P)} q$ and $\vdash_{D(P)} p$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}(I) \qquad \dfrac{}{p,q \vdash q}(I)}{p,(p \rightarrow q) \vdash q}(\rightarrow\vdash)}{p \vdash q}(P\vdash)}{\vdash p \rightarrow q}(\vdash\rightarrow)}{\vdash p}(\vdash P)$$

However, $\vdash_{D(P)} q$ does not hold, because there is no program rule such that $\vdash q$ can be obtained via $(\vdash P)$ (which is the only schema of $D(P)$ which permits inference of a sequent of the form $\vdash A$). This result is not unwanted at all. If transitivity held for $D(P)$, the atom q would be derivable in the program above without assumptions. According to our definitional reading of programs this would mean that it would be possible to establish an atom

starting with a meaningless predicate, since there is no introduction rule for it in the program.

The schemas (I), $(\vdash\rightarrow)$ and $(\rightarrow\vdash)$ give the basic (= program-independent) interpretation of $\rightarrow$. Now in $D(P)$ we may use *any*$\rightarrow$-condition to form definitional clauses. Since there is no restriction we can of course not expect the interpretation of these programs as definitions to be *total* in general. We can write down programs that do not give any sensible definition of certain atoms as the atom p in the example program just mentioned shows. The failure of transitivity may be viewed as a reflection of the fact that the rules of a program sometimes only *partially* define the meaning of a predicate (see [10]). This situation can be compared with that of a language based on partial recursive functions. In both cases we have a very general and elementary framework for presentations of definitions, and in both cases the framework is general enough to give definitions that have no total interpretations.[2]

However, those problems only arise with programs containing implications in the premises of rules. A definite Horn clause program which contains no implications (and thus corresponds to a monotone inductive definition), will always represent a total definition in the sense that transitivity holds:

PROPOSITION 1
If P is a definite Horn clause program (i.e. a program without implications in the premises of clauses), then the following holds: If $X\vdash_{D(P)}F$ and $Y, F\vdash_{D(P)}G$, then $X, Y\vdash_{D(P)}G$.

PROOF. We prove the proposition by induction on the lexical ordering of pairs (m, k), where m is the complexity of F and k is the sum of the lengths of the derivations of $X\vdash F$ and $Y, F\vdash G$:

If $X\vdash F$ or $Y, F\vdash G$ is an initial sequent (i.e. inferred by schema (I)), then we are immediately done. If the last step in the derivation of $X\vdash F$ uses $(\rightarrow\vdash)$ or $(P\vdash)$, the assertion follows by straightforward application of the induction hypothesis with respect to the second component of (m, k), and similarly, if the last step in the derivation of $Y, F\vdash G$ uses $(\vdash\rightarrow)$ or $(\vdash P)$, or uses $(\rightarrow\vdash)$ or $(P\vdash)$ with F not being introduced at this step.

So assume $F = F_1\rightarrow F_2$ and consider the following situation:

$$\frac{X, F_1\vdash F_2}{X\vdash F_1\rightarrow F_2}\qquad\frac{Y\vdash F_1\quad Y, F_2\vdash G}{Y, F_1\rightarrow F_2\vdash G}$$

[2] The definitional reading of programs is based on an attempt to give a partial interpretation of possibly non-monotonic operators (see [10]). So clearly there are connections with notions like Clark's 'completion' and McCarthy's 'circumscription'. We cannot discuss them in this paper, but only point to the fact that here are a lot of questions of interest. (Some aspects of these questions are discussed in [8] and [9]). A particularly interesting point is the relationship between the semantics given by $D(P)$ and the three-valued semantics proposed by Mycroft, Fitting and Kunen (see [12]).

We apply the induction hypothesis to the derivations of $X, F_1 \vdash F_2$ and $Y, F_2 \vdash G$ since the first component of (m, k) is lowered, obtaining a derivation of $X, Y, F_1 \vdash G$. By using the induction hypothesis with respect to this derivation and that of $Y \vdash F_1$ we obtain $X, Y \vdash G$.

Now assume $F = A$ and consider the following situation:

$$\frac{X \vdash Z \quad Y, \mathbf{D}(A) \vdash G}{X \vdash A \quad\quad Y, A \vdash G}$$

where $Z \in \mathbf{D}(A)$. Since P is a definite Horn clause program, the elements of Z are atomic. Therefore the first component of (m, k) does not increase by considering the derivations of $X \vdash Z$ and $Y, Z \vdash G$ (the latter is contained in $Y, \mathbf{D}(A) \vdash G$). So we may apply the induction hypothesis (possibly several times) to these derivations due to the fact that the second component of (m, k) is lower. Thus we have $X, Y \vdash G$. (Note that if P were an arbitrary program, this argument would not be valid, since the elements of Z could be of higher complexity than A.) ⊣

2. Linear derivations: soundness and completeness

Much as in Section 3 of Part I we now define a calculus LD for linear derivations. We then demonstrate quite analogously to Theorems 1.1 and 1.2 that $D(P)$ and $LD(P)$ are equivalent, which gives us soundness and completeness results. The system $LD(P)$ contains as a subsystem the system $LC_{\rightarrow}(P)$ for linear derivations corresponding to $C_{\rightarrow}(P)$. So we obtain at the same time soundness and completeness of $LC_{\rightarrow}(P)$ in relation to $C_{\rightarrow}(P)$.

We define the following ordering on substitutions: $\theta < \sigma$ iff $\theta\delta = \sigma$ for a δ which is nonempty and is not a renaming substitution. Given an atom A, a substitution σ is called *A-sufficient* if for $A\sigma$ the proviso for the application of $(P\vdash)$ in $D(P)$ is fulfilled, i.e., $\mathbf{D}(A\sigma\tau) = (\mathbf{D}(A\sigma))\tau$ for all τ. A substitution σ is called *minimal A-sufficient* if σ is A-sufficient and there is no $\theta < \sigma$ which is A-sufficient. Since obviously, given σ, there can be no infinite descending chain with respect to $<$ starting from σ, minimal A-sufficient substitutions exists given an A-sufficient substitution. In principle such minimal A-sufficient substitutions are computable if they exist (although they are not unique). However, the special algorithmic problems of such computations are not essential for the following completeness proof. (They are treated in Section 3 below.)

The linear calculus $LD(P)$ serves for the derivation of pairs $\langle \Sigma, \sigma \rangle$ where Σ is a set of sequents and σ a substitution. As in Section 3 of Part I, we say that σ and τ agree on Σ if σ and τ become identical after deleting all bindings for variables not in Σ. The sign $\dot{\cup}$ in $\langle \Sigma \dot{\cup} \Gamma, \sigma \rangle$ expresses that Σ and Γ are assumed to be disjoint, and in $\Sigma \dot{\cup} \{X \vdash A\}$ the sequent $X \vdash A$ is

considered the element selected from the goal $\Sigma \dot{\cup} \{X \vdash A\}$. The inference schemata for $LD(P)$ are the following:

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

$$\sigma\frac{\langle \Sigma\sigma, \theta \rangle}{\langle \Sigma \dot{\cup} \{X, A \vdash B\}, \sigma\theta \rangle}(I)_L$$

where σ is a unifier of A and B

$$\frac{\langle \Sigma \cup \{X, F \vdash G\}, \theta \rangle}{\langle \Sigma \dot{\cup} \{X \vdash F \to G\}, \theta \rangle}(\vdash\to)_L$$

$$\frac{\langle \Sigma \cup \{X \vdash F\} \cup \{X, G \vdash H\}, \theta \rangle}{\langle \Sigma \dot{\cup} \{X, (F \to G) \vdash H\}, \theta \rangle}(\to\vdash)_L$$

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

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

$$\sigma\frac{\langle \Sigma\sigma \cup (X\sigma, \mathbf{D}(A\sigma) \vdash F\sigma), \theta \rangle}{(\Sigma \dot{\cup} \{X, A \vdash F\}, \sigma\theta \rangle}(P\vdash)_L$$

provided σ is an A-sufficient substitution.

