Joint NCC & IIS Bull., Comp. Science, 16 (2001), 75–97
© 2001 NCC Publisher

Rule-based versus procedure-based view of logic programming

Krzysztof R. Apt, Jan–Georg Smaus

Logic programming is a rule-based formalism: a program consists of a set of rules activated by an initial query. In contrast, imperative programming is explained by means of a procedure-based view: a program consists of a set of procedure declarations and an initial statement.

We clarify here to what procedure-based formalism logic programming corresponds. To this end we introduce a procedure-based view based on a subset of first-order logic and assign to it an operational semantics. We establish two results. First, we show that a syntactic transformation that forms part of Clark's completion of a logic program provides a translation from the rule-based view to the procedure-based view. Second, we show that the transformations in the spirit of Lloyd-Topor provide a translation from the procedure-based view to the rule-based view.

Finally, we discuss the consequences of adoption of each view for the design of logic programming languages.

1. Introduction

Logic programming is a *rule-based* formalism: a program consists of a collection of rules that are activated by means of an initial query. This approach to programming has to do with the automated theorem proving roots of logic programs. In fact, each rule or query is a Horn clause and the computation process corresponds to a specialised form of the resolution method.

In imperative programs, for example in Pascal or C, a different view is taken according to which a program consists of a collection of procedure declarations together with an initial statement. Each procedure declaration consists of a procedure name followed by (the specification of the types of) the formal parameters and the procedure body, which is a statement. In turn, the statements are defined by structural induction starting with the atomic statements and using compound statements. We call this approach a *procedure-based* formalism.

In this paper, to keep the arguments simple, we restrict our attention to logic programs without negation, usually called *definite programs.* We refer to them simply as "logic programs". Our aim is to clarify to what procedure-based formalism logic programming corresponds. Since syntactically logic programs are defined in a subset of first-order logic —the one consisting of the Horn clauses— it is possible to define syntactically the appropriate procedure-based formalism as a subset of first-order logic, as well. More specifically, we introduce a procedure-based formalism built from a subset $\mathcal{L}_0$ of first-order logic and show that it is *structurally equivalent* to the rule-based formalism of logic programming. $\mathcal{L}_0$ is obtained by allowing equality, disjunction, conjunction, and the existential quantifier.

To assign a computational meaning to the formulas in $\mathcal{L}_0$, a simple operational semantics is used. It employs the following correspondence between the logical and the programming language semantics concepts:

substitution	state
composition of substitutions	state update

Operationally, the logical constructs correspond to the following constructs of imperative programming:

equality	unify operation
disjunction	nondeterministic choice
conjunction	sequencing (";")
existential quantifier	declaration of a local variable

So instead of substitutions and logical constructs, we could have used states and a programming language notation, as has been done by de Bakker [4] in a paper with the telling running title "Logic programming without logic". However, we chose to define the procedure-based formalism in terms of logical constructs because it gives us a natural notion of *correctness* of the computed result. To be specific, given an original statement (which is a formula) ϕ, the computed answer substitution θ validates ϕ, i.e., $\phi\theta$ is a logical consequence of the set of procedure declarations (which is also a formula). This will be stated and proved in the Soundness Corollary 5.5. A reverse relation will be established in the Completeness Corollary 5.6.

Another reason why we defined this procedure-based formalism in terms of logical constructs was our desire to clarify the relation between logic programming and the declarative subset of the implemented imperative programming language Alma-0 of Apt et al. [2]. The precise comparison is made in Section 6.

Here let us just mention that Alma-0 extends a subset of Modula-2 by a small number of features inspired by the logic programming paradigm. These additions encourage declarative programming and lead to a different programming style than Prolog in its stress on arrays and iteration instead of on lists and recursion. Such programs are often easier to write and to verify than corresponding Prolog programs, see, e.g., Apt and Bezem [1].

Structural equivalence between the two formalisms means that the correspondence in each direction can be obtained by means of simple syntactic transformations. For example, the logic program

```
num(0) ← ,
num(s(x)) ← num(x)
```

defining the property of being a numeral can be transformed into the procedure declaration

$$num(x) \leftarrow (x = 0) \vee \exists y\, (x = s(y) \wedge num(y)). \qquad (1)$$

Such a transformation reveals the constructs implicitly present in logic programs and testifies to the remarkable conciseness of the rule-based formalism. Its general form is known both to researchers on logic programming theory and to Prolog programmers (who would use fresh variables instead of existentially quantified variables). It is part of *Clark's completion* which was introduced by Clark [5] to study the meaning of logic programming in presence of negation.

In this paper we provide a formal proof that this transformation from the rule-based to the procedure-based formalism preserves the operational semantics. We also present a translation in the other direction, from the procedure-based view defined w.r.t. $\mathcal{L}_0$ into the rule-based view, that preserves the operational semantics, as well. It is obtained by transformation rules in the style of Lloyd and Topor [9] that were studied there only from the point of view of declarative semantics.

The paper is organised as follows. In the next section we introduce the fragment $\mathcal{L}_0$ of first-order logic that we consider here and recall the syntax and the operational semantics (i.e., procedural interpretation) of logic programs. Then, in Section 3 we introduce the operational semantics of the formulas of $\mathcal{L}_0$. In Sections 4 and 5 we establish the abovementioned results. Then in Section 6 we discuss related work. In the final section we review the consequences of adoption of each view and draw some conclusions about the design of languages that support logic programming.

2. Preliminaries

2.1. Miscellaneous

To simplify the notation, we write a vector of syntactic objects $o_1, \ldots, o_n$ (variables, terms, ...) as $\mathbf{o}$. Occasionally we identify such a vector with a set. The set of variables in a syntactic object o is denoted by $Var(o)$ and the set of free variables by $free(o)$.

The empty substitution is denoted by ϵ. The restriction of the domain of a substitution τ to a set of variables V is denoted by $\tau \restriction_V$. The set of substitutions is denoted by $Subs$.

2.2. The Language $\mathcal{L}_0$

We define a subset $\mathcal{L}_0$ of first-order logic. The relation symbols of $\mathcal{L}_0$ are the equality $=$ and the propositional constants **true** and **false**, and relation symbols $p_1, \ldots, p_k$ that we call *procedures*. Further, we allow in $\mathcal{L}_0$ disjunction, conjunction, and the existential quantifier. Given $i \in [1..k]$, by a *procedure declaration of* p_i we mean a formula $p_i(\mathbf{x}_i) \leftarrow \psi_i$, where ψ_i is a formula in $\mathcal{L}_0$ and $free(\psi_i) \subseteq \mathbf{x}_i$.

A *declaration set* of $p_1, \ldots, p_k$, denoted by $\mathcal{D}$, is simply a set of procedure declarations, one for each p_i. We call an atom $p_i(\mathbf{t}_i)$ a *procedure call.* Note that the procedure declarations are not formulas of $\mathcal{L}_0$, though of course they are formulas of the full first-order language extending $\mathcal{L}_0$. Since each p_j can occur in ψ_i, a declaration set models a set of mutually recursive procedures.

An example of a procedure declaration is given in (1).

In Section 3 we will assign an operational semantics to formulas of $\mathcal{L}_0$ in presence of a declaration set.

2.3. Logic Programs

We assume that logic programs are defined in the language $\mathcal{L}_0$. Therefore logic programs (clause bodies) may contain atoms of the form $s = t$, as well as the atoms **true** and **false**. In the context of logic programming, we call those atoms *built-ins.* We assume without explicit mention that any logic program that uses built-ins contains the following two clauses: **true** $\leftarrow$ and $x = x \leftarrow$. Moreover, there are no other clauses with a built-in in the head.

