Proof-theory for Extensions of Logic Programming

Catholijn M. Jonker
Institut für Informatik und angewandte Mathematik, Universität Bern
Current address: Department of Computer Science, Free University,
De Boelelaan 1081a, 1081 HV Amsterdam, jonker@cs.vu.nl

Keywords: Rule-based calculus, negations, program completions.

Abstract. The focus of this paper lies on the proof-theory for extensions of Logic Programming in which it is possible to draw negative conclusions both in a direct (i.e., based on a proof) and in an indirect (i.e., based on the lack of a proof) way. These extensions are provided with a rule-based deductive system in the sense of the work of Jäger [4] for Normal Logic Programs. Rule-based deductive systems can be used as a powerful tool to study the structural properties of the logic programming languages. Furthermore, in the deductive systems the fundamental semantical properties of the languages can be formalised by proof-rules of the systems. Therefore, different extensions of logic programming can be compared by comparing their deductive systems.

1 Introduction

In [4] Jäger presented rule-based Tait-style calculi for Normal Logic Programs. These calculi proved to be closely related to SLDNF-resolution and Clark's Completion. In the following we show that this work is easily extendible for extensions of Normal Logic Programming.

In this article two extensions are considered that are of practical use in modern knowledge representation. Knowledge representation languages that are to be of practical use require at least options for drawing negative conclusions. It must be possible to derive such information via direct means, but for practical reasons it must also be possible to derive negative information by the absence of other information. In other words, it must be possible to obtain negative statements by giving a proof for them and it must be possible to obtain a negative statement since the corresponding positive statement is believed to be unprovable. In this paper two approaches are considered that have both properties. In the first approach two different interpretations are used to interpret the negative conclusions. The other approach is based on one uniform interpretation. These two approaches can easily be formalised as extensions of Normal Logic Programming. For the first approach a different negation symbol is associated to each of the interpretations, it is known under the name Extended Logic Programming. For the second approach, known as Imex Logic Programming [6] one negation symbol suffices.

As a core semantics Clark's Completion is extended both for Extended and for Imex Logic Programming. In this article the extensions of Clark's Completion for Extended and Imex Logic Programming are formalised syntactically within the rule-based deductive systems. Having the deductive systems and semantics, Extended and Imex Logic Programming are to be compared formally.

In Section 2 the languages for the deductive systems and for Extended and Imex Logic Programming are described. Furthermore, it is formally defined how to interpret these languages.

Section 3 contains the definition of Clark's Completion for Normal Logic Programming and the extensions of Clark's Completion for Extended and Imex Logic Programming.

In Section 4 the rule-based calculi are defined for Normal, Extended and Imex Logic Programming.

Section 5 contains the soundness and completeness results of the deductive systems with respect to the extensions of Clark's Completion.
In Section 6 a notion of program equivalence is defined and transformations are given under which the deductive systems of the (transformed) Extended and Imex Logic Programs are equivalent.

Section 7 briefly discusses the results presented in this paper.

2 Basic Notions

2.1 Languages

In this article, we will make use of three propositional languages, $\mathcal{L}_E$, $\mathcal{L}_I$ and $\mathcal{L}_D$. $\mathcal{L}_E$ is used for Extended and Normal Logic Programming, $\mathcal{L}_I$ for Imex Logic Programming and $\mathcal{L}_D$ for the deductive systems.

In this paper the alphabet of language $\mathcal{L}_E$ consists of atomic symbols A, A_0, A_1, ..., the negation symbols "$-$", for explicit negation, "not" for implicit negation, and the symbols "$\leftarrow$" and ",". The literals of $\mathcal{L}_E$ are all expressions of the form A, $\mathtt{not}A$, $-A$ and $\mathtt{not}-A$, where A is an atom. An Extended Logic Programming clause is an expression of the form:

$$S_0 \leftarrow S_1, \ldots, S_m, \mathtt{not}S_{m+1}, \ldots, \mathtt{not}S_n$$

where each S_i $(0 \leq i \leq n)$ is either an atom A or an explicitly negated atom $-A$. The symbol "not" retains basically the same meaning as in Normal Logic Programming, i.e., $\mathtt{not}A$ means A is *believed to be false.* Whereas $-A$ is made true if there is some sort of proof for it, so that "$-$" stands for *proven to be false*. The nature of the proof depends on the interpretation used and is the same as the nature of a proof for the truth of A.

An Extended Logic Program consists of a finite set of Extended Logic Programming clauses. An Extended Logic Program P is a Normal Logic Program iff "$-$" does not occur in P. The program clauses of $\mathcal{L}_E$ are the Extended Logic Programming clauses.

Example 1 Consider the program consisting of the two rules

$$A \leftarrow \mathrm{not}B$$
$$-A \leftarrow \mathrm{not}B$$

B will be false because there is no rule with B as its head. Thus, not$B$ will be true. Based on the truth of not$B$ both A and $-A$ will be provable. ■

This example also makes it clear that there can be problems in that A and $-A$ can have a proof. In the early versions of Extended Logic Programming a literal $-A$ behaved like an atomic symbol and not as a complement of the symbol A. In other words, nothing was added to Normal Logic Programming. Pereira [12] first noted this problem and then solved it by adding the semantic Coherence Principle to Extended Logic Programming. The Coherence Principle formulates that

$$A \text{ implies } \mathrm{not}{-A} \quad \text{and}$$
$$-A \text{ implies } \mathrm{not}A$$

The Coherence Principle influences the truth of statements not$A$ in a way not known in Normal Logic Programming. Thus, the meaning of the symbol "not" in Extended Logic Programming differs slightly from its meaning in Normal Logic Programming. For Example 1 this means that A (and therefore also $-A$) will get the value *overdefined*, B stays *false*. Several ways to interpret Extended Logic Programs in a declarative way have been developed, see [3, 8, 12, 13] and [15]. In this paper we consider a new extension, $comp_E$, of Clark's Completion which will be introduced in Section 3.

