A Proof Procedure for Extended Logic Programs

Frank Teusink
Centre for Mathematics and Computer Science
P.O. Box 4079 1009 AB Amsterdam
The Netherlands
frankt@cwi.nl

Abstract

In [GL90], M. Gelfond and V. Lifschitz proposed to extend general to so-called *extended logic programs*, by adding strong negation. They proposed *answer sets* as a semantics for these programs. However, this semantics uses the notion of *global consistency*. The necessity of testing for global consistency makes finding a proof for a specific query w.r.t. a program as hard as finding a complete answer set for that program. In this paper, we abandon the idea of preserving global consistency and propose a modified transformation from extended logic programs to general logic programs, based on a semantics in which only *local consistency* is preserved. We use the notion of *conservative derivability*, as defined by G. Wagner in [Wag91], as a proof-theoretic semantics for extended logic programs, and show that the three-valued completion semantics of a transformed program is sound and complete with respect to conservative derivability in the original extended logic program. As a result, we can use any proof procedure for general logic programs that is sound with respect to completion semantics, to answer queries with respect to extended logic programs. We illustrate our proof procedure by using it to prove queries with respect to an extended logic program discussed in [GL90].

1 Introduction

Extended logic programs were introduced by M. Gelfond and V. Lifschitz in [GL90], to overcome some problems in dealing with incomplete information. In this paper, we present a proof procedure for these extended logic programs. The reason for developing this proof procedure is, that we want to be able to compute answers to queries w.r.t. an extended logic program, without first having to compute some intended model of that program.

The proof procedure we present is based on a transformation from extended logic programs to general logic programs (this transformation differs from the one defined by Gelfond and Lifschitz). We have chosen a transfor-

mational approach, because it enables us to profit from work done on proof procedures for general logic programs. The transformation we propose implements the notion of conservative derivability as introduced by G. Wagner in [Wag91]. As a result, for an extended logic programs without function symbols, the three-valued completion semantics of a transformed program is sound and complete with respect to the notion of conservative derivability in the original extended logic program.

As a semantics for extended logic programs, Gelfond and Lifschitz defined the so-called *answer sets* of an extended logic program. These sets are defined in terms of the stable models of a derived general logic program, provided the extended logic program is consistent. The proof procedure we define, will be neither sound nor complete with respect to the answer set semantics. The reason for our proof procedure not being complete is, that the problem of testing whether a general logic program has a stable model is Σ_1^1-complete (see corollary 5.12 in [MNR92]). Consequently, no effective proof procedure can be complete with respect to answer set semantics. The reason for our proof procedure not being sound with respect to answer set semantics is, that conservative reasoning is a form of *paraconsistent* reasoning, i.e. it allows us to derive meaningful answers to queries w.r.t. inconsistent extended logic programs, while the answer set semantics collapses in the case of inconsistent extended logic programs; everything becomes true.

In the next section, we give a short introduction to extended logic programs and introduce some notation used throughout the paper. Section 3 explains the notion of conservative reasoning. In section 4, we define the transformation of an extended logic program P to a general logic program P_{cr}, and prove that a query Q w.r.t. P is conservatively derivable from P if and only if Q' is a logical consequence of $comp(P_{cr})$, where Q' is derived from Q by some transformation. In section 5, we use SLDNF-resolution to compute answers to queries w.r.t. an extended logic program discussed by Gelfond and Lifschitz in [GL90]. Finally in section 6, we relate our transformation to the one proposed by Gelfond and Lifschitz.

2 Preliminaries and notation

A *general logic program* is a finite set of clauses of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$$

where, for $i \in [0..n]$, A_i is an atom. Formulas of the form A or $\mathit{not}\ A$, where A is an atom, are called *literals*. The negation used in general logic programs, is interpreted as negation as (finite) failure: $\mathit{not}\ A$ is true whenever one fails to (finitely) derive A and $\mathit{not}\ A$ is false if one can derive A (finitely). However, in some cases it is useful to have a stronger notion of negation (notation: $\sim$), in which $\sim A$ is true iff $\sim A$ can be derived. This is called *strong negation*. For this, Gelfond and Lifschitz introduced *extended logic*

programs. In extended logic programs, we use both negation as failure (*not*) and strong negation ($\sim$). So, wherever one could write an atom in a general logic program, one can write an atom or a strongly negated atom in an extended logic program. Thus, an *extended logic program* is a finite set of clauses of the form

$$L_0 \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_n$$

where, for $i \in [0..n]$, L_i is a *literal* (i.e. a formula of the form A or $\sim A$, where A is an atom). Formulas of the form L or *not* L, where L is a literal, are called *extended literals.* Note, that in a general logic program, a literal is of the form A or *not* A, while in an extended logic program, a literal is of the form A or $\sim A$. The $\leftarrow$ in extended logic programs should not be read as classical implication. Instead, clauses in an extended logic program should be seen as inference rules.