In correspondence to the semantics of $\mathcal{L}_0$ to be defined in Section 3, we define the operational semantics of logic programs based on SLD-resolution with the leftmost selection rule, usually called *LD-resolution.* We consider pairs of the form $\langle Q\ ;\ \theta\rangle$, where Q is a query and θ is a substitution. Intuitively, $\langle Q\ ;\ \theta\rangle$ denotes the query Q in the "state" θ, that is, the variables of Q should be "interpreted" as in θ. We do not identify $\langle Q\ ;\ \theta\rangle$ with $Q\theta$ for technical reasons that have to do with the usual standardisation apart condition [8]. It suffices to say that $Q\theta$ can be uniquely obtained from $\langle Q\ ;\ \theta\rangle$ but not vice versa.

Fix a logic program P. Consider a pair $\langle B, \mathbf{C}\ ;\ \theta\rangle$ and a clause $c = H \leftarrow \mathbf{B}$ from P. Suppose $Var(c) = \mathbf{y}$ and let $\rho := \{\mathbf{y}/\mathbf{z}\}$, where $\mathbf{z}$ is a vector of fresh variables, and suppose that η is an mgu of $H\rho$ and $B\theta$. We then call the pair $\langle \mathbf{B}\rho, \mathbf{C}\ ;\ \theta\eta\rangle$ an *LD-resolvent of* $\langle B, \mathbf{C}\ ;\ \theta\rangle$ *and* $c\rho$ and write

$$\langle B, \mathbf{C}\ ;\ \theta\rangle \leadsto \langle \mathbf{B}\rho, \mathbf{C}\ ;\ \theta\eta\rangle.$$

Since we assume that any logic program contains the clauses **true** $\leftarrow$ and $x = x \leftarrow$, it turns out that if B is a built-in, we obtain

$$\langle \mathbf{true}, \mathbf{C} \ ; \ \theta\rangle \rightsquigarrow \langle \mathbf{C} \ ; \ \theta\rangle \tag{2}$$

or

$$\langle s = t, \mathbf{C} \ ; \ \theta\rangle \rightsquigarrow \langle \mathbf{C} \ ; \ \theta\eta\rangle, \tag{3}$$

where η is an mgu of s and t.

Note that we make the renaming of variables in the program clause explicit, in contrast to the usual style of definition where one just says "a renamed variant of c is used." This leads to a more natural correspondence with our operational semantics of $\mathcal{L}_0$.

Consider now a program P and a query Q. By an *LD-derivation* of Q w.r.t. P we mean a maximal sequence of pairs $\langle Q_i \ ; \ \theta_i\rangle$ such that $\langle Q_0 \ ; \ \theta_0\rangle = \langle Q \ ; \ \epsilon\rangle$ and for $i \geq 0$ we have $\langle Q_i \ ; \ \theta_i\rangle \rightsquigarrow \langle Q_{i+1} \ ; \ \theta_{i+1}\rangle$. If

$$\langle Q \ ; \ \epsilon\rangle \rightsquigarrow^* \langle \square \ ; \ \tau\rangle,$$

where "$\rightsquigarrow^*$" is the reflexive, transitive closure of "$\rightsquigarrow$" (so "$\rightsquigarrow^*$" is the outcome of iterating "$\rightsquigarrow$" zero or more times), then we say that the query Q has been *proven* with the answer $\tau\restriction_{Var(Q)}$, and call $\tau\restriction_{Var(Q)}$ a *computed answer substitution.*

The operational semantics, usually called *procedural interpretation*, of a logic program P is defined as

$$\mathcal{LD}(P\,;\,Q) := \{\tau\restriction_{Var(Q)} | \ \langle Q \ ; \ \epsilon\rangle \rightsquigarrow^* \langle \square \ ; \ \tau\rangle\}.$$

3. Operational Semantics for $\mathcal{L}_0$-Formulas

We define an operational semantics for the formulas of $\mathcal{L}_0$ in presence of a declaration set $\mathcal{D}$. It is defined by a transition relation $\rightarrow$ between configurations, where a configuration is a pair $\langle \phi \ ; \ \theta\rangle$. Intuitively, ϕ is a program executed in the state θ. Here ϕ is a formula of $\mathcal{L}_0$ or the empty formula $\square$, and θ is a substitution. The empty formula is introduced for notational and terminological convenience and is not a formula of $\mathcal{L}_0$. In what follows we identify $\square \wedge \phi$ or $\phi \wedge \square$ with ϕ. This allows us, for example, to say that the atomic formula $p(\mathbf{t})$ is of the form $\phi_1 \wedge \phi_2$: simply put $\phi_1 \equiv \square$ and $\phi_2 \equiv p(\mathbf{t})$.

Definition 3.1. Consider $\mathcal{L}_0$ and a declaration set $\mathcal{D}$ of its procedures. The *transition relation* $\rightarrow$ between configurations is defined by the following rule and axioms:

1. $\langle \mathbf{true} \ ; \ \theta\rangle \rightarrow \langle \square \ ; \ \theta\rangle$,
2. $\langle s = t \ ; \ \theta\rangle \rightarrow \langle \square \ ; \ \theta\eta\rangle$ if $s\theta$ and $t\theta$ unify and η is an mgu of them,

3. $\dfrac{\langle \phi_1 \ ; \ \theta \rangle \rightarrow \langle \phi_1' \ ; \ \tau \rangle}{\langle \phi_1 \wedge \phi_2 \ ; \ \theta \rangle \rightarrow \langle \phi_1' \wedge \phi_2 \ ; \ \tau \rangle}$,
4. $\langle \phi_1 \vee \phi_2 \ ; \ \theta \rangle \rightarrow \langle \phi_1 \ ; \ \theta \rangle$,
5. $\langle \phi_1 \vee \phi_2 \ ; \ \theta \rangle \rightarrow \langle \phi_2 \ ; \ \theta \rangle$,
6. $\langle \exists x \, \phi \ ; \ \theta \rangle \rightarrow \langle \phi\{x/y\} \ ; \ \theta \rangle$, where the variable y occurs neither in $\exists x \, \phi$ nor in θ,
7. $\langle p(\mathbf{t}) \ ; \ \theta \rangle \rightarrow \langle \psi\{\mathbf{x}/\mathbf{t}\} \ ; \ \theta \rangle$, where $p(\mathbf{x}) \leftarrow \psi \in \mathcal{D}$.

Axiom 1 states that **true** corresponds to the customary *skip* (or empty) statement. Axiom 2 states that the equality "=" is interpreted as "is unifiable with" relation. Rule 3 coincides with the usual rule for the sequential composition ";" in imperative languages, see, e.g., rule $comp^1_{sos}$ of Nielson and Nielson [13, page 33]. Axioms 4 and 5 state that the disjunction is interpreted as a nondeterministic choice. Axiom 6 states that the existential quantifier is interpreted as a declaration of a local variable. Finally, axiom 7 states that procedure calls are interpreted according to the *copy rule*, also sometimes called *backchaining*.

The operational semantics collects the results of all transition sequences:

Definition 3.2. The *operational semantics of a first-order formula* ϕ is a mapping $\mathcal{OP}(.\,;\,.)$ defined by

$$\mathcal{OP}(\mathcal{D}\,;\,\phi) := \{\tau\lceil_{free(\phi)} \mid \langle \phi \ ; \ \epsilon \rangle \rightarrow^* \langle \Box \ ; \ \tau \rangle, \ \tau \in Subs\},$$

where $\rightarrow^*$ is the transitive reflexive closure of $\rightarrow$.

Note that since $\mathcal{OP}(.\,;\,.)$ only records the outcomes of "successful" computations, we disregard the possibility of failures. A failure arises when a configuration of the form $\langle \mathbf{false} \wedge \phi \ ; \ \theta \rangle$ or of the form $\langle s = t \wedge \phi \ ; \ \theta \rangle$, where $s\theta$ and $t\theta$ do not unify, is reached. Such configurations have no successor.