The alphabet of $\mathcal{L}_I$ is the same as that of $\mathcal{L}_E$, except for the negation symbols. $\mathcal{L}_I$ has only one negation symbol: "$\sim$". The literals of $\mathcal{L}_I$ are all expressions of the form A and $\sim A$, where A is an atom. The informal interpretation of "$\sim$" is *believed to be false* and is therefore the same as that of "not". An Imex Logic Programming clause is an expression of the form:

$$S \leftarrow A_1, \ldots, A_m, \sim A_{m+1}, \ldots, \sim A_n$$

where each A_i $(1 \leq i \leq n)$ is an atom A and S is either an atom A or an imex negated atom $\sim A$. An Imex Logic Program consists of a finite set of Imex

Logic Programming clauses. The program clauses of $\mathcal{L}_I$ are the Imex Logic Programming clauses. Several ways to interpret Imex Logic Programs have been developed and we refer to [6]. Like in Extended Logic Programming it is possible to write inconsistent Imex Logic Programs.

Example 2 Consider the program consisting of the two rules

$$A \leftarrow {\sim}B$$
$${\sim}A \leftarrow {\sim}B$$

Since the informal interpretation of "$\sim$" is the same as that of "not", B will be false for the same reason as in Example 1, i.e., there is no rule with B as its head. Thus, ${\sim}B$ will be true. Based on the truth of ${\sim}B$ both A and ${\sim}A$ will be provable. ■

A program is either an Extended or an Imex Logic Program. The literal on the left-hand side of a program clause is called the *head* and the sequence on the right-hand side is called the *body* of the clause. The *Herbrand base* $\mathcal{B}_P$ of program P is the collection of all atoms that occur somewhere in P.

$\mathcal{L}_D$ is a language of standard propositional logic. That is, we have A, A_0, $A_1, \ldots$ as proposition symbols, $\bot$ and $\top$ as logical constants, $\wedge$ and $\vee$ as connectives and the symbol $\bar{}$ to build the complements $\overline{A}$, $\overline{A_0}$, $\overline{A_1}$, ... of the proposition symbols. The formulas are built in the standard way. The negation "$\neg$" is defined inductively: $\neg A := \overline{A}$, $\neg\overline{A} := A$, $\neg\top := \bot$, $\neg\bot := \top$, $\neg(F_1 \vee F_2) := \neg F_1 \wedge \neg F_2$ and $\neg(F_1 \wedge F_2) := \neg F_1 \vee \neg F_2$. The connectives $\leftarrow$, $\leftrightarrow$ are defined as usual.

The complexity of formulas will often be measured in terms of their rank. We define the rank of literals and of $\top$ and $\bot$ to be 0 and the rank of more complex formulas inductively in the standard way.

2.2 Many-valued Interpretations of $\mathcal{L}_D$

We use the well-known four-valued logic FOUR of Belnap [2], which uses the set of truth values {**t**, **f**, **u**, **o**}, standing for *true*, *false*, *undefined* and *overdefined* respectively.
Four truth-values are needed since both Extended and Imex Logic Programs can contain inconsistencies, see Example 3 and also Examples 1 and 2.

Example 3 Informally, the Imex Logic Program P that consists of the clauses:

$$A \leftarrow$$
$${\sim}A \leftarrow$$

means that A should be true and ${\sim}A$ should be true. In other words, A is overdefined. ■

A *three-valued lower interpretation*, denoted by *low*-valued interpretation, of $\mathcal{L}_D$ is a four-valued interpretation $\mathcal{I}$ so that $\mathcal{I}(A) \neq \mathbf{o}$ for all atomic symbols A of $\mathcal{L}_D$. A *three-valued upper interpretation*, denoted by *up*-valued interpretation of $\mathcal{L}_D$ is a four-valued interpretation $\mathcal{I}$ so that $\mathcal{I}(A) \neq \mathbf{u}$ for all atomic symbols A of $\mathcal{L}_D$. A *two-valued interpretation* of $\mathcal{L}_D$ is a four-valued interpretation $\mathcal{I}$ so that $\mathcal{I}(A) \in \{\mathbf{t}, \mathbf{f}\}$ for all atomic symbols A of $\mathcal{L}_D$.
Let $\mathcal{I}$ be a j-valued interpretation ($j \in \{2, up, low, 4\}$) of $\mathcal{L}_D$, F a $\mathcal{L}_D$-formula and T an $\mathcal{L}_D$-theory, then we define as usual: F is *j-valid* in $\mathcal{I}$ and $\mathcal{I}$ is a *logical j-model* of F if $\mathcal{I}(F) \in \{\mathbf{o}, \mathbf{t}\}$. We write $\mathcal{I} \models_j F$. $\mathcal{I}$ is a *logical j-model* of T if all elements of T are j-valid in $\mathcal{I}$. We write $\mathcal{I} \models_j T$. Further we define F to be a *logical j-consequence* of T if $\mathcal{N}(F) \in \{\mathbf{o}, \mathbf{t}\}$ for all logical j-models $\mathcal{N}$ of T. Then we write $T \models_j F$.
The main problem in the procedural and declarative approach to explicit negation in Logic Programming arises as a result of the presence of negations in both the body and the head of the program clauses. Therefore, in a deductive approach to explicit negation, we have to reflect that positive information may depend on negative information and vice versa. Considering that even in Normal Logic Programming many approaches exist to the treatment of negative information and there is a gap between most declarative semantics and the procedural semantics, it is not surprising that there is as yet no clear approach to the procedural and declarative treatment of negative entities in extensions of Normal Logic Programming. In the next section we will give a declarative semantics to Extended and to Imex Logic Programming.