We now want to give a justification for our choice of symbols for strong negation and negation as (finite) failure. The symbol $\neg$ is generally used for classical negation. Moreover, in general logic programs, negation as failure is generally denoted by either '$\neg$' or '*not*'. In [GL90], '*not*' is used for negation as failure and '$\neg$' is used for strong negation. In [Prz90], '$\sim$' is used for negation as failure and '$\neg$' is used for strong negation. (In both [GL90] and [Prz90] they refer to the second form of negation as classical negation.) Finally, in [Wag93] '$-$' is used for negation as failure (or *weak negation*, as it is called there), '$\sim$' is used for strong negation and '$\neg$' is used for classical negation. We use '$\neg$' for classical negation, '$\sim$' for strong negation and '*not*' for negation as failure. The use of '$\neg$' for classical negation is standard. Moreover, the second form of negation used in extended logic programming differs from classical negation. Therefore, one should use a different symbol, so why not follow [Wag93] and use '$\sim$'. Finally, for negation as failure, the obvious choice is that between '*not* ' and '$-$'. We chose '*not* ', because it seems to be more standard than '$-$'.

In this paper, we use $A, A', A_i, \ldots$ to denote atoms, $L, L', L_i, \ldots$ to denote (extended) literals and F, G, H to denote formulas. We identify a sequence $L_1, \ldots, L_k$ of (extended) literals with the conjuction $L_1 \wedge \ldots \wedge L_k$. Moreover, we sometimes identify a conjuction F of (extended) literals with the set of (extended) literals in F. For the sake of simplicity, we treat both negations on (extended) literals as complement operators, i.e. $L \equiv \mathit{not}\ \mathit{not}\ L$ and $L \equiv \sim\sim L$. Note, that *not* and $\sim$ are not commutative, so we do not have that $\mathit{not}\ \sim \mathit{not}\ L \equiv \sim L$.

For a logic program P (either general or extended), $\mathcal{B}_P$ denotes the Herbrand Base of P and $\mathcal{L}_P$ denotes the set of (extended) literals build from atoms in $\mathcal{B}_P$. An *interpretation* for P is a subset of $\mathcal{L}_P$ (note that interpretations can be inconsistent). The set of ground instances of clauses in P is denoted by $\mathit{ground}(P)$.

3 Conservative reasoning

In [Wag91], G. Wagner introduces the notion of *conservative reasoning* as a means to reason with inconsistent programs (he also introduces other systems to deal with inconsistent programs, but in this paper we are only interested in conservative reasoning). The system he proposes in this paper, uses only strong negation. In [Wag93], he presents a system that incorporates negation as failure (he calls it *weak negation*, and uses $-$ to denote it), but is more restricted in other aspects. In this section, we present a combination of these two systems.

The language consists of the logical symbols $\wedge$ (and), $\vee$ (or), $\sim$ (strong negation), *not* (weak negation) and $\mathbf{t}$ (verum), predicate symbols, constants and variables. We obtain this language by adding *not* to the language in [Wag91] or $\vee$ to the language in [Wag93]. Just like in [Wag91] and [Wag93], the language does not contain function symbols. This restriction is necessary, because we will define the derivability relation $\vdash$ in terms of deduction rules; the restriction ensures that the number of premises in the deduction rules for ground literals are finite. As a consequence of this restriction, not every extended logic program can be represented as a program in this language.

The definition of a program is the same as the definition of an extended logic program. As a result, every extended logic program without function symbols is a program in this system. This definition of a program is more restricted than the definition in [Wag91], where the body of a clause is an arbitrary formula. However, we are only interested in extended logic programs, and therefore do not need arbitrary formulas in bodies of clauses.