By $LC_{\to}$ we denote the system which results from LD by taking away $(P\vdash)_L$. The notions defined in Section 3 of Part I are carried over to the present case in the following way. The substitution σ mentioned to the left of an inference line is called the substitution *used* at that application of the inference, similarly we say that $Y \Rightarrow A$ is used at applications of $(\vdash P)_L$; the variables in $Y \Rightarrow A$ are called *rule variables*. If they occur nowhere else in the linear derivation, the linear derivation is called *purified*. A linear derivation is called *strict*, if the unifiers used at applications of $(I)_L$ and $(\vdash P)_L$ are most general and the A-sufficient substitutions used at applications of $(\vdash P)_L$ are minimal. If a linear derivation of $\langle \Sigma, \tau \rangle$ is given, τ can be written as $\theta\delta$, where δ is the *initial substitution* of the step $(\varnothing)_L$ and θ the *computed substitution*.

If we restrict ourselves to $L\mathscr{C}_{\to}$, it can easily be seen that these derivations can be viewed as representing SLD-derivations of an extended kind. Instead of atoms one now selects sequents from the query to be evaluated. Then one either evaluates them according to $(\vdash\to)_L$ or $(\to\vdash)_L$ without unification, or, for a sequent $Z \vdash B$, one either tries to unify B with the head of a program rule via $(\vdash P)_L$ or with a member of the antecedent set Z via $(I)_L$. Concerning search strategies there are now more possibilities of branching than in the case of programming with usual definite Horn clauses. However,

unification still remains the central algorithm applied in the computation. This situation becomes different when we consider the usage of $(P\vdash)_L$ where something essentially new comes in (see Section 3).

In the following we shall establish soundness and completeness of $LD(P)$ with respect to the semantics given by the sequent calculus $D(P)$. If one omits everything that refers to the inference schemata $(P\vdash)$ and $(P\vdash)_L$, this is at the same time a soundness and completeness result for $LC_{\rightarrow}(P)$ with respect to the semantics given by $C_{\rightarrow}(P)$.

THEOREM 2.1
Suppose a linear derivation of $\langle \Gamma, \tau\delta \rangle$ with computed substitution τ and initial substitution δ is given. Then $\Gamma\tau$ is derivable in $D(P)$.

PROOF. The cases $(\varnothing)_L$ and $(\vdash P)_L$ are treated like the corresponding cases in the proof of Theorem 1.1. The cases $(\vdash\!\rightarrow)_L$ and $(\rightarrow\!\vdash)_L$, which do not involve any computation of substitutions, are handled by straightforward applications of the induction hypothesis.

Suppose the schema $(I)_L$ is used in the last step of the linear derivation of $\langle \Gamma, \tau\delta \rangle$:

$$\sigma\frac{\langle \Sigma\sigma, \theta \rangle}{\langle \Sigma \dot{\cup} \{X, A \vdash B\}, \sigma\theta \rangle}$$

Then θ is $\tau_1\delta$ where τ_1 is the computed substitution of the linear derivation of $\langle \Sigma\sigma, \theta \rangle$. Therefore by induction hypothesis we have a derivation of $\Sigma\sigma\tau_1$ in $D(P)$. Furthermore, since σ unifies A and B, by (I) we have a (trivial) derivation of $X\sigma\tau_1, A\sigma\tau_1 \vdash B\sigma\tau_1$ in $D(P)$. Since $\sigma\tau_1$ is τ, we have a derivation of $\Sigma\tau \cup \{X\tau, A\tau \vdash B\tau\}$ in $D(P)$.

Suppose $(P\vdash)_L$ is used in the last step in the linear derivation of $\langle \Gamma, \tau\delta \rangle$:

$$\sigma\frac{\langle \Sigma\sigma \cup (X\sigma, \mathbf{D}(A\sigma) \vdash F\sigma), \theta \rangle}{\langle \Sigma \cup \{X, A \vdash F\}, \sigma\theta \rangle}$$

Then θ is $\tau_1\delta$, where τ_1 is the computed substitution of the linear derivation of $\langle \Sigma\sigma \cup (X\sigma, \mathbf{D}(A\sigma) \vdash F\sigma), \theta \rangle$. Therefore by induction hypothesis we have a derivation of $\Sigma\sigma\tau_1 \cup (X\sigma\tau_1, (\mathbf{D}(A\sigma))\tau_1 \vdash F\sigma\tau_1$ in $D(P)$. Since σ is assumed to be A-sufficient, we have that $(\mathbf{D}(A\sigma))\tau_1 = \mathbf{D}(A\sigma\tau_1)$. And since $\sigma\tau_1$ is also A-sufficient, we can apply schema $(P\vdash)$ in $D(P)$ to obtain a derivation of $\Sigma\sigma\tau_1 \cup (\bar{X}\sigma\tau_1, A\sigma\tau_1 \vdash F\sigma\tau_1)$. This is the desired result, because $\tau = \sigma\tau_1$. ⊣

LEMMA 2.1
A linear derivation of $\langle \Gamma\tau, \theta \rangle$ with computed substitution γ can be transformed into one of $\langle \Gamma, \tau\theta \rangle$ with computed substitution $\gamma\theta$ provided τ does not act on rule variables in the given linear derivation. If the linear derivation of $\langle \Gamma\tau, \theta \rangle$ is purified, then so is that of $\langle \Gamma, \tau\theta \rangle$. The length of the linear derivation remains unchanged by this transformation.

PROOF. Only $(I)_L$ and $(P\vdash)_L$ have to be considered in addition to the proof of Lemma 1.1. Suppose the schema $(I)_L$ is used in the last step:

$$\sigma\frac{\langle \Sigma\tau\sigma,\ \delta\rangle}{\langle \Sigma\tau \dot{\cup} \{X\tau,\ A\tau \vdash B\tau\},\ \sigma\delta\rangle}$$

Since σ is a unifier of $A\tau$ and $B\tau$, $\tau\sigma$ is a unifier of A and B. Therefore we can replace this step by

$$\tau\sigma\frac{\langle \Sigma\tau\sigma,\ \delta\rangle}{\langle \Sigma \dot{\cup} \{X,\ A \vdash B\},\ \tau\sigma\delta\rangle}$$

If γ is the computed substitution of the linear derivation of $\langle \Sigma\tau \dot{\cup} \{X\tau,\ A\tau \vdash B\tau\},\ \sigma\delta\rangle$, then $\tau\gamma$ is the computed substitution of the linear derivation of $\langle \Sigma \dot{\cup} \{X,\ A \vdash B\},\ \tau\sigma\delta\rangle$.

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

$$\sigma\frac{\langle \Sigma\tau\sigma \cup (X\tau\sigma,\ \mathbf{D}(A\tau\sigma) \vdash F\tau\sigma),\ \delta\rangle}{\langle \Sigma\tau \dot{\cup} \{X\tau,\ A\tau \vdash F\tau\},\ \sigma\delta\rangle}$$

Since σ is $A\tau$-sufficient, $\tau\sigma$ is A-sufficient. Therefore we can replace this step by the following application of $(P\vdash)_L$:

$$\tau\sigma\frac{\langle \Sigma\tau\sigma \cup (X\tau\sigma,\ \mathbf{D}(A\tau\sigma) \vdash F\tau\sigma),\ \delta\rangle}{\langle \Sigma \dot{\cup} \{X,\ A \vdash F\},\ \tau\sigma\delta\rangle}$$

If γ is the computed substitution of the linear derivation of $\langle \Sigma\tau \dot{\cup} \{X\tau,\ A\tau \vdash F\tau\},\ \sigma\delta\rangle$, then $\tau\gamma$ is the computed substitution of the linear derivation of $\langle \Sigma \dot{\cup} \{X,\ A \vdash F\},\ \tau\sigma\delta\rangle$.