3 Completion

We define for Extended and Imex Logic Programs P extensions of Clark's Completion that transform P into a theory $comp_E(P)$, respectively $comp_I(P)$, of $\mathcal{L}_D$. We take the models of the $\mathcal{L}_D$-theory $comp_E(P)$, respectively $comp_I(P)$, as the semantics of the program P.

A lot of work has been done on the declarative semantics of Extended Logic Programs [3, 6, 11, 12, 13], leading to extensions of the well-known Stable, Stationary, Supported and Wellfounded Semantics. The declarative semantics mentioned above reflect the more ideal meanings of a Normal or Extended Logic Program. However all have the problem of being very hard to compute once variables and functions are allowed in the language [5, 10]. Clark's Completion can be seen as a core semantics for Normal Logic Programming. All negative conclusions drawn using Clark's Completion can also be drawn in the leading declarative semantics. For some purposes this is a reason to consider Completion too weak to be of interest. For our purposes Completion provides exactly enough; it contains the core issues of the semantics of the negative conclusions. Furthermore, Clark's Completion can be used to give a fairly precise description of the procedural SLDNF-resolution[1]. First Kunen [9] and later Jäger and

[1] For a survey of various versions of SLDNF-resolution we refer to [1].

Stärk [5] gave characterisations of classes of Normal Logic Programs for which SLDNF-resolution is complete with respect to Clark's Completion. These results can be related to soundness and completeness results for derivability in a deductive system and Clark's Completion, see [5].

In this article Clark's Completion and the deductive systems are extended to Extended and Imex Logic Programming. In Section 5 the customary soundness and completeness results are given. In Section 5 transformations are defined between the Extended and the Imex Logic Programs and it is shown that the deductive systems for corresponding Extended and Imex Logic Programs have the same declarative semantics.

The reader is expected to be familiar with the notion *defining formula*, denoted by D_L^P, and Clark's Completion for Normal Logic Programs, denoted $comp_N(P)$, which is the set $\{A \leftrightarrow D_A \mid A \in \mathcal{B}_P\}$.

3.1 Normal Logic Programming

As a first step, we define the usual *defining formula*. In case of Normal Logic Programming, let ρ denote the operator:

$$\rho(A) = A \quad \rho(\mathbf{not} A) = \neg A$$

First, replace every comma by the conjunction symbol"$\wedge$". Then, collect for every atom A the bodies of all clauses with head A into one disjunction D_A, called the *defining formula* of A. Finally, replace all clauses with head A by the formula $A \leftrightarrow D_A$. This way we get a theory $comp_N(P)$ of $\mathcal{L}_D$ instead of a NLP. Formally, we define the defining formula for Normal Logic Programs as follows.

Definition 4 (Defining Formula) *Let P be a Normal or an Imex Logic Program. Suppose there are m clauses in P with head L so that the i-th clause is of the form*

$$L \leftarrow L_{i,1}, \ldots, L_{i,k(i)}$$

and has $k(i)$ literals in its body. Then the defining formula *of L with respect to P is the $\mathcal{L}_D$-formula*

$$D^P_{\rho(L)} := \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k(i)} \rho(L_{i,j})$$

The special cases $m = 0$ and $k(i) = 0$ are included by interpreting empty disjunctions as $\perp$ and empty conjunctions as $\top$. If no confusion is expected we write $D_{\rho(L)}$ instead of $D^P_{\rho(L)}$.

Since the literal L that is the head of the clauses in the above formula can also be a negated atom, we implicitly defined the defining formulas $D_{\neg A}$ of negative literals $\sim A$ of $\mathcal{L}_I$. In Section 3.3 we will see that the same definition can be used for Extended Logic Programming, since every Extended Logic Program is first transformed into a Normal Logic Program. The difference between Normal and Extended Logic Programming lies in the definitions of $comp_N$ and $comp_E$.

Definition 5 *Let P be a Normal Logic Program, then $comp_N(P)$ is the set $\{A \leftrightarrow D_A \mid A \in \mathcal{B}_P\}$.*

Definition 6 *Let P be a Normal Logic Program. A j-valued interpretation $\mathcal{I}$ ($j \in \{2, up, low, 4\}$) of $\mathcal{L}_D$ is a j-*model *of P iff for every atomic symbol A of $\mathcal{L}_E$ $\mathcal{I}(D_A) \leq_k \mathcal{I}(A)$. And we write $\mathcal{I} \models_j P$. Further we define an $\mathcal{L}_D$-formula F to be a* program j-consequence *of P if we have $\mathcal{N}(F) \in \{\mathbf{o}, \mathbf{t}\}$ for all j-models $\mathcal{N}$ of P. Then we write $P \models_j F$.*