The *conservative derivability relation* $\vdash$ is defined by a natural deduction system. The idea of conservative derivability is based on the idea of mutual neutralization, i.e. $\{A, \sim A\} \nvdash A$. Intuitively, this means that if both A and $\sim A$ can be 'proven', we discard all 'proofs' for both A and $\sim A$. As a result, we not only lose conclusions, but also gain new ones, because *not* A and *not* $\sim A$ can be derived. Informally, $P \vdash F$ means that the existential closure of F can be proven in P without using inconsistent knowledge in P. After introducing the deduction rules, we illuminate the idea of conservative derivability by an example. The most important rules in this system are the rules for deriving *ground extended literals*:

$$(l) \qquad \frac{\exists(L \leftarrow F) \in \mathit{ground}(P) : P \vdash F \quad \forall(\sim L \leftarrow F) \in \mathit{ground}(P) : P \vdash \mathit{not}\ F}{P \vdash L}$$

$$(\mathit{not}\ l_1) \qquad \frac{\forall(L \leftarrow F) \in \mathit{ground}(P) : P \vdash \mathit{not}\ F}{P \vdash \mathit{not}\ L}$$

$$(\mathit{not}\ l_2) \qquad \frac{\exists(\sim L \leftarrow F) \in \mathit{ground}(P) : P \vdash F}{P \vdash \mathit{not}\ L}$$

The deduction rule (l) combines the notion of derivability by ground clauses

with the notion of mutual neutralization: $P \vdash L$ if there exists a ground rule for L whose body is conservatively derivable, provided that $\sim L$ is not conservatively derivable. The deduction rules $(not\ l_1)$ and $(not\ l_2)$ state the converse, i.e. $P \vdash not\ L$ means that L is not conservatively derivable, either because there does not exist a ground clause for L whose body is conservatively derivable, or by mutual neutralization.

Furthermore, there are rules for deriving *complex ground formulas*:

$$(not\ not) \quad \frac{P \vdash F}{P \vdash not\ not\ F}$$

$$(\wedge) \quad \frac{P \vdash F, G}{P \vdash F \wedge G} \qquad (not\ \wedge) \quad \frac{P \vdash not\ F}{P \vdash not\ (F \wedge G)}$$

$$(\vee) \quad \frac{P \vdash F}{P \vdash F \vee G} \qquad (not\ \vee) \quad \frac{P \vdash not\ F, not\ G}{P \vdash not\ (F \vee G)}$$

Note that these rules only hold for *ground* formulas.

Example 3.1 Consider the program P_1 with clauses $p(a) \leftarrow$ and $q(b) \leftarrow$. It is reasonable to deduce that $P_1 \vdash p(x), q(x)$ (i.e. $\exists x\ p(x)$ and $\exists x\ q(x)$), but to deduce $P_1 \vdash p(x) \wedge q(x)$ (i.e. $\exists x\ p(x) \wedge q(x)$) by deduction rule $(\wedge)$ is clearly wrong. ∘

Finally, there is a rule for deriving *complex non-ground formulas*:

$$(\exists) \quad \frac{P \vdash F\theta \text{ for some substitution } \theta}{P \vdash F}$$

and of course the rule to derive verum: $\vdash \mathbf{t}$.

Example 3.2 Consider following program P_2:

$$\begin{aligned} r &\leftarrow \mathbf{t} \\ p &\leftarrow r \\ \sim p &\leftarrow r \\ q &\leftarrow not\ p \\ \sim q &\leftarrow \sim r \end{aligned}$$

We deduce $P_2 \vdash r$ by (l) using $\vdash \mathbf{t}$ and $P_2 \vdash not\ \sim r$ by $(not\ l_1)$. Moreover, we have by $(not\ l_2)$ (mutual neutralization) $P_2 \vdash not\ p$ and $P_2 \vdash not\ \sim p$. Finally, we deduce $P_2 \vdash not\ \sim q$ by $(not\ l_1)$ and $P_2 \vdash q$ by (l). ∘

The derivability relation defined by these deduction rules differs from both the system in [Wag91] and the system in [Wag93]. In contrast with [Wag91] and in accordance with [Wag93], we can only derive $\sim F$, if F is an atom. This is reasonable, because we can use *not* to negate complex formulas. Extending the derivability relation to strongly negated complex formulas is beyond the scope of this paper. With this relation, we can derive non-ground formulas. This can be done with the system in [Wag91], but not with the system in [Wag93]. We need the derivability of non-ground formulas for the soundness and completeness results in section 4.