4. Embedding Result

In this section, we show how a logic program can be transformed into an operationally equivalent $\mathcal{L}_0$-formula. The transformation is part of Clark's completion [5].

Definition 4.1. Let P be a logic program. We fix a sequence $x_1, \ldots, x_n$ of variables that are fresh w.r.t. P, where n is the highest arity of any predicate occurring in P. The declaration set $IF(P)$ is obtained by performing successively the following steps:

Step 1: Transform each clause $p(\mathbf{t}) \leftarrow B_1, \ldots, B_k$ of P, where $p(\mathbf{t})$ is not a built-in, into $p(\mathbf{x}) \leftarrow \mathbf{x} = \mathbf{t} \wedge B_1 \wedge \ldots \wedge B_k$, where $\mathbf{x} = x_1, \ldots, x_{n'}$, n' being the arity of p.

Step 2: Transform each formula $p(\mathbf{x}) \leftarrow F$ obtained from a clause c in the previous step into $p(\mathbf{x}) \leftarrow \exists \mathbf{y} F$, where $\mathbf{y} = \mathit{Var}(c)$.

Step 3: Let $p(\mathbf{x}) \leftarrow G_1, \ldots, p(\mathbf{x}) \leftarrow G_k$ be all formulas obtained in the previous step with a predicate p on the left-hand side. Replace them by one formula $p(\mathbf{x}) \leftarrow G_1 \vee \ldots \vee G_k$. In the special case that $k > 0$ and all G_i are empty, replace $G_1 \vee \ldots \vee G_k$ by **true** (since it is a non-empty disjunction of empty conjunctions). In the special case that $k = 0$, replace $G_1 \vee \ldots \vee G_k$ by **false** (since it is the empty disjunction).

Clark's completion of a logic program P is defined by replacing in $\mathit{IF}(P)$ each reversed implication $\leftarrow$ by the equivalence $\leftrightarrow$ (the result is usually denoted by $\mathit{IFF}(P)$), and by adding so-called Clark's Equality Theory CET. Since CET plays no role in our considerations, we omit its treatment.

Note that $\mathit{free}(p(\mathbf{x}) \leftarrow G) = \mathbf{x}$ for each $p(\mathbf{x}) \leftarrow G \in \mathit{IF}(P)$. Thus each element of $\mathit{IF}(P)$ is indeed a procedure declaration as defined in Subsection 2.2.

For technical reasons that become apparent in the proof of Lemma 4.4, it is convenient to be able to distinguish equalities introduced in $\mathit{IF}(P)$ from any equalities that may have already been present in clause bodies of the original program. For this purpose, we assume from now on that each clause has the form $H \leftarrow \mathbf{true} \wedge B_1 \wedge \ldots \wedge B_k$, with the clause $\mathbf{true} \leftarrow$ as only exception. The only purpose of the atom **true** is to mark the separation between the introduced equalities and the ones already present in the original program. This assumption is no loss of generality as it is easy to see that the operational semantics $\mathcal{LD}$ of a logic program is not affected by such additions of the atom **true**.

The following lemma states that an LD-derivation step can be faithfully simulated as a sequence of transitions.

Lemma 4.2. Let P be a logic program and A a one-atom query. If $\langle A \; ; \; \theta \rangle \rightsquigarrow \langle (B_1, \ldots, B_k)\{\mathbf{y}/\mathbf{z}\} \; ; \; \theta\eta \rangle$ is an LD-derivation step using the clause $c = H \leftarrow B_1, \ldots, B_k$ of P, then $\langle A \; ; \; \theta \rangle \rightarrow^* \langle (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{z}\} \; ; \; \theta\eta \rangle$ w.r.t. the declaration set $\mathit{IF}(P)$.

Proof. To deal with the case when A is a built-in it suffices to compare (2) to axiom 1 and (3) to axiom 2. Suppose now that A is not a built-in.

Recall that $\mathbf{y} = \mathit{Var}(c)$ and $\mathbf{z}$ is a vector of fresh variables, and that

$$\eta \text{ is an mgu of } H\{\mathbf{y}/\mathbf{z}\} \text{ and } A\theta. \tag{4}$$

Suppose $A \equiv p(\mathbf{s})$ and $H \equiv p(\mathbf{t})$, and the procedure declaration of p in $\mathit{IF}(P)$ is $p(\mathbf{x}) \leftarrow (G_1 \vee \ldots \vee G_l)$. By definition of $\mathit{IF}(P)$, for some $i \in [1..l]$,

$$G_i \equiv \exists \mathbf{y}.\, \mathbf{x} = \mathbf{t} \wedge B_1 \wedge \ldots \wedge B_k.$$

To simplify the notation, we assume without loss of generality that $i = 1$. By axiom 7, we have

$$\langle A\ ;\ \theta\rangle \to \langle((\exists \mathbf{y}.\, \mathbf{x} = \mathbf{t} \wedge B_1 \wedge \ldots \wedge B_k) \vee G_2 \vee \ldots \vee G_l)\{\mathbf{x}/\mathbf{s}\}\ ;\ \theta\rangle. \qquad (5)$$

Let $\mathbf{y}'$ be the vector of variables that is identical to $\mathbf{y}$ except that any variables in $\mathit{Var}(\mathbf{s})$ are renamed using fresh variables. By the definition of the application of a substitution, noting that $\mathbf{x} \cap \mathbf{y} = \emptyset$, we have

$$(\exists \mathbf{y}.\, \mathbf{x} = \mathbf{t} \wedge B_1 \wedge \ldots \wedge B_k)\{\mathbf{x}/\mathbf{s}\} \equiv \exists \mathbf{y}'.\, \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{y}'\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{y}'\}$$

and hence (5) is in fact identical to

$$\begin{array}{l}\langle A\ ;\ \theta\rangle \to \langle((\exists \mathbf{y}'.\, \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{y}'\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{y}'\}) \vee \\ \qquad (G_2 \vee \ldots \vee G_l)\{\mathbf{x}/\mathbf{s}\})\ ;\ \theta\rangle.\end{array} \qquad (6)$$

We now first consider transition sequences for

$$\langle(\exists \mathbf{y}'.\, \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{y}'\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{y}'\})\ ;\ \theta\rangle.$$

By repeated application of axiom 6 we have, noting that $\mathbf{y}'$ was defined so that $\mathit{Var}(\mathbf{s}) \cap \mathbf{y}' = \emptyset$,

$$\begin{array}{l}\langle(\exists \mathbf{y}'.\, \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{y}'\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{y}'\})\ ;\ \theta\rangle \to^* \\ \quad \langle \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{y}'\}\{\mathbf{y}'/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{y}'\}\{\mathbf{y}'/\mathbf{z}\}\ ;\ \theta\rangle \equiv \\ \quad \langle \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{z}\}\ ;\ \theta\rangle.\end{array} \qquad (7)$$

Here we mean the very same $\mathbf{z}$ that occurs in the statement of this lemma. Note that the variables in $\mathbf{z}$ are fresh w.r.t. this transition sequence.

We have

$$p(\mathbf{t})\{\mathbf{y}/\mathbf{z}\}\theta = p(\mathbf{t})\{\mathbf{y}/\mathbf{z}\} \qquad (8)$$

by the freshness of $\mathbf{z}$ and the fact that $\mathit{Var}(p(\mathbf{t})) \subseteq \mathbf{y}$ (since $\mathbf{y} = \mathit{Var}(c)$).