3.2 Imex Logic Programming

The idea behind Clark's Completion for Normal Logic Programs is that the program contains all necessary information. That is, the truth of an atom is completely defined by its defining formula. If we enrich our programming language by allowing atoms to be made explicit false, then this information can only influence the truth of an atom if its defining formula provides not enough information to make the atom either true or undefined. In other words the defining formula of a negated literal is not complete; the information in the defining formula of its opposite positive literal has a direct influence on it. Therefore, we see in Definition 7 and also for Extended Logic Programming in Definition 11 that there is an asymmetry with respect to the negations. Definition 4 needs to be extended to a defining formula $D_{\neg A}$ of $\sim A$ for Imex Logic Programs in order to be able to cope with the Imex negation symbol $\sim$. This implies that the body of every clause must be translated into $\mathcal{L}_D$ by replacing every negation symbol $\sim$ by the classical negation symbol $\neg$ and every comma by the conjunction symbol"$\wedge$". This way we get a theory $comp_I(P)$ of $\mathcal{L}_D$ instead of a program of $\mathcal{L}_I$. Therefore, we extend the definition of the operator ρ to:

$$\rho(A) = A \; \rho(\text{not}A) = \neg A \; \rho(\sim A) = \neg A$$

Using ρ we define the completion of P as:

Definition 7 *Let P be an Imex Logic Program, then $comp_I(P)$ is the set of $\mathcal{L}_D$-formulas $\{A \leftrightarrow D_A \; , \; \neg A \leftarrow D_{\neg A} \mid A \in \mathcal{B}_P\}^2$.*

[2] Note that $A \leftarrow \neg D_{\neg A}$ is unwanted in the completion of P for an atomic symbol A, since then $A \leftrightarrow \neg D_{\neg A}$ as well. If we consider the Imex Logic Program $P = \{A \leftarrow \sim A\}$, then $D_A = \neg A$, $\neg D_{\neg A} = \top$ and thus the completion of this program would be: $\{A \leftrightarrow \neg A \; , \; \neg A \leftrightarrow \bot\}$. Hence, $A \leftrightarrow \top \leftrightarrow \neg A$. In other words, from P one would be able to conclude both A and $\neg A$ instead of the generally excepted undefinedness of A.

Definition 8 *Let P be an Imex Logic Program. A j-valued interpretation $\mathcal{I}$ ($j \in \{2, up, low, 4\}$) of $\mathcal{L}_D$ is a j-*model *of P iff for every atomic symbol A of $\mathcal{L}_I$*

1. $\mathcal{I}(D_A) \leq_k \mathcal{I}(A)$.
2. $\mathcal{I}(D_{\neg A}) \in \{\mathbf{o}, \mathbf{t}\}$ *implies* $\mathcal{I}(\neg A) \in \{\mathbf{o}, \mathbf{t}\}$.

And we write $\mathcal{I} \models_j P$. Further we define an formula F to be a program j-consequence *of P if we have $\mathcal{N}(F) \in \{\mathbf{o}, \mathbf{t}\}$ for all j-models $\mathcal{N}$ of P. Then we write $P \models_j F$.*

The second condition in the above definition allows for example $\mathcal{I}(D_{\neg A}) = \mathbf{o}$ while $\mathcal{I}(\neg A) = \mathbf{t}$. This reflects that the information in D_A is more important than the information in $D_{\neg A}$.

Example 9 Consider the program P consisting of the clauses:

$$\begin{aligned} {\sim}A_1 &\leftarrow A_2 \\ A_2 &\leftarrow \\ {\sim}A_2 &\leftarrow \end{aligned}$$

Of course A_2 must get the value $\mathbf{o}$, however, the lack of clauses with head A_1 implies that A_1 must false. The overdefinedness of A_2 should not influence the value of A_1. We aim for locality of inconsistency. ■

3.3 Extended Logic Programming

In Extended Logic Programming two kinds of negation are used, "**not**" is the standard negation of Normal Logic Programming and "$-$" is an explicit negation related to classical negation. The way an explicitly negated atom $-A$ is treated is by considering such a literal as a new relation symbol, say nA, which represents the opposite of A. To formalise this principle we let an operator ψ transform every Extended program P into a Normal program $\psi(P)$ in the language $\mathcal{L}'_E$. For every atomic symbol A of $\mathcal{L}_E$ operator ψ substitutes every occurrence of $-A$ by nA and all other occurrences of A by pA. The symbols nA and pA are new atomic symbols which replace the symbol A in $\mathcal{L}_E$, creating the language $\mathcal{L}'_E$. Similarly, we have the languages $\mathcal{L}'_D$ and $\mathcal{L}'_I$. $\mathcal{L}'_D$ is interpreted in the same way as $\mathcal{L}_D$. The idea of using new atomic symbols to stand for $-A$ is not new, see for example Alferes and Pereira [12] and Minker and Ruiz [11].

Definition 10 *Let P be an Extended Logic Program. We define the operator ψ inductively by*

$$\psi(L) = \begin{cases} pA & \textit{if } L = A \quad \textit{for some atom } A \\ \mathbf{not}\, pA & \textit{if } L = \mathbf{not} A \quad \textit{for some atom } A \\ nA & \textit{if } L = \neg A \quad \textit{for some atom } A \\ \mathbf{not}\, nA & \textit{if } L = \mathbf{not}\neg A \quad \textit{for some atom } A \end{cases}$$

$$\psi(L_0 \leftarrow L_1, \ldots, L_n) = \psi(L_0) \leftarrow \psi(L_1), \ldots, \psi(L_n)$$

$$\psi(P) = \{\psi(c) \mid c \in P\}$$

On the syntactical level, this representation thus reduces the Extended program P to a Normal Logic Program $\psi(P)$ in the language $\mathcal{L}'_E$. However, nA and pA are unrelated in $\psi(P)$. Therefore, a so called *Coherence Principle* [12] extends the semantics of $\psi(P)$ to a semantics of P by saying that