Neither the property of being purified nor the length of the considered derivation are changed by these transformations. ⊣

LEMMA 2.2

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

PROOF. With respect to applications of $(I)_L$, we can argue as in the proof of Lemma 1.2, since that argument only depended on the way a most general unifier is related to a (not necessarily most general) unifier σ used at an application of an inference schema. With respect to applications of $(P\vdash)_L$ we can also rely on this argument, taking into account that minimal A-sufficient substitutions are related to A-sufficient substitutions in the same way as most general unifiers are related to unifiers. (The uniqueness of most general unifiers, which has no analogue in the case of minimal A-sufficient substitutions, is not used in the argument.) ⊣

LEMMA 2.3
Suppose a purified derivation of Γ in $D(P)$ is given. Then there is a linear derivation of $\langle \Gamma, \tau \rangle$ such that τ only acts on V, where V is the set of rule variables of the derivation of Γ.

PROOF. We have to consider the case of applications of (I) and $(P\vdash)$ in the derivation in $D(P)$ assumed to be given.

Suppose Γ is $\Sigma \dot{\cup} \{X, A \vdash A\}$. By induction hypothesis there is a linear derivation of $\langle \Sigma, \tau \rangle$. By applying $(I)_L$, we obtain a linear derivation of $\langle \Sigma \dot{\cup} \{X, A \vdash A\}, \tau \rangle$. The condition on variables is fulfilled by the assumption that the given derivation of Γ in $D(P)$ is purified.

Suppose Γ is $\Sigma \dot{\cup} \{X, A \vdash F\}$ and the derivation of $X, A \vdash F$ uses the schema $(P\vdash)$ in the last step. Then by induction hypothesis there is a linear derivation of $\langle \Sigma \cup (X, \mathbf{D}(A) \vdash F), \tau \rangle$ where τ fulfils the variable condition stated in the assertion. Since the proviso for the application of $(P\vdash)$ is fulfilled, we know that the empty substitution is A-sufficient. Therefore by applying $(P\vdash)_L$ we obtain a linear derivation of $\langle \Sigma \dot{\cup} \{X, A \vdash F\}, \tau \rangle$. ⊣

THEOREM 2.2
From a derivation of $\Gamma\tau$ in $D(P)$ a purified strict linear derivation of $\langle \Gamma, \tau' \rangle$ can be obtained, such that τ and τ' agree on Γ.

PROOF. The proof is exactly the same as that of Theorem 1.2, with Γ in the place of Z and $D(P)$ in the place of $C(P)$. ⊣

As in the case of definition Horn clause programming, we may remark that we have proved an abstract completeness result with respect to a certain semantics of higher-level rules. This semantics is given by the inference schemata of the system $D(P)$ (or $C_{\neg}(P)$, if we restrict ourselves to the system without the definitional reading of program clauses). The completeness result is abstract, since we only assume that, given a unifier, a most general unifier exists and, given an A-sufficient substitution, a minimal A-sufficient substitution exists. For our completeness result we do not rely on any particular algorithm to compute a unifier or most general unifier, given two expressions A and B, or to compute an A-sufficient or minimal A-sufficient substitution, given an atom A.

3. Computation of *A*-sufficient substitutions

The linear calculus which has been shown to be complete with respect to the calculus $D(P)$ giving the semantics of generalized Horn clauses and their definitional reading, was considered an abstract description of a computation. It relied on the fact that there are A-sufficient substitutions for any A, even minimal ones, and that they can in principle be computed. Being in principle computable means that there is a recursive function of arbitrary

complexity by means of which they can be computed. In order to obtain an evaluation procedure that can be implemented, we need, however, an effective algorithm. In the following we describe an easy procedure which computes A-sufficient substitutions. The situation here is completely parallel to the one in ordinary definite Horn clause programming. When proving soundness and completeness of SLD-resolution, one does not rely on any specific unification algorithm. But such an algorithm is, of course, needed when one wants to implement the evaluation procedure. The algorithm for the computation of A-sufficient substitutions σ to be described in the following does not always compute *minimal* ones. This has to do with the fact that in this algorithm only the conclusions (i.e. heads) of program clauses are taken into account from which $A\sigma$ can be obtained by substitution, and not the premises as required by the definition of $\mathbf{D}(A\sigma)$. This restriction does however make the algorithm very efficient, which is important for implementation purposes.

In the following we throughout assume that variables can be arbitrarily renamed. Since the algorithm only acts on the conclusions of program clauses, we define

$$\mathbf{P}(A) = \{B \mid X \Rightarrow B \in P \text{ and } A = B\tau \text{ for some } \tau\}$$

In other words $\mathbf{P}(A)$ is defined to be the set of all heads of program clauses, which have A as a substitution instance. It is obvious that

$$\mathbf{P}(A) \subseteq \mathbf{P}(A\theta)$$

for any θ, for if $B\tau = A$ for some $X \Rightarrow B \in P$ and some substitution τ, then $B\tau\theta = A\theta$ for any θ. The converse

$$\mathbf{P}(A\theta) \subseteq \mathbf{P}(A)$$

is not true for all θ: Consider the program P consisting of the single rule $p(1) \Rightarrow q(1)$. Then $q(1) \in \mathbf{P}(q(x)\theta)$ where θ is the substitution which binds x to 1, but $q(1) \notin \mathbf{P}(q(x))$. Therefore the following condition

$$\mathbf{P}(A\sigma\theta) = \mathbf{P}(A\sigma) \text{ for all } \theta \qquad (*)$$

is a non-trivial condition on substitutions σ. We shall show that $(*)$ together with one further condition implies that σ is A-sufficient. We say that a program clause $X \Rightarrow B$ fulfils the *no-extra-variable-condition,* if every variable occurring in X also occurs in B. Suppose now that A is fixed. Then we say that σ passes the *variable-check,* if all program clauses $X \Rightarrow B$, for which $B\tau = A\sigma$ holds for some τ, fulfil the no-extra-variable-condition. Intuitively, the variable-check for σ means that every clause from which $A\sigma$ can be obtained by substitution, fulfils the no-extra-variable-condition.

PROPOSITION 2.1
If σ satisfies $(*)$ and passes the variable-check, then σ is A-sufficient.

PROOF. We have to show that $\mathbf{D}(A\sigma\theta) = (\mathbf{D}(A\sigma))\theta$ for any θ. Suppose $Y \in \mathbf{D}(A\sigma\theta)$, which means that there is a program clause $X \Rightarrow B$ such that $Y = X\tau$ and $A\sigma\theta = B\tau$ for some substitution τ. Since $(*)$ is assumed to hold for σ, there is a τ' such that $A\sigma = B\tau'$, therefore $B\tau = B\tau'\theta$. Since σ passes the variable-check, we can infer from $B\tau = B\tau'\theta$ that $X\tau = X\tau'\theta$ also holds. Hence, $Y = X\tau'\theta$. Since $A\sigma = B\tau'$, this means that $Y \in (\mathbf{D}(A\sigma))\theta$. ⊣

Now consider the algorithm which is described by the following recursive definition: Let $mgu(A, B)$ be the mgu of A and B if there is one and the empty substitution otherwise. Let '$\circ$' be an explicit notation for the composition of substitutions. Let $X_1 \Rightarrow B_1, \ldots, X_n \Rightarrow B_n$ be an ordering of all program clauses. Then we define

$$\sigma_0 = \varnothing$$
$$\sigma_{i+1} = \sigma_i \circ mgu(A\sigma_i, B_{i+1})$$
$$\sigma = \sigma_n$$

Intuitively, σ is computed by first trying to unify A with B_1, then, if successful, the unification result with B_2, and so on. If σ is the empty substitution, then either A is not unifiable with any B_i, or A is identical (and therefore unifiable) with at least one B_i. If σ is nonempty, then A is unifiable with at least one B_i. When implementing this procedure one would, of course, consider only those program clauses $X_i \Rightarrow B_i$ whose head B_i starts with the same predicate symbol as A. The algorithm for the computation of σ can be viewed as computing at the same time a set $U \subseteq \{B_1, \ldots, B_n\}$ in the following way: If at stage i of the computation of σ, $A\sigma_i$ is unifiable with B_{i+1}, then B_{i+1} is put into U. More formally, U can be defined as:

$$U = \{B_{i+1} \mid 0 \leq i < n, A\sigma_i \text{ is unifiable with } B_{i+1}\}$$

This set U is called the *set of heads of program clauses computed by the algorithm.*

PROPOSITION 2.2
Let U be the set of heads of program clauses computed by the algorithm and σ the computed substitution. Then U is a maximal subset of $\{B_1, \ldots, B_n\}$ such that $\{A\} \cup U$ is unifiable in the following sense:

(i) σ is an mgu of $\{A\} \cup U$
(ii) if $\{A\} \cup U'$ is unifiable, where $U \subseteq U' \subseteq \{B_1, \ldots, B_n\}$, then $U = U'$
(iii) $U = \mathbf{P}(A\sigma)$.