Continuing the transition sequence (7), because of (4,8) we get by repeated application of axiom 2

$$\langle \mathbf{s} = (\mathbf{t}\{\mathbf{y}/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{z}\}\ ;\ \theta\rangle \to^* \langle(B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{z}\}\ ;\ \theta\eta\rangle. \qquad (9)$$

Using axiom 4, we conclude from (6), (7), (9) that

$$\langle A\ ;\ \theta\rangle \rightarrow^{*} \langle (B_1 \wedge \ldots \wedge B_k)\{\mathbf{y}/\mathbf{z}\}\ ;\ \theta\eta\rangle.$$

□

The next lemma states that certain sequences of transitions (Def. 3.1) can be faithfully simulated by an LD-derivation step. Before we formulate it, we first give a name to non-empty transition sequences (denoted by $\rightarrow^{+}$) that correspond to exactly one derivation step. This definition relies on the fact that it is assumed that clause bodies start with **true**.

Definition 4.3. We call a transition sequence $\langle A\ ;\ \theta\rangle \rightarrow^{+}\langle B_1 \wedge \ldots \wedge B_k\ ;\ \theta\eta\rangle$ *good w.r.t. a declaration set* $\mathcal{D}$ if $A, B_1, \ldots, B_k$ are all atoms, $B_1 \equiv \mathbf{true}$ and for each proper prefix $\langle A\ ;\ \theta\rangle \rightarrow^{+}\langle \phi\ ;\ \theta\gamma\rangle$ of it, $\phi \not\equiv \mathbf{true} \wedge \ldots$.

Note in particular the boundary case when $k = 1$, in which $B_2 \wedge \ldots \wedge B_k$ is the empty formula.

Lemma 4.4. Consider a logic program P and an atom A. If

$$\langle A\ ;\ \theta\rangle \rightarrow^{+}\langle B'_1 \wedge \ldots \wedge B'_k\ ;\ \theta\eta\rangle$$

is a good transition sequence w.r.t. $IF(P)$, then there exists a clause $c = H \leftarrow B_1, \ldots, B_k$ in P where $\mathbf{y} = Var(c)$ and $(B_1, \ldots, B_k)\{\mathbf{y}/\mathbf{z}\} \equiv B'_1, \ldots, B'_k$, and

$$\langle A\ ;\ \theta\rangle \rightsquigarrow \langle (B_1, \ldots, B_k)\{\mathbf{y}/\mathbf{z}\}\ ;\ \theta\eta\rangle$$

is an LD-derivation step using c.

Proof. To deal with the case when A is a built-in again it suffices to compare axiom 1 to (2) and axiom 2 to (3). Suppose now that A is not a built-in.

By Def. 3.1, in particular axiom 7, there is exactly one possible transition for $\langle A\ ;\ \theta\rangle$, namely

$$\langle A\ ;\ \theta\rangle \rightarrow \langle (G_1 \vee \ldots \vee G_l)\{\mathbf{x}/\mathbf{t}\}\ ;\ \theta\rangle, \tag{10}$$

where $A \equiv p(\mathbf{t})$ and $p(\mathbf{x}) \leftarrow (G_1 \vee \ldots \vee G_l)$ is the procedure declaration[1] of p in $IF(P)$. Thus by the hypothesis in the statement of the lemma

$$\langle (G_1 \vee \ldots \vee G_l)\{\mathbf{x}/\mathbf{t}\}\ ;\ \theta\rangle \rightarrow^{*} \langle B'_1 \wedge \ldots \wedge B'_k\ ;\ \theta\eta\rangle. \tag{11}$$

By possibly multiple applications of axioms 4 or 5, the existence of (11) implies that for some $i \in [1..l]$,

[1]It is easy to see that the procedure declaration cannot be $p(\mathbf{x}) \leftarrow \mathbf{false}$, since that would contradict the existence of a derivation $\langle A\ ;\ \theta\rangle \rightarrow^{+} \langle (B'_1 \wedge \ldots \wedge B'_k)\ ;\ \theta\eta\rangle$.

$$\langle G_i\{\mathbf{x}/\mathbf{t}\} \ ; \ \theta\rangle \to^* \langle B_1' \wedge \ldots \wedge B_k' \ ; \ \theta\eta\rangle. \tag{12}$$

Suppose $G_i \equiv \exists \mathbf{w}.\ \mathbf{x} = \mathbf{s} \wedge B_1 \wedge \ldots \wedge B_m$. Since $IF(P)$ is defined in such a way that $Var(B_1, \ldots, B_m, \mathbf{s}) \cap \mathbf{x} = \emptyset$, we have

$$G_i\{\mathbf{x}/\mathbf{t}\} \equiv \exists \mathbf{w}'.\ \mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{w}'\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{w}'\}, \tag{13}$$

where $\mathbf{w}'$ is identical to $\mathbf{w}$ except that any variables also in $\mathbf{t}$ are renamed using fresh variables (this is the usual definition of application of a substitution). By repeated application of rule 3 and axiom 6, noting that $Var(\mathbf{t}) \cap Var(\mathbf{w}') = \emptyset$, we have

$$\begin{aligned}&\langle \exists \mathbf{w}'.\ \mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{w}'\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{w}'\} \ ; \ \theta\rangle \to^* \\ &\quad \langle (\mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{w}'\}\{\mathbf{w}'/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{w}'\}\{\mathbf{w}'/\mathbf{z}\} \ ; \ \theta\rangle \equiv \\ &\quad \langle (\mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{z}\} \ ; \ \theta\rangle.\end{aligned} \tag{14}$$

Here we mean the very $\mathbf{z}$ that occurs in the statement of this lemma. Note that the variables in $\mathbf{z}$ are fresh w.r.t. any transition sequence mentioned so far in this proof.

Moreover, by the form of the transition axioms and rules, *every* maximal transition sequence for $\langle \exists \mathbf{w}'.\ \mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{w}'\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{w}'\} \ ; \ \theta\rangle$ has (14) as a prefix, and so in particular the transition sequence (12). Thus

$$\langle \mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{z}\} \ ; \ \theta\rangle \to^* \langle (B_1' \wedge \ldots \wedge B_k') \ ; \ \theta\eta\rangle. \tag{15}$$

But by rule 3, (15) has the prefix

$$\langle \mathbf{t} = (\mathbf{s}\{\mathbf{w}/\mathbf{z}\}) \wedge (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{z}\} \ ; \ \theta\rangle \to^* \langle (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{z}\} \ ; \ \theta\eta'\rangle \tag{16}$$

for some η'. This implies by (10), (11), (12) that

$$\langle A \ ; \ \theta\rangle \to^* \langle (B_1 \wedge \ldots \wedge B_m)\{\mathbf{w}/\mathbf{z}\} \ ; \ \theta\eta'\rangle$$

is a prefix of

$$\langle A \ ; \ \theta\rangle \to^* \langle (B_1' \wedge \ldots \wedge B_k') \ ; \ \theta\eta\rangle.$$

Hence, by the assumed goodness of the second transition sequence it follows that $m = k$, $(B_1 \wedge \ldots \wedge B_k)\{\mathbf{w}/\mathbf{z}\} \equiv B_1' \wedge \ldots \wedge B_k'$, and $\theta\eta = \theta\eta'$. By axiom 2, (16) implies that η is an mgu of $\mathbf{t}\theta$ and $\mathbf{s}\{\mathbf{w}/\mathbf{z}\}\theta$. Moreover, by definition of $IF(P)$, $Var(\mathbf{s}) \subseteq \mathbf{w}$, and so by the freshness condition on $\mathbf{z}$, we have $\mathbf{s}\{\mathbf{w}/\mathbf{z}\}\theta \equiv \mathbf{s}\{\mathbf{w}/\mathbf{z}\}$. But by the definition of $IF(P)$, we have that $p(\mathbf{s}) \leftarrow B_1, \ldots, B_k$ is a clause in P whose variables are $\mathbf{w}$, and so by the definition of LD-resolution (η is an mgu of $\mathbf{s}\{\mathbf{w}/\mathbf{z}\}$ and $\mathbf{t}\theta$)

$$\langle A \ ; \ \theta\rangle \leadsto \langle (B_1, \ldots, B_k)\{\mathbf{w}/\mathbf{z}\} \ ; \ \theta\eta\rangle.$$

□

We now prove the following result on embedding of the operational semantics of logic programming into our operational semantics of $\mathcal{L}_0$. It is a proof-theoretic counterpart of the fact that P and $IF(P)$ are semantically equivalent, i.e., they have the same models (when equality is interpreted as identity).