$$pA \text{ implies not } nA \quad \text{and}$$
$$nA \text{ implies not } pA$$

Note that the Coherence Principle is not expressible in Normal Logic Programming. Furthermore, the explicit negation symbol is weaker than the symbol $\neg$ of $\mathcal{L}_D$, which can be seen from the fact that **not** nA does not imply pA. Instead of considering Extended programs P in $\mathcal{L}_E$, we will as of now consider only their corresponding Normal programs $\psi(P)$ in $\mathcal{L}'_E$, however, with the extra addition of the Coherence Principle. The defining formula of an ELP is defined using the same operator ρ that was used for Imex Logic Programming. This way we get a theory $comp_N(\psi(P))$ of $\mathcal{L}'_D$ instead of a program of $\mathcal{L}_E$.

When no confusion is expected, we often address $\psi(P)$ as P. We define the *completion* of Extended Logic Program P using the Coherence Principle.

Definition 11 *Let P be an Extended Logic Program, then $comp_E(P)$ is the set of $\mathcal{L}'_D$-formulas:*
$\{nA \leftrightarrow D_{nA}\ ,\ pA \leftrightarrow D_{pA}\ ,\ \neg nA \leftarrow pA\ ,\ \neg pA \leftarrow nA \mid A \in \mathcal{B}_P\}$[3].

Definition 12 *Let P be an Extended Logic Program. A j-valued interpretation $\mathcal{I}$ ($j \in \{2, up, low, 4\}$) of $\mathcal{L}'_D$ is a j-*model *of P iff for all atomic symbols pA and nA of $\mathcal{L}'_E$*

1. $\mathcal{I}(D_{pA}) \leq_k \mathcal{I}(pA)$
2. $\mathcal{I}(D_{nA}) \leq_k \mathcal{I}(nA)$
3. $\mathcal{I}(nA) \in \{\mathbf{o}, \mathbf{t}\}$ *implies* $\mathcal{I}(pA) \in \{\mathbf{o}, \mathbf{f}\}$
4. $\mathcal{I}(pA) \in \{\mathbf{o}, \mathbf{t}\}$ *implies* $\mathcal{I}(nA) \in \{\mathbf{o}, \mathbf{f}\}$

And we write $\mathcal{I} \models_j P$. Further we define an formula F to be a program j-consequence *of P if we have $\mathcal{N}(F) \in \{\mathbf{o}, \mathbf{t}\}$ for all j-models $\mathcal{N}$ of P. Then we write $P \models_j F$.*

Since the program j-models of a Logic Program P agree with the models of P's Completion, it is obvious that a formula F is a program j-consequence of P if and only if it is a logical j-consequence of the Completion of P. In light of this we will just use consequence instead of logical or program consequence. Whenever deemed appropriate, one can restrict oneself to the 2-, *up*- or *low*-consequences.

[3] Note that $\neg nA \leftarrow pA$ and $\neg pA \leftarrow nA$ correspond to the Coherence Principle. In other words both the implicit **not** and the explicit $-$ negation are translated using the classical negation $\neg$. Furthermore, $comp_E(P) = comp_N(\psi(P)) \cup \{\neg nA \leftarrow pA\ ,\ \neg pA \leftarrow nA \mid A \in \mathcal{B}_P\}$. The ψ-transformation is used to formalise the Coherence Principle in the definition of a model of $comp_E$.

4 Deductive systems

Although we defined the meaning of a program to be the set of all consequences of a specific set of axioms associated with the program, there is also an alternative method to interpret Logic Programming that is conceptually closer to a procedural understanding of Logic Programming. This method can be seen as an interpretation of Logic Programming by a *clauses-as-rules* paradigm instead of the traditional *clauses-as-axioms* interpretation. We introduce a rule-based calculus $\mathcal{R}^i(P)$ for every Imex Logic Program P and a rule-based calculus $\mathcal{R}^e(P)$ for every Extended Logic Program P. The systems $\mathcal{R}^i(P)$ and $\mathcal{R}^e(P)$ are designed for a proof-theoretic treatment and form a link between Logic Programming and inductive definability. Furthermore, these systems provide us with a way to prove that Imex and Extended Logic Programming are equivalent methods to extend Normal Logic Programming with an explicit negation.

The deduction systems are presented in a Tait-style manner. Accordingly, the axioms and derivation rules are formulated for finite sets of $\mathcal{L}_D$-formulas which have to be interpreted disjunctively. If $\mathcal{I}$ is a j-valued $\mathcal{L}_D$-structure and Γ a finite set of $\mathcal{L}_D$-formulas, then $\mathcal{I}(\Gamma)$ is the truth value of the disjunction of the elements of Γ with respect to $\mathcal{I}$. The systems $\mathcal{R}^i(P)$ and $\mathcal{R}^e(P)$ are extensions of the usual Tait calculus for propositional logic by adding so called *Program Rules.* The Program Rules correspond to the formulas in the program Completions for Extended (Imex) Logic Programs. We consider the following five classes of axioms and rules, where Γ is a finite set of $\mathcal{L}_D$-formulas (respectively $\mathcal{L}'_D$-formulas, when considering Extended programs).