4 The *cr* transformation

The idea of our proof procedure is, to find out whether a goal is conservatively derivable from a program. If the goal is conservatively derivable, the proof procedure should answer *yes*; otherwise, it should answer *no*. We define our proof procedure in terms of a derived general logic program P_{cr}. The three-valued completion of P_{cr} will be sound and complete with respect to conservative derivability in P (for extended logic programs without function symbols). As a result, we are free to use any proof procedure for general logic programs that is sound with respect to the three-valued completion semantics, as a proof procedure for extended logic programs.

The idea of P_{cr} is, to split the declaration of a predicate in P into a positive and a negative part, just like Gelfond and Lifschitz did when transforming an extended logic program P into a general logic program P'. The difference is, that we then combine these positive and negative declarations of a predicate into a declaration of the original predicate, in a way that ensures consistency of the derived program (with respect to $\sim$; a general or extended logic program is inherently consistent with respect to negation as finite failure).

First, we present the transformation used by Gelfond and Lifschitz (the transformed program $\widehat{P}$ we define, is the program Gelfond and Lifschitz refer to as P').

Definition 4.1 Let L be a language.

- The language $\widehat{L}$ is the same as L, but
  - without the logical connective $\sim$, and
  - with an additional predicate symbol $\sim p$, for every predicate symbol p in L.
- For a formula F in L, $\widehat{F}$ is the formula in $\widehat{L}$ that is obtained from F by interpreting every combination $\sim p$ of the logical symbol $\sim$ and a predicate symbol p as the predicate symbol $\sim p$. If $\sim$ appears in F other than in front of an atom, $\widehat{F}$ is not defined.
- For a clause R of the form $L \leftarrow F$, $\widehat{R}$ is the clause $\widehat{L} \leftarrow \widehat{F}$.
- For a program P, $\widehat{P}$ is the program $\{\widehat{R} \mid R \in P\}$.

□

Note that $\widehat{F}$ is not always defined. However, by construction of the derivability relation, the fact that $\widehat{F}$ is not defined implies that $P \vdash F$ does not hold.

Definition 4.2 Let P be an extended logic program. P_{cr} is the general logic program such that

- for every clause $A \leftarrow F$ (resp. $\sim A \leftarrow F$) in P, P_{cr} contains the clause $A^p \leftarrow \widehat{F}$ (resp. $A^n \leftarrow \widehat{F}$), and
- for every atom A in P, P_{cr} contains the clauses $\widehat{A} \leftarrow A^p, not\ A^n$ and $\widehat{\sim A} \leftarrow A^n, not\ A^p$.

□

Note that $\mathcal{B}_{\widehat{P}} \subseteq \mathcal{B}_{P_{cr}}$.

In the remainder of this section, we prove that $comp(P_{cr})$ is sound and complete with respect to conservative derivability in P, in the sense that $comp(P_{cr}) \models_3 \exists \widehat{F}$ iff $P \vdash F$. We cannot prove soundness or completeness for arbitrary extended logic programs, simply because Wagner's definition of a program does not provide for function symbols. So, the soundness and completeness theorems are restricted to extended logic programs without function symbols.

First, we need the following lemma, which proves that the least fixpoint of the Fitting operator $\Phi_{P_{cr}}$ (see [Fit85]) is 'sound' with respect to the conservative derivability relation.

Lemma 4.3 *Let P be an extended logic program without function symbols, and let L be a ground extended literal in $\mathcal{L}_P$. Then, for all natural numbers n, $\widehat{L} \in \Phi^n_{P_{cr}}$ implies $P \vdash L$.*

Proof: We prove the claim by induction on n. For $n = 0$, the claim holds trivially, because $\Phi^0_{P_{cr}} = \emptyset$. Assume that, for all m less than n, $\widehat{L} \in \Phi^m_{P_{cr}}$ implies $P \vdash L$. First, we make the following observations:

1. $A^p \in \Phi^n_{P_{cr}}$, where A^p is ground, implies that there exists a $A \leftarrow F$ in $ground(P)$ such that $P \vdash F$.

   Suppose that $A^p \in \Phi^n_{P_{cr}}$. By construction of P_{cr} and $\Phi_{P_{cr}}$, there exists a formula F such that $A^p \leftarrow \widehat{F}$ in $ground(P_{cr})$ and $\widehat{F} \subseteq \Phi^m_{P_{cr}}$, for some m less than n. By induction hypothesis, for all conjuncts $L \in F$, $P \vdash L$ and therefore, by deduction rule $(\wedge)$, $P \vdash F$. Moreover, by construction of P_{cr}, $A \leftarrow F \in ground(P)$.