Theorem 4.5 (Embedding). Consider a logic program P and a query $A_1, \ldots, A_n$. Then

$$\mathcal{LD}(P\,;\, A_1, \ldots, A_n) = \mathcal{OP}(IF(P)\,;\, A_1 \wedge \ldots \wedge A_n).$$

Proof. Since $A_1, \ldots, A_n$ contains no quantifiers, we have $free(A_1, \ldots, A_n) = Var(A_1, \ldots, A_n)$, and therefore it is sufficient to show that

$$\langle A_1, \ldots, A_n \; ; \; \theta\rangle \rightsquigarrow^* \langle \Box \; ; \; \tau\rangle \text{ iff } \langle A_1 \wedge \ldots \wedge A_n \; ; \; \theta\rangle \rightarrow^* \langle \Box \; ; \; \tau\rangle.$$

But this equivalence follows directly from Def. 3.1 and the definition of LD-derivations by repeated application of Lemmas 4.2 and 4.4. □

The above theorem states the equivalence of two semantics that were both defined in terms of *all* solutions to a query, respectively an initial formula. But clearly Lemmata 4.2 and 4.4 establish a more specific correspondence between each individual LD-derivation and some transition sequence, and vice versa.

When logic programs are viewed as Prolog programs, the derivations are enumerated by trying the clauses in order of their textual appearance in the program. This corresponds to a depth-first left-to-right traversal of the LD-tree. It is straightforward to define the corresponding enumeration order for transition sequences, i.e., paths of a disjunction are tried from left to right. Using this more specific correspondence one can then show that in both formalisms, the solutions are enumerated in the same order. That is, both formalisms closely correspond when implemented by means of the depth-first left-to-right search. This has implications for (existential) termination, to give an example.

5. Reduction Result

In this section, we show how an $\mathcal{L}_0$-formula can be transformed into an operationally equivalent logic program. The transformation is in the style of Lloyd and Topor [9].

Consider a declaration of a procedure p. It has one of the following two forms:

$$p(\mathbf{x}) \leftarrow \mathbf{false}$$

or

$$p(\mathbf{x}) \leftarrow \exists \mathbf{y}_1 \phi_1 \vee \ldots \vee \exists \mathbf{y}_n \phi_n, \tag{17}$$

where $n \geq 1$, each $\mathbf{y}_i$ is a possibly empty vector of variables such that $\mathbf{y}_i \cap \mathbf{x} = \emptyset$, and each ϕ_i is a conjunction (possibly consisting of just one conjunct).

We now define two transformation rules that preserve the semantics. Here ψ and φ can be empty:

Disjunction Replace each $\exists \mathbf{y}_i \phi_i$ of the form $\exists \mathbf{y}_i(\psi \wedge (\chi_1 \vee \chi_2) \wedge \varphi)$ by $\exists \mathbf{y}_i(\psi \wedge \chi_1 \wedge \varphi) \vee \exists \mathbf{y}_i(\psi \wedge \chi_2 \wedge \varphi)$.

Quantification Replace each $\exists \mathbf{y}_i \phi_i$ of the form $\exists \mathbf{y}_i(\psi \wedge \exists x \chi \wedge \varphi)$ by $\exists \mathbf{y}_i \exists z(\psi \wedge \chi\{x/z\} \wedge \varphi)$, where z is a variable that does not occur in ϕ_i and $z \notin \mathbf{x}$.

Lemma 5.1. The disjunction and quantification rules preserve the operational semantics $\mathcal{OP}(\,;\,)$ of a declaration set.

Proof. Let $\mathcal{D}$ be a declaration set and $\mathcal{D}_d$ (respectively $\mathcal{D}_q$) be obtained from $\mathcal{D}$ by applying the disjunction (respectively quantification) rule. By Def. 3.1 it suffices to establish the following two claims.

1. A computation w.r.t. $\mathcal{D}$ is of the form

   $$\langle \phi_0 \,;\, \epsilon \rangle \rightarrow^* \langle p(\mathbf{t}) \wedge \phi \,;\, \theta \rangle \rightarrow^* \langle \psi \wedge (\chi_1 \vee \chi_2) \wedge \varphi \wedge \phi \,;\, \eta \rangle \rightarrow^* \langle \Box \,;\, \tau \rangle,$$

   if and only if a computation w.r.t. $\mathcal{D}_d$ is of the form

   $$\langle \phi_0 \,;\, \epsilon \rangle \rightarrow^* \langle p(\mathbf{t}) \wedge \phi \,;\, \theta \rangle \rightarrow^* \langle \psi \wedge \chi_1 \wedge \varphi \wedge \phi \,;\, \eta \rangle \rightarrow^* \langle \Box \,;\, \tau \rangle,$$

   or of the form

   $$\langle \phi_0 \,;\, \epsilon \rangle \rightarrow^* \langle p(\mathbf{t}) \wedge \phi \,;\, \theta \rangle \rightarrow^* \langle \psi \wedge \chi_2 \wedge \varphi \wedge \phi \,;\, \eta \rangle \rightarrow^* \langle \Box \,;\, \tau \rangle.$$

2. A computation w.r.t. $\mathcal{D}$ is of the form

   $$\langle \phi_0 \,;\, \epsilon \rangle \rightarrow^* \langle p(\mathbf{t}) \wedge \phi \,;\, \theta \rangle \rightarrow^* \langle \psi \wedge \exists x \chi \wedge \varphi \wedge \phi \,;\, \eta \rangle \rightarrow^* \langle \Box \,;\, \tau \rangle$$

   if and only if a computation w.r.t. $\mathcal{D}_q$ is of the form

   $$\langle \phi_0 \,;\, \epsilon \rangle \rightarrow^* \langle p(\mathbf{t}) \wedge \phi \,;\, \theta \rangle \rightarrow^* \langle \exists z(\psi \wedge \chi\{x/z\} \wedge \varphi) \wedge \phi \,;\, \eta \rangle \rightarrow^* \langle \Box \,;\, \tau \rangle,$$

   where z is a variable that does not occur in the configuration $\langle \psi \wedge \exists x \chi \wedge \varphi \wedge \phi \,;\, \eta \rangle$.

Now, claim 1 is a direct consequence of the fact that only axioms 4 and 5 deal with disjunction. In turn, claim 2 is a direct consequence of the fact that the only axiom that deals with the existential quantification is axiom 6. □

By repeated use of these two transformation rules we can transform each procedure declaration (17) into the form

$$p(\mathbf{x}) \leftarrow \exists \mathbf{z}_1 \psi_1 \vee \ldots \vee \exists \mathbf{z}_m \psi_m, \tag{18}$$

where $m \geq 1$, $\mathbf{z}_i$ is a possibly empty vector of variables such that $\mathbf{z}_i \cap \mathbf{x} = \emptyset$, and each ψ_i is a conjunction of atoms.

Since each transformation rule preserves the set of free variables of the right-hand side of a procedure declaration, we have

$$\mathit{free}(\exists \mathbf{z}_1 \psi_1 \vee \ldots \vee \exists \mathbf{z}_m \psi_m) \subseteq \mathbf{x},$$

i.e., (18) is a procedure declaration, too.