I. Logical axioms. For all atomic $\mathcal{L}_D$- (respectively $\mathcal{L}'_D$-) formulas F:

$\Gamma, \top$ and the *identity* axioms $\Gamma, \neg F, F$

II. Logical rules. For all $\mathcal{L}_D$- (respectively $\mathcal{L}'_D$-) formulas F_1 and F_2 :

$$\frac{\Gamma,F_1}{\Gamma,F_1\vee F_2}\ (\vee\ 1)\quad \frac{\Gamma,F_2}{\Gamma,F_1\vee F_2}\ (\vee\ 2)\quad \frac{\Gamma,F_1 \qquad \Gamma,F_2}{\Gamma,F_1\wedge F_2}\ (\wedge)$$

III. Cut rules. For all $\mathcal{L}_D$- (respectively $\mathcal{L}'_D$-) formulas F:

$$\frac{\Gamma,F \qquad \Gamma,\neg F}{\Gamma}\ (\text{cut})$$

The formulas F and $\neg F$ are called the *cut formulas* of this cut; the rank of a cut is the rank of its cut formulas.

IV. Imex Program Rules for P. For every atomic symbol A of $\mathcal{L}_I$ and its defining formulas D^P_A and $D^P_{\neg A}$ we have the following program rules:

$$\frac{\Gamma,D_A}{\Gamma,A}\ (\text{I.1})\quad \frac{\Gamma,\neg D_A}{\Gamma,\neg A}\ (\text{I.2})\quad \frac{\Gamma,D_{\neg A}}{\Gamma,\neg A}\ (\text{I.3})$$

V. Extended Program Rules for P. For every atomic symbol A of $\mathcal{L}_E$ and its defining formulas $D^{\psi(P)}_{nA}$ and $D^{\psi(P)}_{pA}$ in $\mathcal{L}'_D$ we have the following program rules:

$$\frac{\Gamma,D_{nA}}{\Gamma,nA}\ (\text{E.1})\ \frac{\Gamma,\neg D_{nA}}{\Gamma,\neg nA}\ (\text{E.2})\ \frac{\Gamma,D_{pA}}{\Gamma,pA}\ (\text{E.3})\ \frac{\Gamma,\neg D_{pA}}{\Gamma,\neg pA}\ (\text{E.4})\ \frac{\Gamma,nA}{\Gamma,\neg pA}\ (\text{E.5})\ \frac{\Gamma,pA}{\Gamma,\neg nA}\ (\text{E.6})$$

The last two rules correspond to the Coherence Principle of Extended Logic Programming. In Section 3.3 the explicit negation was linked to the implicit negation by way of the Coherence Principle. In the Completion and in the Program Rules, the implicit negation is expressed using the classical negation. Therefore, it is not surprising that also the link between the explicit and implicit negation formed by the Coherence Principle is translated into rules using the classical negation. The $\mathcal{L}_D$-calculi $\mathcal{R}^i(P)$ for Imex programs P consist of the classes **I** through **IV**, the $\mathcal{L}'_D$-calculi $\mathcal{R}^e(P)$ for Extended programs P consist of the classes **I** through **III** and **V**. Based on these axioms and rules, derivability in $\mathcal{R}^i(P)$ and $\mathcal{R}^e(P)$ are introduced in the standard way. The notation $\mathcal{R}^i(P) \vdash^n_r \Gamma$ (c.q. $\mathcal{R}^e(P) \vdash^n_r \Gamma$) expresses that Γ is provable in $\mathcal{R}^i(P)$ (c.q. $\mathcal{R}^e(P)$) by a proof whose length and cut complexity are bounded by n and r, respectively. A proof is called *cut-free* if it does not use the cut-rule, which is therefore denoted by $\vdash_0$. A proof is called *identity-free* if the identity axioms are not used in the proof, this is denoted by the symbol $\Vdash$.

Example 13 Consider again the Imex Logic Program P from Example 3. $\mathcal{R}^i(P)$ is determined by the Program Rules:

$$\frac{\Gamma,\top}{\Gamma,A}\quad\frac{\Gamma,\bot}{\Gamma,\neg A}\quad\frac{\Gamma,\top}{\Gamma,\neg A}$$

Thus, $\mathcal{R}^i(P) \Vdash \bot$, $\mathcal{R}^i(P) \Vdash_0 A$ and $\mathcal{R}^i(P) \Vdash_0 \neg A$, but not $\mathcal{R}^i(P) \Vdash_0 \bot$. ■

Example 14 Consider the Extended Logic Program P:

$$A \leftarrow$$
$$-A \leftarrow$$

Then the Program Rules for P are:

$$\frac{\Gamma,\top}{\Gamma,nA}\quad\frac{\Gamma,\bot}{\Gamma,\neg nA}\quad\frac{\Gamma,\top}{\Gamma,pA}\quad\frac{\Gamma,\bot}{\Gamma,\neg pA}\quad\frac{\Gamma,nA}{\Gamma,\neg pA}\quad\frac{\Gamma,pA}{\Gamma,\neg nA}$$

Thus, $\mathcal{R}^e(P) \Vdash \bot$, $\mathcal{R}^e(P) \Vdash_0 pA$ and $\mathcal{R}^e(P) \Vdash_0 \neg pA$, but not $\mathcal{R}^e(P) \Vdash_0 \bot$. ■

In the following sections, we will prove Soundness and Completeness of $\mathcal{R}^i(P)$ and $\mathcal{R}^e(P)$, and show that these rule based calculi prove that Imex and Extended programming are equivalent.