2. $not\ A^p \in \Phi^n_{P_{cr}}$, where A^p is ground, implies that, for all $A \leftarrow F$ in $groundP$, $P \vdash not\ F$.

   Suppose $not\ A^p \in \Phi^n_{P_{cr}}$. By construction of P_{cr} and $\Phi_{P_{cr}}$, for every formula F such that $A^p \leftarrow \widehat{F}$ is in $ground(P_{cr})$, $not\ \widehat{F} \cap \Phi^m_{P_{cr}} \neq \emptyset$, for some m less than n. By induction hypothesis, for every $A^p \leftarrow \widehat{F}$ in $ground(P_{cr})$ there exists an extended literal $L \in F$ such that $P \vdash not\ L$, and therefore by deduction rule $(not\ \vee)$, $P \vdash not\ F$. Moreover, by construction of P_{cr}, $A^p \leftarrow \widehat{F} \in ground(()P_{cr})$ iff $A \leftarrow F \in ground(()P)$. Therefore, for all $A \leftarrow F$ in $ground(P)$, $P \vdash not\ F$.

3. $A^n \in \Phi^n_{P_{cr}}$, where A^n is ground, implies that there exists a $\sim A \leftarrow F$ in *ground*(P) such that $P \vdash F$.

   The proof of this is a variant of the proof in 1.

4. *not* $A^n \in \Phi^n_{P_{cr}}$, where A^n is ground, implies that, for all $\sim A \leftarrow F$ in *groundP*, $P \vdash$ *not* F.

   The proof of this is a variant of the proof in 2.

Using these observations, we can prove the lemma. Suppose that $\widehat{L} \in \Phi^n_{P_{cr}}$. There are two cases:

- $L \equiv A$ or $L \equiv \sim A$. By construction, P_{cr} contains exactly one clause with conclusion $\widehat{L}$.

  If $L \equiv A$, this clause is of the form $A \leftarrow A^p$, *not* A^n. Because $A \in \Phi^n_{P_{cr}}$, A^p, *not* $A^n \in \Phi^n_{P_{cr}}$. By observation 1, there exists a $A \leftarrow F$ in P such that $P \vdash F$. By observation 4, for all $\sim A \leftarrow F$ in P, $P \vdash$ *not* F. By deduction rule (l), it follows that $P \vdash A$.

  The case where $L \equiv \sim A$ is symmetric.

- $L \equiv$ *not* A or $L \equiv$ *not* $\sim A$. By construction, P_{cr} contains exactly one clause with conclusion *not* $\widehat{L}$.

  If $L \equiv$ *not* A, this clause is of the form $A \leftarrow A^p$, *not* A^n. Because *not* $A \in \Phi^n_{P_{cr}}$ we have that *not* $A^p \in \Phi^n_{P_{cr}}$ or $A^n \in \Phi^n_{P_{cr}}$. Therefore, by observations 2 and 3, for all $A \leftarrow F$ in P, $P \vdash$ *not* F or there exists a $\sim A \leftarrow F$ in P such that $P \vdash F$. Therefore, either by deduction rule (*not* l_1) or by deduction rule (*not* l_2), we have that $P \vdash$ *not* A.

  The case where $L \equiv$ *not* $\sim A$ is symmetric.

□

We now prove soundness and completeness of the *cr* transformation. For this, we use three-valued completion semantics (Kunen semantics) of general logic programs, as proposed by K. Kunen in [Kun87]. One should note, that the idea of (three-valued) completion semantics is, that negation as finite failure in a general logic program P is characterized by classical negation in *comp*(P). Thus, the negation used in *comp*(P) is $\neg$ instead of *not*. In the following, we keep this conversion between negation as finite failure and classical negation implicit, and will consistently use *not* in the context of general logic programs and $\neg$ in the context of three-valued completion semantics.

Theorem 4.4 (Soundness of the *cr* transformation) *Let P be an extended logic program and let F be a formula in the language of P. Then,* $\text{comp}(P_{cr}) \models_3 \exists \widehat{F}$ *implies* $P \vdash F$.

Proof: Suppose that *comp*$(P_{cr}) \models_3 \exists \widehat{F}$ for some formula F in the language of P. We prove that $P \vdash F$ by induction on the complexity of F.