Definition 5.2. Let $\mathcal{D}$ be a declaration set. We define the logic program $P(\mathcal{D})$ as follows. For each procedure declaration of the form (18) obtained by transforming $\mathcal{D}$, $P(\mathcal{D})$ contains the clauses

$$p(\mathbf{x}) \leftarrow \psi_1, \ldots, p(\mathbf{x}) \leftarrow \psi_m, \tag{19}$$

where we identified each conjunction of atoms with their sequence. In addition, $P(\mathcal{D})$ contains the clauses that define the built-ins (see Subsec. 2.3).

In the proof of the main result of this section we shall also use the following transformation rule, to be applied to a procedure declaration of the form (18).

Substitution Replace each formula of the form $\exists \mathbf{z}_i \psi_i$ by

$$\exists \mathbf{z} \exists \mathbf{z}_i (\mathbf{x} = \mathbf{z} \wedge \psi_i\{\mathbf{x}/\mathbf{z}\}),$$

where $\mathbf{z}$ is a sequence of fresh variables of the same length as $\mathbf{x}$.

Of course, the substitution rule could be used for an arbitrary formula ψ_i and not just a conjunction of atoms, but we shall not need such a general formulation. The following lemma states correctness of the rule.

Lemma 5.3. The substitution rule preserves the operational semantics $\mathcal{OP}(\,;\,)$ of a declaration set.

Proof. Let $\mathcal{D}$ be a declaration set and $\mathcal{D}_s$ be obtained from $\mathcal{D}$ by applying the substitution rule.

It suffices to note that by the equality axiom 2 and the transition axiom 6 for $\exists$ a computation w.r.t. $\mathcal{D}$ is of the form

$$\langle \phi_0 \ ; \ \epsilon \rangle \rightarrow^* \langle \exists \mathbf{z}_i \psi_i \wedge \phi \ ; \ \eta \rangle \rightarrow^* \langle \Box \ ; \ \tau \rangle,$$

if and only if a computation w.r.t. $\mathcal{D}_s$ is of the form

$$\langle \phi_0 \ ; \ \epsilon \rangle \rightarrow^* \langle \exists \mathbf{z} \exists \mathbf{z}_i (\mathbf{x} = \mathbf{z} \wedge \psi_i \{\mathbf{x}/\mathbf{z}\}) \wedge \phi \ ; \ \eta \rangle \rightarrow^* \langle \Box \ ; \ \tau \rangle.$$

□

The following result clarifies the relation between $\mathcal{D}$ and $P(\mathcal{D})$.

Theorem 5.4 (Reduction). Let $\mathcal{D}$ be a declaration set and $A_1, \ldots, A_n$ a query. Then

$$\mathcal{LD}(P(\mathcal{D}) \, ; A_1, \ldots, A_n) = \mathcal{OP}(\mathcal{D} ; A_1 \wedge \ldots \wedge A_n).$$

Proof. Thanks to Lemma 5.1 it suffices to prove the claim under the assumption that each procedure declaration in $\mathcal{D}$ is of the form (18). We compare such a declaration set $\mathcal{D}$ with $\mathit{IF}(P(\mathcal{D}))$.

In $\mathit{IF}(P(\mathcal{D}))$ the logic program (19) is represented by the procedure declaration

$$p(\mathbf{z}) \leftarrow \exists \mathbf{w}_1 (\mathbf{z} = \mathbf{x} \wedge \psi_1) \vee \ldots \vee \exists \mathbf{w}_m (\mathbf{z} = \mathbf{x} \wedge \psi_m),$$

where for $i \in [1..m]$, $\mathbf{w}_i = \mathit{free}(\psi_i) \cup \mathbf{x}$ and $\mathbf{z} \cap \mathbf{w}_i = \emptyset$. But (18) is a procedure declaration, so $\mathit{free}(\psi_i) \subseteq \mathbf{x} \cup \mathbf{z}_i$, i.e., $\mathbf{w}_i \subseteq \mathbf{x} \cup \mathbf{z}_i$. By possibly adding additional quantifiers we can thus assume that (19) is represented in $\mathit{IF}(P(\mathcal{D}))$ by the procedure declaration

$$p(\mathbf{z}) \leftarrow \exists \mathbf{x} \exists \mathbf{z}_1 (\mathbf{z} = \mathbf{x} \wedge \psi_1) \vee \ldots \vee \exists \mathbf{x} \exists \mathbf{z}_m (\mathbf{z} = \mathbf{x} \wedge \psi_m),$$

or, by reversing the roles of $\mathbf{x}$ and $\mathbf{z}$,

$$p(\mathbf{x}) \leftarrow \exists \mathbf{z} \exists \mathbf{z}_1 (\mathbf{x} = \mathbf{z} \wedge \psi_1 \{\mathbf{x}/\mathbf{z}\}) \vee \ldots \vee \exists \mathbf{z} \exists \mathbf{z}_m (\mathbf{x} = \mathbf{z} \wedge \psi_m \{\mathbf{x}/\mathbf{z}\}).$$

By Lemma 5.3 we now get

$$\mathcal{OP}(\mathit{IF}(P(\mathcal{D})) \, ; A_1 \wedge \ldots \wedge A_n) = \mathcal{OP}(\mathcal{D} ; A_1 \wedge \ldots \wedge A_n),$$

which by the Embedding Theorem 4.5 yields the claim. □

This yields the following conclusions.

Corollary 5.5 (Soundness). Consider a declaration set $\mathcal{D}$ and a formula ϕ of $\mathcal{L}_0$. Suppose that $\tau \in \mathcal{OP}(\mathcal{D};\, \phi)$. Then $\mathcal{D} \models \phi\tau$.

Proof. First we prove the claim when ϕ is atomic. By the Reduction Theorem 5.4 we have $\tau \in \mathcal{LD}(P(\mathcal{D})\,;\, \phi)$, so by the soundness of LD-resolution (see, e.g., [8]) $P(\mathcal{D}) \models \phi\tau$.

But each transformation rule considered in this section maintains semantic equivalence and so does the final transformation of a procedure declaration of the form (18) into the logic program (19). So $P(\mathcal{D})$ and $\mathcal{D}$ are semantically equivalent and consequently $\mathcal{D} \models \phi\tau$.

Suppose now that ϕ is not atomic. Let

$$\mathcal{D}' := \mathcal{D} \cup \{p_\phi(\mathbf{x}) \leftarrow \phi\},$$

where $\mathbf{x}$ is the sequence of free variables of ϕ and p_ϕ a new relation symbol with the appropriate arity.

Since $\tau \in \mathcal{OP}(\mathcal{D};\, \phi)$ we get $\tau \in \mathcal{OP}(\mathcal{D}';\, p_\phi(\mathbf{x}))$. By the fact that the claim is already established for atomic formulas, we have $\mathcal{D}' \models p_\phi(\mathbf{x})\tau$. Take now an arbitrary model I for $\mathcal{D}$. Interpret in it the relation symbol p_ϕ as the set $\{\mathbf{d} \mid I \models \phi\{\mathbf{x}/\mathbf{d}\}\}$. This extends I to the model I' of $\mathcal{D}'$, since in I' the equivalence $p_\phi(\mathbf{x}) \leftrightarrow \phi$ holds. Thus $I' \models p_\phi(\mathbf{x})\tau$, i.e., $I \models \phi\tau$. So $\mathcal{D} \models \phi\tau$. □

Corollary 5.6 (Completeness). Consider a declaration set $\mathcal{D}$ and a formula ϕ of $\mathcal{L}_0$. Suppose that $\mathcal{D} \models \phi\theta$. Then a $\tau \in \mathcal{OP}(\mathcal{D}\,;\, \phi)$ exists such that $\phi\tau$ is more general than $\phi\theta$.