5 Soundness and Completeness of the extensions

As is already the case for Normal Logic Programs, it makes sense to specifically consider the identity-free derivations since the identity axioms create problems in the point of view of three- and four-valued models of the completions of programs (see also [4]). For Extended- and Imex Logic Programs these problems even occur for identity-free derivations, forcing us to consider four-valued logics instead. In order to isolate inconsistencies within an Extended or Imex Logic Program we have to forbid cuts. This means that for Extended and Imex Logic Programming cut-free provability is an interesting notion. We have the following soundness and completeness results:

Theorem 15. *Let P be an Imex Logic Program and P' an Extended Logic Program. For every finite set Γ of $\mathcal{L}_D$-formulas we have*

$$\begin{array}{ll}
\mathcal{R}^i(P) \vdash \Gamma \iff comp_I(P) \models_2 \Gamma & \mathcal{R}^e(P') \vdash \Gamma \iff comp_E(P') \models_2 \Gamma \\
\mathcal{R}^i(P) \vdash_0 \Gamma \iff comp_I(P) \models_{up} \Gamma & \mathcal{R}^e(P') \vdash_0 \Gamma \iff comp_E(P') \models_{up} \Gamma \\
\mathcal{R}^i(P) \Vdash \Gamma \iff comp_I(P) \models_{low} \Gamma & \mathcal{R}^e(P') \Vdash \Gamma \iff comp_E(P') \models_{low} \Gamma \\
\mathcal{R}^i(P) \Vdash_0 \Gamma \iff comp_I(P) \models_4 \Gamma & \mathcal{R}^e(P') \Vdash_0 \Gamma \iff comp_E(P') \models_4 \Gamma
\end{array}$$

We proved the above theorem using Schütte's deduction-chains (see [14]) for programs with a countable number of variables, constants and function symbols.

6 Program equivalence of ELP and Imex

The following results concern the relation between Imex and Extended programming. In [6] we proved that for every Extended Logic Program there is an Imex Logic Program that has the same models and vice versa. This statement was proved using essentially the same transformations Ψ and ϕ between Extended and Imex Logic Programs that will be described below. One can prove a similar theorem for the model-theoretical Stable, Static, Supported and Wellfounded semantics [6]. For that theorem we proved a one-to-one correspondence between the interpretations of an Extended Logic Program P and its corresponding Imex Logic Program P' and vice versa.
A similar theorem for the deductive systems of Extended and Imex Logic Programs will be formulated below. We use the following definitions.

Definition 16 *For any $\mathcal{L}'_D$-formula F we define F^* to be $F[\neg A/nA\ ,\ A/pA]$, i.e., the substitution of nA by $\neg A$ and pA by A for all atoms of the forms pA and nA. For any set Γ of $\mathcal{L}'_D$-formulas we define Γ^* as the set $\{F^* \mid F \in \Gamma\}$.*

Definition 17 *If P_1 is an Extended Logic Program in $\mathcal{L}_E$ and P_2 is an Imex Logic Program in $\mathcal{L}'_I$ (respectively $\mathcal{L}_I$), then P_1 and P_2 are equivalent iff for every set Γ of $\mathcal{L}'_D$-formulas*

1. *$\mathcal{R}^e(P_1) \vdash_r \Gamma$ iff $\mathcal{R}^i(P_2) \vdash_r \Gamma$ (respectively iff $\mathcal{R}^i(P_2) \vdash_r \Gamma^*$).*

2. $\mathcal{R}^e(P_1) \Vdash_r \Gamma$ *iff* $\mathcal{R}^i(P_2) \Vdash_r \Gamma$. *(respectively iff* $\mathcal{R}^i(P_2) \Vdash_r \Gamma^*$*).*

If P_2 is a program in $\mathcal{L}_I$, then the new symbols pA and nA have to be converted back to the original literals before a comparison to P_1 can be made.

Theorem 18. *For every Extended Logic Program there is an equivalent Imex Logic Program and vice versa.*

The reader is referred to [7] for a formal proof. Here only a sketch is given.

(Imex $\Leftarrow$ ELP) Let P be an Extended Logic Program. We define $\Psi(P)$ to be the $\mathcal{L}'_I$-Imex Logic Program $\nu(\psi(P)) \cup \{\sim pA \leftarrow nA, \sim nA \leftarrow pA \mid A \in \mathcal{L}_E\}$, where ψ is the operator defined in Section 3.3 and ν substitutes every occurrence of the symbol "not" by the symbol "$\sim$". Now it is easy to check that $\mathcal{R}^e(P) = \mathcal{R}^i(\Psi(P))$.

Example 19 For program P of Example 14 the program rules of $\mathcal{R}^e(P)$ are syntactically the same as those of $\mathcal{R}^i(\Psi(P))$, where $\Psi(P)$ is the Imex Program given by the clauses:

$$\begin{aligned} pA &\leftarrow \\ nA &\leftarrow \\ \sim pA &\leftarrow nA \\ \sim nA &\leftarrow pA \end{aligned}$$

■

The converse (Imex $\Rightarrow$ ELP) is less obvious; we translate every Imex Logic Program P in an Extended program by way of the following inductively defined ϕ-operator:

$$\phi(L) = \begin{cases} A & \text{if } L = A \\ \mathbf{not} A & \text{if } L = \sim A \end{cases}$$

$$\phi(L_0 \leftarrow L_1, \dots, L_n) = \begin{cases} A \leftarrow \phi(L_1), \dots, \phi(L_n) & \text{if } L_0 = A, \text{ for some atom } A \\ -A \leftarrow \phi(L_1), \dots, \phi(L_n) & \text{if } L_0 = \sim A, \text{ for some atom } A \end{cases}$$

$$\phi(P) = \{\phi(c) \mid c \in P\} \cup \{-A \leftarrow \mathbf{not} A \mid A \in \mathcal{B}_P\}$$

Assume there are m clauses in P with head $\sim A$ so that the i-th clause is of the form $\sim A \leftarrow L_{i,1}, \dots, L_{i,k(i)}$ and has $k(i)$ literals in its body. For convenience we use

$$\phi(D^P_{\neg A}) := \bigvee_{i=1}^{m} \bigwedge_{j=1}^{k(i)} \rho(\phi(L_{i,j}))$$

Note that $D^{\phi(P)}_{nA} = \neg pA \vee \phi(D^P_{\neg A})$. The presence of the clauses $-A \leftarrow \mathbf{not} A$ in $\phi(P)$ in combination with the Coherence Principle, ensures the following lemma.