Suppose that F is a ground literal. Then $\exists\widehat{F}$ is also ground, and therefore $\exists\widehat{F} \leftrightarrow \widehat{F}$. So, $comp(P_{cr}) \models_3 \widehat{F}$. By theorem 6.3 in [Kun87], $\widehat{F} \in \Phi^n_{P_{cr}}$, for some finite n. Because $\widehat{F}$ is a ground literal, we conclude by lemma 4.3 that $P \vdash F$.

Suppose that F is a ground formula. Then $\exists\widehat{F}$ is also ground, and therefore $\exists\widehat{F} \leftrightarrow \widehat{F}$. We prove by induction on the structure of F that $P \vdash F$. Suppose that $F \equiv \neg(G \vee H)$. Because $comp(P_{cr}) \models_3 \widehat{F}$, we have that $comp(P_{cr}) \models_3 \neg\widehat{G}$ and $comp(P_{cr}) \models_3 \neg\widehat{H}$. By induction, it follows that $P \vdash not\ G$ and $P \vdash not\ H$. Thus by deduction rule ($not\vee$), $P \vdash not\ (G \vee H)$. For F equivalent to $\neg\neg G$, $G \wedge H$, $\neg(G \wedge H)$ or $G \vee H$, the proofs are similar.

Suppose F is a non-ground formula. $comp(P_{cr}) \models_3 \exists\widehat{F}$ implies that, for some ground instantiation θ, $comp(P_{cr}) \models_3 \widehat{F}\theta$. By induction, it follows that $P \vdash F\theta$. Thus, by deduction rule ($\exists$), $P \vdash F$. □

Theorem 4.5 (Completeness of the *cr* transformation) *Let P be an extended logic program. Let F be a formula in the language of P. If $P \vdash F$ then* comp$(P_{cr}) \models_3 \exists\widehat{F}$.

Proof: $P \vdash F$ implies that there exists a finite sequence $F_1, \ldots, F_k \equiv F$ of formulas in the language of P such that, for all $i \in [1..k]$, F_i is the result of applying one of the deduction rules for which, for every condition of the form $P \vdash F'$, $F' \equiv F_j$ for some j less than i. Therefore, in order to prove that $comp(P_{cr}) \models_3 \exists\widehat{F}$, it is sufficient to prove for each of the deduction rules that (in $comp(P_{cr})$) the conclusion is implied by the conditions.

The only deduction rules that are less than straightforward, are (l), ($not\ l_1$) and ($not\ l_2$), the rules for deriving ground extended literals.

- Consider deduction rule (l). Suppose there exists a clause $A \leftarrow F$ in $ground(P)$ such that $comp(P_{cr}) \models_3 \widehat{F}$. Then, there exists a clause $A^p \leftarrow \widehat{F}$ in $ground(P_{cr})$, and therefore $comp(P_{cr}) \models_3 A^p$. Moreover, suppose that, for all clauses $\sim A \leftarrow F$ in $ground(P)$, $comp(P_{cr}) \models_3 \neg\widehat{F}$. Then, for all clauses $A^n \leftarrow \widehat{F}$ in $ground(P_{cr})$, $comp(P_{cr}) \models_3 \neg\widehat{F}$. Thus, by construction of $comp(P_{cr})$, $comp(P_{cr}) \models_3 \neg A^n$. Because $comp(P_{cr})$ models P_{cr} and $A \leftarrow A^p, not\ A^n$ is in $ground(P_{cr})$, $comp(P_{cr}) \models_3 A$.

  The case for deriving $\sim A$ using (l) is similar.

- Consider deduction rule ($not\ l_1$). Suppose that for all clauses $A \leftarrow F$ in $ground(P)$, $comp(P_{cr}) \models_3 \neg\widehat{F}$. Then, by construction of P_{cr}, for all clauses $A^p \leftarrow \widehat{F}$ in $ground(P_{cr})$, $comp(P_{cr}) \models_3 \neg\widehat{F}$. Thus, by construction of $comp(P_{cr})$, $comp(P_{cr}) \models_3 \neg A^p$. Because $A \leftarrow A^p, not\ A^n$ is the only clause in $ground(P_{cr})$ with conclusion A, $comp(P_{cr}) \models_3 \neg A$.

  The case for deriving $not\ \sim A$ using ($not\ l_1$) is similar.

- Consider deduction rule ($not\ l_2$). Suppose that there exists a clause $\sim A \leftarrow F$ in $ground(P)$ such that $comp(P_{cr}) \models_3 \widehat{F}$. Then, there exists

a clause $A^n \leftarrow \widehat{F}$ in $ground(P_{cr})$ such that $comp(P_{cr}) \models_3 \widehat{F}$, and therefore $comp(P_{cr}) \models_3 A^n$. Because $A \leftarrow A^p, not\ A^n$ is the only clause in $ground(P_{cr})$ with conclusion A, $comp(P_{cr}) \models_3 \neg A$.