Proof.

Again we first prove the claim when ϕ is atomic. Since, as already noted, $P(\mathcal{D})$ and $\mathcal{D}$ are semantically equivalent, we have $P(\mathcal{D}) \models \phi\theta$. By the completeness of LD-resolution (see, e.g., [8]) a $\tau \in \mathcal{LD}(P(\mathcal{D})\,;\, \phi)$ exists such that $\phi\tau$ is more general than $\phi\theta$. By the Reduction Theorem 5.4, $\tau \in \mathcal{OP}(\mathcal{D};\, \phi)$.

Suppose now that ϕ is not atomic. Let, as in the proof of the Soundness Corollary 5.5,

$$\mathcal{D}' := \mathcal{D} \cup \{p_\phi(\mathbf{x}) \leftarrow \phi\},$$

where $\mathbf{x}$ is the sequence of free variables of ϕ and p_ϕ a new relation symbol with the appropriate arity.

Then $\mathcal{D}' \models p_\phi(\mathbf{x})\theta$. By the fact that the claim is already established for atomic formulas, we get that a $\tau \in \mathcal{OP}(\mathcal{D}'\,;\, p_\phi(\mathbf{x}))$ exists such that $p_\phi(\mathbf{x})\tau$ is more general than $p_\phi(\mathbf{x})\theta$. The latter is equivalent to the statement that τ is more general than θ, which implies that $\phi\tau$ is more general than $\phi\theta$.

Moreover, by Def. 3.1, in particular axiom 7, there is exactly one possible transition for $\langle p_\phi(\mathbf{x}) \;;\; \epsilon\rangle$, namely

$$\langle p_\phi(\mathbf{x}) \ ; \ \epsilon\rangle \rightarrow \langle \phi \ ; \ \epsilon\rangle.$$

So $\tau \in \mathcal{OP}(\mathcal{D}' \, ; \, p_\phi(\mathbf{x}))$ implies $\tau \in \mathcal{OP}(\mathcal{D} \, ; \, \phi)$. □

6. Related Work

The literature on logic programming abounds in studies of various extensions of the basic framework considering more powerful fragments of first-order logic. We mention some works, without claim of completeness. Clark [5] studied an extension to general programs, where negative literals are allowed in clause bodies and queries. Lloyd and Topor [9] studied programs where each clause body is an arbitrary first-order formula with negation and universal quantifiers allowed. Miller et al. [12] studied an extension to hereditary Harrop formulas, where universal quantification and (embedded) implications are allowed. In a series of works, Minker and his co-authors (see e.g. Lobo, Minker and Rajasekar [10]) considered an extension to disjunctive logic programs, where disjunctions are allowed in the clause heads. Gelfond and Lifschitz [6] introduced an extension where two types of negations were allowed, classical negation and negation-as-failure.

In each of these works an appropriate computation mechanism was proposed to deal with the corresponding syntax extensions. In [9], in contrast to our work, the operational semantics of programs was obtained by reducing them first to general programs by means of transformations in the style of Section 5.

Let us clarify now the relation of our work with the Alma-0 language mentioned in the introduction and its semantics. In [1] we proposed an operational semantics for first-order formulas defined as an extension of $\mathcal{L}_0$ allowing negation, and we proved it to be correct with respect to the truth definition. The formulas were interpreted over an arbitrary algebra, which implies that equations had to be solved effectively in a way which is correct for any algebra. This was achieved by limiting the considerations to (1) the case of an assignment of a *ground* term (which thus corresponds to a value from the domain of the algebra) to an uninstantiated variable, (2) the case of an equality test between two ground terms. To ensure that the semantics was total it was stipulated that any other equation yields an *error state*. This semantics was introduced to study the declarative fragment of Alma-0.

As a follow-up on this work in Apt [3] we introduced a denotational semantics for first-order logic in which additionally arbitrary, i.e. not necessarily ground, terms could be assigned to an uninstantiated variable.

Recall now the following definition.

Definition 6.1. The *Herbrand algebra Her* for a language $\mathcal{L}_0$ of terms is defined as follows:

- its domain is the set HU_L of all ground terms of $\mathcal{L}_0$ (usually called the *Herbrand universe*),
- if f is an n-ary function symbol in $\mathcal{L}_0$, then its interpretation is the mapping from $(HU_L)^n$ to HU_L which maps the sequence $t_1, \ldots, t_n$ of ground terms to the ground term $f(t_1, \ldots, t_n)$. □

We can now clarify that the operational semantics of Section 3 is a special case of "computing in first-order logic over some algebra". Indeed, the following well-known lemma shows that in our case computations implicitly take place over the Herbrand algebra.

Lemma 6.2. Let $\mathbf{x}$ be the variables in the formula $s = t$. Then

$$Her \models \exists \mathbf{x}.\, s = t \text{ if and only if } s \text{ and } t \text{ unify.}$$

Proof. In the Herbrand algebra, a finite variable assignment coincides with a substitution. By the standard interpretation of the existential quantifier, $Her \models \exists \mathbf{x}.\, s = t$ if and only if there exists a substitution (= assignment) θ such that $Her \models (s = t)\theta$. By the definition of *Her*, $Her \models (s = t)\theta$ if and only if $s\theta$ and $t\theta$ are identical. So $Her \models \exists \mathbf{x}.\, s = t$ if and only if s and t unify. □

The difference is that in case of Herbrand algebras a stronger, effective and error-free, semantics of equations can be provided.

Results related to ours have recently been established by Seres [14]. She translated the formulas of $\mathcal{L}_0$ into Haskell programs in a compositional way and established algebraic laws concerning this translation. Subsequently she proved that the successful outcomes of the LD-resolution can be simulated by means of these algebraic laws, in the sense that given the IFF version of the considered logic program, a query can be transformed using the algebraic laws to the equational representation of the computed substitution. This provides an alternative, indirect, proof of the $\subseteq$ inclusion of the Embedding Theorem 4.5 of Section 4.

As mentioned in the introduction, the procedure-based view of logic programming was also used in de Bakker [4], the purpose of which was to compare several operational semantics for logic programs. The adopted syntax used the fragment of $\mathcal{L}_0$ without the existential quantier, but an extension with the cut operator was considered.

7. Some Reflections on the Rule-based and Procedure-based Views

While our results show, as expected, that the rule-based and the procedure-based view of logic programming are equivalent, in practice adopting one or the other view can have far-reaching consequences from the point of view of programming languages design. We review here these consequences to draw some conclusions about the design of languages that support logic programming.

7.1. Rule-based View

Consider first the rule-based view. Our discussion of it provides an a posteriori justification of the choices made in Prolog. As is well known, programs written in this format are very concise, among other reasons because the control is implicit.

The crucial characteristic concerning control in logic programming is the fact that in general, there are mutliple rules with the same relation symbol in the head. For one thing, this can be used, in combination with automatic backtracking, to implement non-determinism. However, non-determinism is not always intended, and then, patterns in combination with different input arguments in the clause heads are often used to select the right clause. In some cases, the textual ordering of the rules (for the same relation symbol) is also crucial, as will become clear below.

While the features mentioned in the previous paragraph are sometimes sufficient to implement adequate control, often more powerful primitives are needed. Adding these primitives while maintaining the rule-based view (according to which each clause body must be a sequence of atoms) leads to several contrivances and problems, which we want to clarify in this section. The alternative is to adopt a different view altogether, as will be discussed in Subsection 7.3.

Consider first the customary **if** B **then** S **else** T **fi** construct. If B is a simple test whether a term is of a certain form, this construct can be simulated by means of patterns and multiple rules with the same relation symbol in the head. But if B is a complex query, we need to resort to a different solution.