Lemma 20. *Let P be an Imex Logic Program. For every set Γ of $\mathcal{L}'_D$-formulas:*

- $\mathcal{R}^i(P) \vdash_r \Gamma^*$ *iff* $\mathcal{R}^e(\phi(P)) \vdash_r \Gamma$ *and*
- $\mathcal{R}^i(P) \Vdash_r \Gamma^*$ *iff* $\mathcal{R}^e(\phi(P)) \Vdash_r \Gamma$.

For the proof of this lemma we refer to [7].

Example 21 Consider again the Imex Logic Program P from Example 3. $\mathcal{R}^i(P)$ is determined in Example 13. Then $\phi(P)$ is the Extended Program given by the clauses:

$$A \leftarrow$$
$$-A \leftarrow$$
$$-A \leftarrow \mathbf{not} A$$

And $\mathcal{R}^e(\phi(P))$ has the following program rules:

$$\frac{\Gamma, \top \vee \neg pA}{\Gamma, nA} \quad \frac{\Gamma, \bot \wedge pA}{\Gamma, \neg nA} \quad \frac{\Gamma, \top}{\Gamma, pA} \quad \frac{\Gamma, \bot}{\Gamma, \neg pA} \quad \frac{\Gamma, nA}{\Gamma, \neg pA} \quad \frac{\Gamma, pA}{\Gamma, \neg nA}$$

Compare $\mathcal{R}^i(P)$ and $\mathcal{R}^e(\phi(P))$ with the calculi of the Examples 14 and 19. Then we see, that, in this case, P is also equivalent to the program of Example 14. ■

In conclusion, we established that Extended and Imex Logic Programming are equivalent with respect to Clark's Completion, the Stable, Stationary, Supported and Wellfounded Semantics.

7 Conclusion

Every ELP/Imex program has been provided with a deductive system. In these deductive systems only one negation symbol is used, however the expressive power of both negation concepts in ELP is obtained. Under extensions of Clark's Completion, these deductive systems adhere to the principles of Soundness and Completeness.

Moreover, different semantical principles of Logic Programming can be formalised syntactically in the deductive systems. The idea of negation as failure as used in Clark's Completion is present in the form of the program rules (I.2), (E.2) and (E.4). Also, the deductive systems for Extended Logic Programs contain the Coherence Principle syntactically in the form of the program rules (E.5) and (E.6). For Imex Logic Programming, the extra possibility for deriving falsehood is present in the form of the program rules (I.3).

Furthermore, ELP and Imex were proved to be semantically equivalent under the aforementioned extensions of Clark's Completion. For equivalent results under the Stable, Stationary, Supported and Wellfounded semantics, we refer to [6].

References

1. K.R. Apt and R.N. Bol. Logic programming and negation: a survey. Technical Report Report CS-R9402, January 1994.
2. N.D. Belnap. A useful four-valued logic. In J.M. Dunn and G. Epstein, editors, *Modern uses of Multiple-valued Logic*, pages 8–37. D. Reidel Publishing Company, 1977. Dordrecht.
3. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D.H.D. Warren and P. Szeredi, editors, *7th International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.
4. G. Jäger. A deductive approach to logic programming. In H. Schwichtenberg, editor, *Proof and Computation*, pages 133–172. 1994. Series F: Computer and Systems Sciences, NATO Advanced Study Institute.
5. G. Jäger and R. Stärk. A proof-theoretic framework for logic programming. In S. Buss, editor, *Handbook of Proof Theory*. To appear.
6. C.M. Jonker. *Constraints and Negations in Logic Programming.* PhD thesis, Dept. of Philosophy, Utrecht University, 1994. Quæstiones Infinitæ vol. 10, Dissertation.
7. C.M. Jonker. Rule-based calculi for extensions of logic programming. Technical Report IAM 95–006, Institut für Informatik und angewandte Mathematik, Universität Bern, 1995.
8. R.A. Kowalski and F. Sadri. Logic programs with exceptions. In D.H.D. Warren and P. Szeredi, editors, *Logic Programming, Proceedings 7th International Conference*, pages 598–613. MIT Press, 1990.
9. K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7:231–245, 1989.
10. V. Marek and M. Truszczyński. *Nonmonotonic logic.* Springer Verlag Heidelberg, 1993.
11. J. Minker and C. Ruiz. Semantics for disjunctive logic programs with explicit and default negation. *Fundamenta Informaticae*, 20:145–192, 1994.
12. L.M. Pereira and J.J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *Proceedings 10th European Conference on Artificial Intelligence, ECAI'92*, pages 102–106, 1992.
13. T.C. Przymusinski. Static semantics for normal and disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 1994.
14. K. Schütte. *Proof Theory.* Springer, 1977.
15. G. Wagner. A database needs two kinds of negation. In B. Thalheim, J. Demetrovics, and H.D. Gerhardt, editors, *MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 357–371. Springer-Verlag, 1991.