PROOF. (i) If U is empty, then $\{A\} \cup U = \{A\}$ and σ is empty, so that σ is an mgu of $\{A\} \cup U$ for trivial reasons.

If U is not empty, it can be written as $\{B^1, \ldots, B^m\}$ such that $A\sigma^1 = B^1\sigma^1$ and $B^i\sigma^i\sigma^{i+1} = B^{i+1}\sigma^{i+1}$ for most general unifiers $\sigma^1, \ldots, \sigma^m$, such that

$\sigma^1 \circ \ldots \circ \sigma^m = \sigma$. Then σ is a most general unifier of the *set* $\{A\} \cup U$ (and therefore independent of any order of its elements).

(ii) Suppose $\{A\} \cup U \cup \{B_j\}$ is unifiable for $B_j \notin U$. Then B_j is unifiable with $A\sigma$ since $\sigma = \sigma_{j-1}\delta$ for some δ. Then B_j is also unifiable with $A\sigma_{j-1}$, since the variables in B_j may be considered distinct from those bound by δ. But then B_j belongs to U, contrary to the assumption.

(iii) If $B \in U$, then $A\sigma = B\sigma$ by (i). Thus $B \in \mathbf{P}(A\sigma)$. Conversely, if $B \in \mathbf{P}(A\sigma)$, then $B\tau = A\sigma$ for some τ. Since variables in B can be assumed to be distinct from variables in A or $A\sigma$, we have $A\sigma = A\sigma\tau$. Thus $A\sigma$ and B are unifiable, i.e. $\{A\} \cup U \cup \{B\}$ is unifiable. By (ii) this means that $B \in U$. ⊣

Instead of the set U of heads of program clauses computed by the algorithm we may, for the substitution σ computed by the algorithm, consider the set Z_σ of clauses whose heads are in $\mathbf{P}(A\sigma)$, i.e., $Z_\sigma = \{X_i \Rightarrow B_i \mid B_i \in \mathbf{P}(A\sigma)\}$. It follows from Proposition 2.2 that if each element in Z_σ fulfills the no-extra-variable-condition, then σ passes the variable check. Therefore, by adding the check of the no-extra-variable-condition, the algorithm can easily be extended to an algorithm for the computation of σ which passes the variable check: compute σ as described above, then check the no-extra-variable-condition for Z_σ. If this condition is not fulfilled, the σ computed is not appropriate, and the whole procedure has to start again by choosing another permutation of the program clauses. This choice of another permutation is a step in backtracking. There are of course various possibilities in determining the choice of another permutation which differ with respect to efficiency that we do not discuss here. It may be pointed out that the check of the no-extra-variable-condition poses no specific problem, since this check can be performed by preprocessing the whole program.

From the following result we can, together with Proposition 2.1, infer that we have obtained an algorithm for the computation of A-sufficient substitutions:

PROPOSITION 2.3
If σ is computed by the algorithm described, then σ satisfies (*).

PROOF. Suppose $B \in \mathbf{P}(A\sigma\theta)$, i.e. $A\sigma\theta = B\tau$ for some τ. Then B and $A\sigma$ are unifiable, since the variables in B may be renamed arbitrarily. Then by Proposition 2.2 (i), B is unifiable with every element of U, where U is the set of heads of program clauses computed by the algorithm. Hence $B \in U$ by Proposition 2.2 (ii). Therefore $B\sigma = A\sigma$ by Proposition 2.2 (i), i.e. $B \in \mathbf{P}(A\sigma)$. ⊣

The following result shows that the substitution computed by the algorithm is a minimal substitution satisfying the condition (*).

PROPOSITION 2.4
Suppose σ is computed by the algorithm. Suppose σ' satisfies $(*)$ and $\sigma = \sigma'\delta$ for some δ. Then $A\sigma$ and $A\sigma'$ are variants of each other (i.e. are equal modulo renaming).

PROOF. Since $\sigma = \sigma'\delta$, we have $\mathbf{P}(A\sigma) = \mathbf{P}(A\sigma'\delta)$. Therefore $\mathbf{P}(A\sigma) = \mathbf{P}(A\sigma')$, since σ' is supposed to satisfy $(*)$. If $\mathbf{P}(A\sigma) = \varnothing$ then σ is the empty substitution. Since $\sigma = \sigma'\delta$, this implies that σ' is a renaming substitution and the assertion is fulfilled. Now suppose $\mathbf{P}(A\sigma) = \{B_1, \ldots, B_m\}$. Then $B_1\tau_1 = \ldots = B_m\tau_m = A\sigma'$ for certain $\tau_1, \ldots, \tau_m$. Since we may assume that the variables in B_i and B_j are distinct for any $i \neq j$ and distinct from variables in A and $A\sigma'$ and variables bound by σ', we have that $\sigma'\tau_1 \ldots \tau_m$ unifies the set $\{A, B_1, \ldots, B_m\}$. On the other hand we have by Proposition 2.2 (i) and (iii) that σ is an mgu of $\{A, B_1, \ldots, B_m\}$. Thus $\sigma'\tau_1 \ldots \tau_m = \sigma\theta$ for some θ. Hence $A\sigma' = A\sigma\theta$ since $\tau_1 \ldots \tau_m$ does not contain bindings for variables in A. By assumption we also have $A\sigma = A\sigma'\delta$. This means that $A\sigma$ and $A\sigma'$ are variants of each other. ⊣

We have shown that our algorithm computes a minimal substitution satisfying $(*)$ and therefore, when extended with the variable check, computes an A-sufficient substitution by Proposition 2.1. This A-sufficient substitution need not be minimal, as the following simple example shows:

Consider the program consisting of the two clauses

$$q(x) \Rightarrow p(x, a)$$

$$q(y) \Rightarrow p(a, y)$$

Then both clauses fulfil the no-extra-variable-condition, where a is an individual constant and x and y are individual variables. Let A be $p(x, y)$. Then the A-sufficient unifier computed by the algorithm is $\sigma = \{x/a, y/a\}$ (independently of in which order the program clauses are considered when performing the algorithm). However, both $\sigma_1 = \{x/a\}$ and $\sigma_2 = \{y/a\}$ are A-sufficient and both $\sigma_1 < \sigma$ and $\sigma_2 < \sigma$ hold. In fact, both σ_1 and σ_2 are minimal A-sufficient, whereas σ is not. The obvious reason is that the algorithm 'does too much', when successively unifying heads of program clauses, not taking into account whether new bodies of program clauses are obtained. On the other hand, extensive checking of the bodies of clauses strongly impairs the effectiveness of execution.