As is well-known, modelling by means of two rules

```
p(x) :- B, S.
p(x) :- not B, T.
```

where **x** is the sequence of all the variables that appear in `B,S` and `T`, introduces an inefficiency in the form of a reevaluation of `B` or its negation. [2] This naturally leads to the introduction of the cut operator with the revised modelling:

```
p(x) :- B, !, S.
p(x) :- T.
```

The cut can be also used in combination with patterns to simulate the `case` statement in Prolog. But its introduction obviously compromises the declarative character of the language: the rules cannot anymore be considered separately.

An alternative is to adopt a built-in **if** `B` **then** `S` **else** `T` **fi** construct, written in Prolog as `B -> S;T`. Here "$\rightarrow/2$" is a built-in relation symbol declared as an infix operator. So `B -> S;T` is an atom. But since `B,S` and `T` are supposed to be arbitrary queries we need to adjust the original framework. Namely, each rule has a format $H \leftarrow B_1, \ldots, B_n$, where H and all B_i are atoms. Since `B -> S;T` is an atom, we need to abandon the distinction between function and relation symbols, so that `B,S` and `T` can also be terms. Further, to allow `B,S` and `T` to be arbitrary queries we also need to view the comma "," as a binary relation symbol, so that each query is both an atom and a term.

This identification of function and relation symbols is one of the striking features of Prolog. It is called ambivalent syntax. It allows us to maintain the view that each clause body is a sequence of atoms, but in a way, this is very contrived, since ambivalent syntax is so powerful that the restriction "clause body is a sequence of atoms" becomes almost meaningless.

Ambivalent syntax can be accounted for in logical terms (see, e.g., Martens and De Schreye [11]), but in absence of typing it leads to a number of problems (see, e.g., Hill and Lloyd [7, pages 12-13]).

Identifying function and relation symbols has as a consequence identifying terms and atoms, and thus it is natural to introduce *meta-variables*, which are variables that appear in the position of an atom. After all, if $\rightarrow/2$ is a relation symbol, why shouldn't some of its arguments be variables? Meta-variables provide support for meta-programming and allow the programmer to introduce new programming constructs. As an example take the well-known definition of negation:

```
not(X) :- X, !, false.
not(_).
```

Meta-variables give rise to the possibility of run-time errors, of two kinds:

[2] We assume here, for simplicity, that `not B` can be rewritten to a query without the `not` symbol, just like `not X > Y` can be rewritten to `X =< Y`.

- if a query contains an atom `X` and `X` is uninstantiated at call time, an instantiation error will take place;
- if a query contains an atom `X` and `X` is instantiated to a term which cannot be interpreted as an atom, for example 42, a type error will take place.

The occurrence of each kind of error is in general an undecidable problem.

To conceal the use of the meta-variables one could employ the built-in relation `call/1` but this amounts to sweeping the problem under the carpet, since `call/1` is defined internally by

```
call(X) :- X.
```

So one design decision almost leads to another. The rule-based format explains the remarkable conciseness of Prolog, its flexible style of programming (in which several options can be chosen to model the control mechanism), and its support for meta-programming. For the sake of the subsequent discussion let us mention that this realization of meta-programming relies on so-called *non-ground representation* in which an object level variable is (implicitly) represented by a meta-level variable.

But these features come with a heavy price tag in the form of the cut operator, ambivalent syntax, meta-variables, lack of types and difficulty of type inference. They all look as inevitable consequences of the adoption of the rule-based view. The rule-based format thus forms a straitjacket that explains several striking aspects of Prolog.

7.2. Procedure-based View

In contrast, the procedure-based view is more verbose and often less readable. However, it is much more familiar to those who mastered first such programming languages as Pascal or C. Moreover, this format is much easier to extend and modify. Indeed, it is straightforward to add new syntactic constructs while maintaining the benefits of type checking. The reason is that the statements are already defined by structural induction. Moreover, different forms of parameter mechanisms can be easily introduced by just appropriately annotating the occurrences of the formal parameters in the procedure head.

The procedure-based view was adopted in Alma-0. This view made it relatively straightforward to integrate various features of logic programming into the imperative programming paradigm. For example, to account for the double use of queries, for computing and for testing, a new parameter mechanism was added to the language and it amalgamated easily with the customary Pascal and Modula-2 call-by-value and call-by-variable parameter mechanisms. Alma-0 supports type checking. However, as a side effect, meta-programming is no longer supported and no new user-defined programming constructs can be added.

7.3. A Hybrid View

Now, both views can be combined in a third view that we call a *hybrid view.*

The idea is to generalize the format of the rules and not to insist that in a rule $H \leftarrow B_1, \ldots, B_n$ all $B_1, \ldots, B_n$ are atoms. This is what is done in the theory of logic programming, for example by allowing B_i's to be literals when studying Prolog programs that use negation. In principle B_i's can be arbitrary formulas (representing programming constructs) in a specific first-order or higher-order language, as long as one can provide to such an extension a computational interpretation.

This view has been adopted in the the programming languages Mercury of Somogyi, Henderson and Conway [15] and Gödel of Hill and Lloyd [7]. In Mercury the rules are of the form $H \leftarrow \mathbf{B}$ where the body $\mathbf{B}$ is a "valid goal". Valid goals are defined by structural induction and are specific first-order formulas. In particular `B -> S;T` (which can be also written as `if B then S else T`) is a valid goal if `B,S` and `T` are valid goals. There are no meta-variables, hence a variable cannot be a goal and thus `B` is not a variable. The queries `X = true, (X -> true ; q(b))` and `X -> true ; q(b)` mentioned above are both syntactically illegal and rejected at compile-time.

This view allows for type checking and type inference. Further, the use of modes allows one to formalize the notion of an input argument (i.e., parameter). However, also in this approach, meta-programming is no longer supported and no new user-defined programming constructs can be added.

A similar approach is adopted in Gödel. The Gödel programs are typed but modes are absent. In contrast to Mercury meta-programming is supported but in a radically different way than in Prolog. It is realized by means of the system module `Programs` together with the so-called *ground representation* in which all object level expressions, including variables, are represented as ground terms at the meta-level. Since there are no meta-variables in Gödel, a goal `if B then S else T`, where `B` is a variable, will be rejected, just as mentioned for Mercury above.

In conclusion, the decision which view to adopt can have far-reaching consequences for the design of the logic programming languages. The most typical languages, exemplified by Mercury and Gödel, rely on the h ybrid view that reconciles the rule-based and the procedure-based view.

Acknowledgements

We thank Kees Vermeulen and the referees of an early version of this paper for helpful comments.

References

[1] K. R. Apt and M. A. Bezem. Formulas as programs. In K.R. Apt, V.W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: A 25 Year Perspective*, pages 75–107, 1999. Available via `http://arXiv.org/archive/cs/`.

[2] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5):1014–1066, 1998.

[3] K. R. Apt. A denotational semantics for first-order logic. In J. Lloyd, editor, *Proc. of Computational Logic 2000 (CL2000)*, Lecture Notes in Artificial Intelligence 1861, pages 53–69. Springer-Verlag, 2000. Available via `http://arXiv.org/archive/cs/`.

[4] J. W. de Bakker. Comparative semantics for flow of control in logic programming without logic. *Information and Computation*, 94(2):123–179, 1991.

[5] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[6] M. Gabbrielli and G. Levi. A solved form algorithm for ask and tell Herbrand constraints. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91, Volume 1 (CAAP'91)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1991.

[7] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[8] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[9] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1:225–240, 1984.

[10] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* The MIT Press, 1992.

[11] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *Journal of Logic Programming*, 22(1):47–99, 1995.

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