4. Falsity

This section relies throughout on the definitional reading of program rules which underlies the sequent calculus $D(P)$. Assume $\perp$ is a zero-place predicate constant which has not been given any definition by the program P, i.e., $\perp$ is not head of a program rule. Then $\mathbf{D}(\perp)$ is empty, and by $(P\vdash)$

we can derive $X, \perp \vdash F$ for any X and F. An implication formula F is said to be *true* according to P if $\vdash_{D(P)} F$, and *false* if $F \vdash_{D(P)} \perp$. So the connection of falsity with the intuitive concept of *negation* is simply that $\perp$ is any expression that is *not* defined by the given program P. Thus if $\vdash_{D(P)}$ is a natural notion of consequence, $A \vdash_{D(P)} \perp$ should imply that $\vdash_{D(P)} A$ does not hold, for which it suffices to know that $\vdash_{D(P)}$ is transitive.

LEMMA 3
If F is false according to P, then $X, F \vdash G$ is derivable in $D(P)$ for any X and G.

PROOF. By induction on the length of derivations of $F \vdash \perp$ in $D(P)$. If (I) has been applied in the last step, then F is $\perp$, and we use that $X, \perp \vdash G$ is derivable. The schema $(\vdash\!\!\rightarrow)$ cannot be applied in the last step since $\perp$ is an atom. If $(\rightarrow\!\vdash)$ has been applied in the last step, we use the induction hypothesis with respect to the right premise of this application. The schema $(\vdash P)$ cannot be applied in the last step, since by assumption no program rule has $\perp$ as its head. If $(P\vdash)$ has been applied in the last step, we use the induction hypothesis with respect to the premise of this application. $\dashv$

To understand *F is false in P* really means to understand what kind of notion of *consequence* $\vdash_{D(P)}$ represents. Intuitively, a program (definition) P generates a *local* notion of consequence $\vdash_{D(P)}$. If this 'consequence relation' is transitive one may view $\vdash_{D(P)}$ as a natural notion of consequence given by P viewed as a 'logic'. But still $F \vdash_{D(P)} \perp$ may be much weaker than simply *not* $\vdash_{D(P)} F$. If either $\vdash_{D(P)} F$ or $F \vdash_{D(P)} \perp$, then we may say that P *decides* F. $F \vdash_{D(P)} \perp$ is an *internal* (*intensional*) notion while *not* $\vdash_{D(P)} F$ is an *external* (*extensional*) one: in the first case we can see that F is false by using the tools P itself provides us with, but in the second case we may use any technique whatsoever to establish that F does not hold. Analogously, $X \vdash_{D(P)} F$ means that F follows from X *within* P and not something like *if* $\vdash_{D(P)} X$, *then* $\vdash_{D(P)} F$. This is the basic reason why it is natural to view $\vdash_{D(P)}$ as a local notion of consequence intrinsic to P considered as the basis of a logic. It is easy to see that the internal and external perspectives will coincide for a set of sequents if and only if P is locally complete in a certain strong sense, i.e. if transitivity holds for $\vdash_{D(P)}$ for a certain collection of sets of rules and P decides a large enough collection of atoms (see [10]).

Now we consider the connection between a certain notion of finite failure for sequents and the proposed notion of falsity. What we want to establish is that if F is ground and $\vdash F$ fails finitely, then F is false, or more generally that if X and F are ground and $X \vdash F$ fails finitely, then all elements of X are true and F is false (according to the program P). This means that the

inference schema of negation as finite failure, formulated as

$$(\text{NFF})\ \frac{\vdash F \text{ fails finitely}}{F \vdash \bot}\ \text{provided } F \text{ is ground}$$

is a derived inference schema of $D(P)$. Obviously, (NFF) is not an inference schema in the usual sense, because in its premise it refers to computational aspects of proofs. This is why one calls an inference schema like (NFF), which in its standard formulation allows one to pass over from the finite failure of a ground atom to the negation of this atom, a 'meta-rule'. In practically all logic programming systems which contain a notion of negation, a schema corresponding to (NFF) is at least implicitly used as a primitive schema defining negation. This would also have to be the case if we wanted to add a notion of negation or falsity to a logic programming system based on $C_{\rightarrow}(P)$. Systems based on $D(P)$ do not need schemata like (NFF) because of the schema $(P\vdash)$, which so to speak directly incorporates genuine negation into the calculus. To define negation by failure, we need a notion of a computational successor of a sequent.

The *computational successors* (in short: *successors*) of a sequent, which are sets of sequents, are defined as follows:

(i) The empty set $\varnothing$ is a successor of $X, A \vdash B$ if A and B are unifiable;
(ii) $\{X, F \vdash G\}$ is a successor of $X \vdash F \rightarrow G$;
(iii) $(X - \{F \rightarrow G\} \vdash F) \cup \{X - \{F \rightarrow G\}, G \vdash H\}$ is a successor of $X, (F \rightarrow G) \vdash H$;
(iv) $X\sigma \vdash Y\sigma$ is a successor of $X \vdash B$ if σ is an mgu of A and B for some rule $Y \Rightarrow A$ which is a program rule or a variant thereof;
(v) $X\sigma - \{A\sigma\}, \mathbf{D}(A\sigma) \vdash H\sigma$ is a successor of $X, A \vdash H$ if σ is a minimal A-sufficient substitution.

In this definition, the clauses (iii) and (v) differ from what would be expected by simply reading the inference schemata of $LD(P)$ backwards. In the antecedents of sequents in successors, we used in (iii) '$X - \{F \rightarrow G\}$' rather than 'X' and in (v) '$X\sigma - \{A\sigma\}$' rather than '$X\sigma$' in order to give the notion of a computational successor a non-trivial rendering. What we have essentially done is considering a contraction-free variant of $D(P)$ and $LD(P)$. Otherwise, if Z contains at least one implicational formula $F \rightarrow G$, the sequent $Z, G \vdash H$ would be in one of the successors of $Z \vdash H$, and $Z, G \vdash H$ would be in one of its own successors. Similarly, if Z contains at least one atom for which a minimal A-sufficient substitution σ exists, the sequent $Z\sigma, \mathbf{D}(A\sigma) \vdash H\sigma$ would be in one of the successors of $Z \vdash H$, and $Z\sigma, \mathbf{D}(A\sigma) \vdash H\sigma$ would be in one of its own successors. However, we do not require that computational successors must be defined in exactly the way proposed. In particular, there are several possibilities of restricting contrac-

tion, which differ in strength. In the following, we only assume that some notion of computational successor like the one defined above is given.

We then say that a sequent $X \vdash F$ *fails finitely* if every computational successor of $X \vdash F$ contains an element which fails finitely. We call a program P *normal,* if the body of a rule in P does not contain variables beyond those occurring in the head of the rule. Using the terminology introduced in Section 1, this can also be expressed by requiring that $\mathbf{D}(A\sigma) = (\mathbf{D}(A))\sigma$ for all ground A.

One may think of the following proposition as expressing the soundness of a certain notion of finite failure.

THEOREM 3.1
Let P be a normal program. If $X \vdash H$ is ground and fails finitely, then all elements of X are true and H is false according to P(i.e. $\vdash X$ and $H \vdash \bot$ are derivable in $D(P)$).

PROOF. By induction on the notion '$X \vdash H$ fails finitely'. First we observe that $X \vdash H$ cannot have the empty set as a successor, since every successor to $X \vdash H$ must contain an element which fails finitely. Next we consider the possible successors of $X \vdash H$ according to clauses (ii)–(v) of the definition of a computational successor.

We first assume that $X \vdash H$ has a successor according to the clauses (ii) and (iv) (i.e. $X \vdash H$ has a successor which results from evaluating the succedent H of $X \vdash H$).

(ii) H is $F \rightarrow G$, and $\{X, F \vdash G\}$ is a successor of $X \vdash H$. Since $X, F \vdash G$ fails finitely, we have by induction hypothesis that $\vdash X$, F and $G \vdash \bot$ are derivable in $D(P)$. Since from $\vdash F$ and $G \vdash \bot$ we can obtain $F \rightarrow G \vdash \bot$ by application of $(\rightarrow\vdash)$, both $\vdash X$ and $H \vdash \bot$ are derivable in $D(P)$.

(iv) H is A, and for each Y in $\mathbf{D}(A)$, $X \vdash Y$ is a successor of $X \vdash H$ (remember that A is ground). Each Y must contain a G such that G is ground (since P is normal) and $X \vdash G$ fails finitely. Thus by induction hypothesis, $G \vdash_{D(P)} \bot$ from which by $(P\vdash)$ we obtain $A \vdash \bot$. The induction hypothesis also gives $\vdash X$.

We now suppose that $X \vdash H$ has no successors according to clauses (ii) and (iv) (i.e. $X \vdash H$ has no successor which results from evaluating H rather than some element of X). Then H must be an atom A such that $\mathbf{D}(A) = \varnothing$. This means that $H \vdash \bot$ is derivable in $D(P)$ by means of $(P\vdash)$. Thus, when considering successors of $X \vdash H$ according to clauses (iii) and (v) of the definition of a computational successor, we only have to show that $\vdash X$ is derivable in $D(P)$.

(iii) X is Z, $(F \rightarrow G)$ and $\{Z - \{F \rightarrow G\} \vdash F\} \cup \{Z - \{F \rightarrow G\}, G \vdash H\}$ is a successor of $X \vdash H$. By induction hypothesis, we have either $\vdash_{D(P)} Z - \{F \rightarrow G\}$ and $F \vdash_{D(P)} \bot$, or $\vdash_{D(P)} Z - \{F \rightarrow G\}$, $\vdash_{D(P)} G$ and $H \vdash_{D(P)} \bot$. In the first case we obtain from Lemma 3 $F \vdash_{D(P)} G$ and thus $\vdash_{D(P)} F \rightarrow G$, in the

second case we obtain $\vdash_{D(P)} F \rightarrow G$ from $\vdash G$. Thus in both cases we have $\vdash_{D(P)} Z, (F \rightarrow G)$, i.e. $\vdash_{D(P)} X$.

(v) X is Z, A and $Z - \{A\}, \mathbf{D}(A) \vdash H$ is a successor of $X \vdash H$ (we again use that A is ground and P is normal). Then, by induction hypothesis, we have $\vdash_{D(P)} Z - \{A\}$ and $\vdash_{D(P)} Y$ for some Y in $\mathbf{D}(A)$. By $(\vdash P)$ we obtain $\vdash_{D(P)} A$ and thus $\vdash_{D(P)} X$. ⊣

We consider finite failure only in connection with ground sequents and normal programs. This is to say that all ground A have a definite meaning according to P, which is a natural condition on a definition. Now of course lots of programs—'computation' programs, etc.—do not satisfy this condition for all ground A. So there is room for refinements of this theorem. Furthermore, this does not mean that our notion of falsity is useless for such programs, but only that the relationship established between finite failure and our notion of falsity does not hold in general in such cases.

It is easy to see that the converse of Theorem 3.1—which would express the completeness of a notion of finite failure—does not hold in general. Consider the following program for the unary predicate t, the individual constant a (representing an atomic proposition), and the variables x and y ranging over terms built up from a by means of the function symbols $\neg$ and $\wedge$:

$$t(a) \Rightarrow t(a)$$

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

$$(t(x) \rightarrow \bot) \Rightarrow t(\neg x)$$

The sequent $t(a \wedge \neg a) \vdash \bot$ is derivable in $D(P)$, as the following derivation shows:

$$\dfrac{\dfrac{\dfrac{t(a) \vdash t(a) \qquad t(a), \bot \vdash \bot}{t(a), (t(a) \rightarrow \bot) \vdash \bot}\ (\rightarrow\vdash)}{t(a), t(\neg a) \vdash \bot}\ (P\vdash)}{t(a \wedge \neg a) \vdash \bot}\ (P\vdash)$$

However, it can easily be seen that $\vdash t(a \wedge \neg a)$ does not fail finitely.

If we restrict ourselves to definite Horn clause programs, i.e. programs without implications in premises of clauses, completeness of finite failure can be achieved in the following sense:

THEOREM 3.2
If P is a normal definite Horn clause program, A is ground and $A \vdash_{D(P)} \bot$, then $\vdash A$ fails finitely.

PROOF. First we prove for definite Horn clause programs P that if $X \vdash_{D(P)} \bot$ for a set of atoms X, then $B \vdash_{D(P)} \bot$ for some $B \in X$, whereby the derivation of $B \vdash \bot$ is not longer than that of $X \vdash \bot$. We use induction on the length of

the derivation of $X \vdash \bot$: if it consists of an initial sequent, the assertion is trivially fulfilled. If this is not the case, then, since $\bot$ is not defined by the program and X consists only of atoms, we have the following situation:

$$\frac{Z, \mathbf{D}(B) \vdash \bot}{Z, B \vdash \bot}$$

where $X = Z \cup \{B\}$. By induction hypothesis there is for each $Y \in \mathbf{D}(B)$ an $A \in Z \cup Y$ such that $A \vdash_{D(P)} \bot$. If $A \in Z$ the assertion is fulfilled. So assume there is an $A \in Y$ for each $Y \in \mathbf{D}(B)$ such that $A \vdash_{D(P)} \bot$. But then $\mathbf{D}(B) \vdash_{D(P)} \bot$, so $B \vdash_{D(P)} \bot$. Clearly the length of the derivation does not increase.

Now we prove the main assertion by induction on the length of the derivation of $A \vdash \bot$: If this derivation consists of the initial sequent $\bot \vdash \bot$, the assertion is satisfied, since $\vdash \bot$ fails finitely. If the derivation does not consist of an initial sequent we have the following situation:

$$\frac{\mathbf{D}(A) \vdash \bot}{A \vdash \bot}$$

Assume $\vdash A$ has a successor $\vdash X$ where $X \in \mathbf{D}(A)$. Since P is a normal program, X consists of ground atoms. We have $X \vdash_{D(P)} \bot$, so $B \vdash_{D(P)} \bot$ for some ground $B \in X$. Therefore by induction hypothesis, since the derivation of $B \vdash \bot$ is not longer than that of $X \vdash \bot$, $\vdash B$ fails finitely. So $\vdash A$ fails finitely. $\dashv$

To illustrate the relationship between our notion of finite failure and our notion of falsity, we give the following simple example. Let P be the program consisting of the rules

$$\Rightarrow t(a)$$

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

where p is a given atomic proposition and x, y range over propositions in the pure implication calculus.

Then the sequent $\vdash t(a \supset b)$ fails finitely as can be seen in the following way: $\vdash t(a \supset b)$ has $\{\vdash t(a) \rightarrow t(b)\}$ as its only successor (according to (iv)). $\vdash t(a) \rightarrow t(b)$ has $\{t(a) \vdash t(b)\}$ as its only successor (according to (ii)). The only successor of $t(a) \vdash t(b)$ is $\{\vdash t(b)\}$ (according to (v)). The sequent $\vdash t(b)$, however, has no successor. The corresponding derivation in $D(P)$ showing that $\vdash t(a \supset b)$ is false, which according to Theorem 3.1 exists, runs as follows:

$$\dfrac{\dfrac{\dfrac{}{\vdash t(a)}\,(\vdash P) \qquad \dfrac{}{t(b) \vdash \bot}\,(P \vdash)}{t(a) \rightarrow t(b) \vdash \bot}\,(\rightarrow \vdash)}{t(a \supset b) \vdash \bot}\,(P \vdash)$$

If we add the rule $t(x) \Rightarrow t(x)$ to the program P, yielding an extended program P', the situation changes. Although $t(a \supset b)$ is not true according to P' (and thus 'fails' extensionally), it is not false according to P', i.e. $t(a \supset b) \vdash \bot$ is not derivable in $D(P)$. This is due to the fact that $t(b)$, though false according to P, is not false according to P'. Whereas in P the atom $t(b)$ was not given any 'introduction rule', in P' it is given one (namely the instance $t(b) \Rightarrow t(b)$ of the program rule $t(x) \Rightarrow t(x)$), so that it is no longer considered completely meaningless. This shows that falsity in our sense represents an intensional notion of negation which expresses something like 'falsity by definition'.

5. Discussion and examples

The main purpose of this paper was to treat logic programming from a proof-theoretic point of view. We discussed the declarative reading of a program in proof-theoretic terms by means of sequent systems. Then, we gave an operational interpretation in terms of linear derivations. Since we have not dealt here with algorithms for constructing linear derivations and their complexity, we may speak of a *static* operational model describing the structure of the general search space. Finally, we have established connections between the declarative semantics and the operational model, again, in a proof-theoretic context.

From a declarative point of view, the extension of logic programming discussed in Part II gives an interpretation of hypothetical reasoning based on a definitional reading of programs. In the static operational model this extension gives rise to new points where substitutions are computed: we may *bind* variables in assumptions. A single goal is a sequent

$$X \vdash A$$

with two sides: conclusion (succedent) A and assumptions (antecedent) X. For each of these sides there is a schema ($(\vdash P)$ and $(P \vdash)$), which, in the operational interpretation, requires the computation of bindings to variables ($(\vdash P)_L$ and $(P \vdash)_L$). The schema (I) for initial sequents, which can be viewed as establishing a communication link between the two sides, also leads computationally (i.e. as $(I)_L$) to bindings.

This means that the structure of queries is rather rich. Since we may ask hypothetical queries, we do not have to rely on any facts in the given program (see example 1 below). We can also make a distinction between asking for bindings in the conclusion of a sequent like

$$p \vdash q(x)$$

(for which t does $q(t)$ follow from p in the given program?), and asking for

bindings in the assumptions of a sequent like

$$q(x) \vdash p$$

(for which t does p follow from $q(t)$ in the given program?). Here we mean genuine bindings which are obtained by directly evaluating $q(x)$ and are not communicated by the other side of the sequent via the schema (I) or come in through the evaluation of other sequents of the goal to which the sequent in question belongs. A special case is to ask for bindings in negative queries

$$q(x) \vdash \bot$$

This means that even in the case of definite Horn clause programs (without $\rightarrow$ in bodies of clauses), we have a real extension of the reasoning mechanism as compared to 'standard' definite Horn clause programming. By using $\rightarrow$ in bodies of clauses (generalized Horn clause programs) we have a further extension of expressive power.

It is clear that this type of extension is not obtained for free when it comes to dynamical aspects of the operational model, i.e. when we also consider questions concerning the actual construction of linear derivations *in* time. We have added many new choicepoints to the well-known search space for definite Horn clauses based on the resolution rule. This makes the procedural part of our approach much more complex. If one thinks of logic programming as a purely declarative activity, i.e. the writing of a definition with a logical reading in mind, then one might perhaps argue that the increased complexity due to the $(P\vdash)$ schema essentially means to switch from *programming* to *theorem proving*. If, on the other hand, one is prepared to follow the present practice of logic programming and view procedural matters (such as search strategies, etc.) as part of the programming activity itself and not just as implementation issues, then the added complexity is not only a burden to be handled in one way or the other, but it also opens up new possibilities for a systematic view of control matters. The reason for this is that the added complexity is mainly introduced via a schema dual to the resolution rule, i.e. a rule, which, like the resolution rule, works on atomic formulae. This means that the complexity of the search space is not just a more or less chaotic mess centred around only one pole, the resolution rule, but that there is a natural duality in the structure centred around two poles: the resolution rule and its dual.

Based on these ideas, two experimental programming languages, GCLA I (see [1]) and GCLA II (see [2]), have been designed and implemented. GCLA I is based on a fixed search order, corresponding to PROLOG's depth first search, with a set of primitive control operators added. Experiments with GCLA I have shown that one needs a more flexible machinery for the control part than just a set of primitive operators without internal structure. GCLA II is designed with the aim of providing more

flexible control tools. The general control mechanism introduced in GCLA II can be specified in terms of partial inductive definitions over a suitable universe of sequents, i.e. the control part has a declarative interpretation in the given framework.

If one takes away the $(P\vdash)$-schema and its associated evaluation principle $(P\vdash)_L$, then our system corresponds *extensionally* to subsystems of Beeson [3], Gabbay and Reyle [5, 6] and Miller [15], who also use iterated implications as assumptions (called 'hereditary Harrop formulas' by Miller). By this correspondence, we mean that programs and queries can be translated into each other and yield the same answer substitutions. So there is a common core in present proof-theoretic approaches to logic programming that can be described by reference to iterated implications in clause bodies. In Beeson's approach it is part of a specific approach to theorem proving, in Gabbay's and Reyle's it is extended to systems including classical logic by means of a new inference principle ('restart rule') and in Miller's system higher-order quantification is used. *Intensionally,* however, there is still a fundamental difference between these approaches and ours, even with respect to this common core (i.e. without $(P\vdash)$), since they translate clauses into formulae and treat them as assumptions, whereas we keep them conceptually separate from assumptions (even if technically this difference comes to bear only in the context of the definitional approach with $(P\vdash)$).

Although in this paper we have basically relied on the structural principles of intuitionistic logic, our approach also permits a computational treatment of logics with restricted structural postulates ('substructural logics' like relevant logics, BCK logic, linear logic, Lambek calculus) which are receiving increasing attention in logic and computer science. Since the restrictions on structural postulates have to do with the structuring of assumptions, our approach with hypothetical queries represents a natural framework for considering such systems. Furthermore, it turns out that a schema like $(P\vdash)$ is a natural way to gain the power of logical elimination inferences (or left-introduction inferences) in such systems (see [21]). While in intuitionistic logic, it is in principle possible to formulate elimination inferences as specific program rules such as, e.g.,

$$t(p \vee q), (t(p) \rightarrow t(c)), (t(q) \rightarrow t(c)) \Rightarrow t(c)$$

this is not viable in certain substructural logics, since there the assumptions being discharged ($t(p)$ and $t(q)$ in the above example) may be embedded in heavily structured contexts. So, from the viewpoint of modelling logics in a uniform way, $(P\vdash)$ is fundamental, since it allows us to incorporate a wide area of logic into logic programming (see [20]). According to our definitional view, logic programming contains the computational treatment of logic itself in the sense that the introduction rules contain the computa-

tionally basic semantic content and thus form the database of logical reasoning.

We conclude by giving three examples that illustrate the functioning of our approach in an elementary way. The first example gives a simple illustration of the duality between $(\vdash P)$ and $(P\vdash)$. Consider the following program:

$$symptom(b) \Rightarrow disease(a)$$
$$symptom(d), disease(c) \Rightarrow symptom(b)$$

If we pose the query

$$symptom(b) \vdash disease(x)$$

then, using the $(\vdash P)$ schema, we obtain

$$symptom(b) \vdash symptom(b)$$

binding x to a. This means that by observing the symptom b we know by means of the first program rule that a is a possible disease.

If, on the other hand, we pose the query

$$symptom(x) \vdash disease(c)$$

any attempt to use the $(\vdash P)$-schema will fail, but using $(P\vdash)$ we obtain

$$symptom(d), disease(c) \vdash disease(c)$$

binding x to b. This means that the symptom to look for to detect the disease c is the symptom b.

The second example gives a demonstration of the use of implications in the body of clauses:

$$\Rightarrow even(0)$$
$$odd(x) \Rightarrow even(s(x))$$
$$even(x) \rightarrow \bot \Rightarrow odd(x)$$

Consider the query

$$even(s(0)) \vdash \bot$$

Then, using $(P\vdash)$, we obtain

$$odd(0) \vdash \bot$$

Using $(P\vdash)$ once again we obtain

$$even(0) \rightarrow \bot \vdash \bot$$

Splitting the implication on the left we get

$$\vdash even(0)$$

and

$$\bot \vdash \bot$$

both of which succeed. So the answer is 'yes'.

The final example concerns function definition (see [1], [2], [4]). Consider the program

$$0 \Rightarrow min(x, 0)$$
$$0 \Rightarrow min(0, x)$$
$$s(min(x, y)) \Rightarrow min(s(x), s(y))$$
$$(min(x, y) \rightarrow z) \rightarrow s(z) \Rightarrow s(min(x, y))$$

The first three clauses of the program can be read as an equational definition of the *min*-function, the last one as a scheme for computation inside the successor constructor. It may be read as: '$s(z)$ is the value of $s(min(x, y))$, if $min(x, y)$ computes to z'. Of course, in the context of function definition, the double arrow '$\Rightarrow$ for 'is the value of' is somewhat awkward. By reversing sides of rules and replacing '$\Rightarrow$' with '=', one would obtain a more standard formulation.

By asking queries like

$$min(n, m) \vdash x$$

we can now compute the value of $min(n, m)$. Such computations rely heavily on the $(P\vdash)$ schema, which takes care of all the basic computation steps, as the following example of a linear derivation computing $min(2, 1)$ shows:

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{}{\langle \varnothing, \varnothing \rangle}}{\langle \{1 \vdash x\}, \{x/1\} \rangle}}{\langle \{0 \vdash z, s(z) \vdash x\}, \{z/0\}\{x/1\} \rangle}}{\langle \{min(1, 0) \vdash z, s(z) \vdash x\}, \{z/0\}\{x/1\} \rangle}}{\langle \{\vdash min(1, 0) \rightarrow z, s(z) \vdash x\}, \{z/0\}\{x/1\} \rangle}}{\langle \{(min(1, 0) \rightarrow z) \rightarrow s(z) \vdash x\}, \{z/0\}\{x/1\} \rangle}}{\langle \{s(min(1, 0)) \vdash x\}, \{z/0\}\{x/1\} \rangle}}{\langle \{min(2, 1) \vdash x\}, \{z/0\}\{x/1\} \rangle}$$

Here '1' stands for '$s(0)$' and '2' for '$s(s(0))$'. The substitution $\{z/0\}\{x/1\}$ is computed by the linear derivation, where the part $\{z/0\}$ is only internally used and $\{x/1\}$ answers the query. (The internal substitutions obtained by unifying antecedents with heads of program clauses in applications of $(P\vdash)_L$ are not shown.)

Two logical points must be mentioned. Firstly, the last clause of the program should properly be formulated with universal quantification over

the body in a way like

$$\Pi_z((min(x, y) \rightarrow z) \rightarrow s(z)) \Rightarrow s(min(x, y))$$

However, this poses no specific problem, since this clause is only used in connection with $(P\vdash)$, and evaluation of universal quantification on the left side of sequents can just be carried out by omitting the universal quantifier, as implicitly done in the example derivation. Secondly, the logic of the intended computations here cannot allow for contraction. (This also is implicitly followed in the example derivation.) So, writing functional programs in the present context gives a good example of the necessity of analysing substructural parts of the logic considered here.

Obviously, this example of function definition also provides new aspects for the issue of logic programming versus functional programming.

Acknowledgments

Completion of Part II was supported by DFG-grant Schr 275/6-1 to Peter Schroeder-Heister. We should like to thank Venkat Ajjanagadde for helpful comments.

References

[1] M. Aronsson, L.-H. Eriksson, A. Gäredal, L. Hallnäs and P. Olin (1990). The programming language GCLA: a definitional approach to logic programming, *New Generation Computing*, **4**, 381–404.

[2] M. Aronsson, L.-H. Eriksson, L. Hallnäs and P. Kreuger (1991). A survey of GCLA: a definitional approach to logic programming. In P. Schroeder-Heister, (ed.), *Extensions of Logic Programming: International Workshop, Tübingen, FRG, December 1989, Proceedings*. Springer Lecture Notes in Artificial Intelligence, 475: Berlin.

[3] M. Beeson (1991). Some applications of Gentzen's proof theory in automated deduction. In P. Schroeder-Heister, (ed.), *Extensions of Logic Programming: International Workshop, Tübingen, FRG, December 1989, Proceedings*, Springer Lecture Notes in Artificial Intelligence, 475: Berlin.

[4] D. Fredholm (1990). *On function definitions I. Basic notions and primitive recursive function definitions*. Lic. thesis, Department of Computer Science, University of Göteborg and Chalmers University of Technology.

[5] D. M. Gabbay (1985). N-PROLOG: an extenstion of PROLOG with hypothetical implication. II. Logical fundations, and negation as failure, *Journal of Logic Programming*, **2**, 251–283.

[6] D. M. Gabbay and U. Reyle (1984). N-PROLOG: an extension of PROLOG with hypothetical implications. I., *Journal of Logic Programming*, **1**, 319–355.

[7] G. Gentzen (1935). Untersuchungen über das logische Schließen, *Mathematische Zeitschrift*, **39**, 176–210, 405–431.

[8] L. Hallnäs (1987a). A note on the logic of a logic program. In *Proceedings of the Workshop on Programming Logic*, PMG-R 37, University of Göteborg and Chalmers University of Technology.

[9] L. Hallnäs (1987b). A note on non-monotonic reasoning. In *Proceedings of the 1987 Workshop on the Frame Problem*, Morgan Kaufmann Publ., Los Altos.

[10] L. Hallnäs (1991). Partial inductive definitions, *Theoretical Computer Science* (in press).
[11] S. Kanger (1963). A simplified proof method for elementary logic. In P. Braffort and D. Hirschberg, (eds.), *Computer Programming and Formal Systems* (Amsterdam: North-Holland).
[12] K. Kunen (1987). Negation in logic programming, *Journal of Logic Programming*, **4**, 281–308.
[13] P. Lorenzen (1955). *Einführung in die operative Logik und Mathematik* (Berlin: Springer).
[14] P. Martin-Löf (1985). On the meanings of the logical constants and the justifications of the logical laws. In *Atti degli Incontri di Logica Matematica*, Vol. 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena.
[15] D. Miller (1990). Abstractions in Logic Programs. In P. Odifreddi, (ed.), *Logic and Computer Science* (New York: Academic Press).
[16] D. Prawitz (1974). On the idea of a general proof theory, *Synthese*, **27**, 63–77.
[17] D. Prawitz (1979). Proofs and the meaning and completeness of the logical constants. In J. Hintkka *et al.* (eds.), *Essays on Mathematical and Philosophical Logic* (Dordrecht: Reidel).
[18] P. Schroeder-Heister (1984a). A natural extension of natural deduction, *Journal of Symbolic Logic*, **49**, 1284–1300.
[19] P. Schroeder-Heister (1984b). Generalized rules for quantifiers and the completeness of the intuitionistic operators &, $\vee$, $\supset$, $\bot$, $\forall$, $\exists$. In W. Oberschelp *et al.*, (eds.) *Computation and Proof Theory, Proceedings of the Logic Colloquim held in Aachen, 1983. Part II.* Springer Lecture Notes in Mathematics, 1104: Berlin.
[20] P. Schroeder-Heister (1990). Hypothetical reasoning and definitional reflection in logic programming. In P. Schroeder-Heister, (ed.) *Extensions of Logic Programming: International Workshop, Tübingen, FRG, December 1989, Proceedings.* Springer Lecture Notes in Artificial Intelligence, 475: Berlin.
[21] P. Schroeder-Heister (1991). Structural frameworks, substructural logics, and the role of elimination inferences. In G. Huet and G. Plotkin, (eds.), *Logical Frameworks.* Cambridge University Press.
[22] P. Schroeder-Heister (1987). *Structural Frameworks with Higher-Level Rules*: *Proof-Theoretic Investigations.* Habilitationsschrift, Universität Konstanz.
[23] P. Suppes (1957). *Introduction to Logic* (London: Van Nostrand).
[24] R. Harper, F. Honsell and G. Plotkin (1987). A framework for defining logics. Proceedings of the 2nd Annual Symposium on Logic in Computer Science, Ithaca, N.Y. IEEE Computer Science Press, Washington, D.C., 1987, pp. 28–46.

Received October 1990