The case for deriving $not\ \sim A$ using $(not\ l_2)$ is similar.

□

Corollary 4.6 *Let P be an extended logic program and let F be a conjunction of extended literals.*

(i) *If θ is an SLDNF computed answer substitution for $P_{cr} \cup \{\widehat{F}\}$, then, for every substitution σ, $P \vdash F\theta\sigma$.*

(ii) *If $P_{cr} \cup \{\widehat{F}\}$ has a finitely failed SLDNF-tree, then, for every substitution σ, $P \vdash not\ F\sigma$.*

5 An example of using SLDNF-resolution

This section is dedicated to an example of using the transformation to answer queries. For this we use the program presented by Gelfond and Lifschitz in [GL90]. Consider the following program *School*:

$$
\begin{array}{l}
Eligible(x) \leftarrow HighGPA(x) \\
Eligible(x) \leftarrow Minority(x), FairGPA(x) \\
\sim Eligible(x) \leftarrow \sim FairGPA(x) \\
Interview(x) \leftarrow not\ Eligible(x), not\ \sim Eligible(x) \\
FairGPA(Ann) \leftarrow \\
\sim HighGPA(Ann) \leftarrow
\end{array}
$$

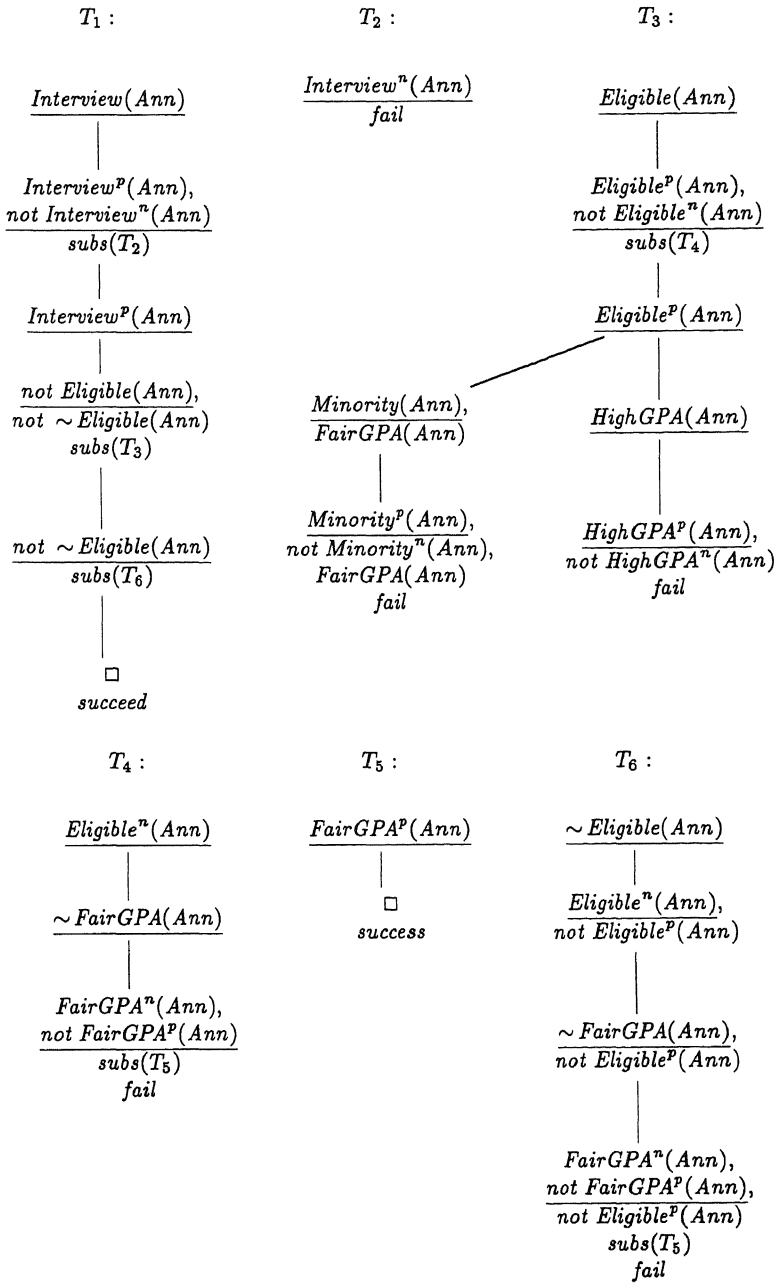
The general logic program $School_{cr}$ consists of the following clauses:

$$
\begin{array}{l}
Eligible^p(x) \leftarrow HighGPA(x) \\
Eligible^p(x) \leftarrow Minority(x), FairGPA(x) \\
Eligible^n(x) \leftarrow \sim FairGPA(x) \\
Interview^p(x) \leftarrow not\ Eligible(x), not\ \sim Eligible(x) \\
FairGPA^p(Ann) \leftarrow \\
HighGPA^n(Ann) \leftarrow
\end{array}
$$

and

$$
\begin{array}{l}
Eligible(x) \leftarrow Eligible^p(x), not\ Eligible^n(x) \\
\sim Eligible(x) \leftarrow Eligible^n(x), not\ Eligible^p(x) \\
FairGPA(x) \leftarrow FairGPA^p(x), not\ FairGPA^n(x) \\
\sim FairGPA(x) \leftarrow FairGPA^n(x), not\ FairGPA^p(x) \\
HighGPA(x) \leftarrow HighGPA^p(x), not\ HighGPA^n(x) \\
\sim HighGPA(x) \leftarrow HighGPA^n(x), not\ HighGPA^p(x) \\
Interview(x) \leftarrow Interview^p(x), not\ Interview^n(x) \\
\sim Interview(x) \leftarrow Interview^n(x), not\ Interview^p(x) \\
Minority(x) \leftarrow Minority^p(x), not\ Minority^n(x) \\
\sim Minority(x) \leftarrow Minority^n(x), not\ Minority^p(x)
\end{array}
$$

Now, consider the query $Interview(Ann)$. One of the SLDNF-trees for this query (according to the definition of SLDNF-tree given in [AD92]) is:



Here, $subs(T_i)$ denotes a "pointer" to the subsidiary tree T_i.

As we can see, we get the same answer as Gelfond and Lifschitz got with their answer set semantics. This is not very surprising. For a large class of consistent extended logic programs, completion semantics for $\widehat{P}$ and P_{cr} will coincide. In the next section we will say more about this relation between $\widehat{P}$ and P_{cr}.

6 On the relation between $\widehat{P}$ and P_{cr}

If we know that an extended logic program is consistent, the most intuitive and simple translation is the $\widehat{\ }$ translation. Therefore, we would like the *cr* translation to coincide with the $\widehat{\ }$ translation for extended logic programs that happen to be consistent.

First some good news: for consistent extended logic programs, the *cr* transformation is 'sound' with respect to the $\widehat{\ }$ transformation.

Lemma 6.1 *Let P be a consistent extended logic program and let F be a formula in the language of P. Then,* $\mathrm{comp}(P_{cr}) \models_3 \widehat{F}$ *implies* $\mathrm{comp}(\widehat{P}) \models_3 \widehat{F}$.

Note that this lemma hold also for two-valued completion. In fact, it seems reasonable to expect it to hold for any reasonable semantics for general logic programs. A conjecture to this lemma is that, for consistent extended logic programs, conservative derivability is sound with respect to answer-set semantics.

We cannot prove the converse of this lemma, as shown in the following example.

Example 6.2 Consider the extended logic program P_3:

$$\sim q \leftarrow not\ q$$
$$q \leftarrow$$

P_{3cr} is the general logic program

$$q^p \leftarrow$$
$$q^n \leftarrow not\ q$$
$$q \leftarrow q^p, not\ q^n$$
$$\sim q \leftarrow q^n, not\ q^p$$

For P_3 we have that $comp(\widehat{P}) \models_3 q$, but $comp(P_{cr}) \models_3 \neg q$ does not hold, because after some unfolding we derive $comp(P_{cr}) \models_3 q \leftrightarrow \neg q$.

Clearly, the behaviour of $\widehat{P}$ is more intuitive, and we would like P_{cr} to mimic it. ∘

This somewhat counterintuitive behaviour with respect to consistent programs also arises with the conservative derivability relation given in this paper; we can derive neither $P_3 \vdash q$ nor $P_3 \vdash not\ q$. The problem is, that

in the conservative derivability relation as defined in this paper (as well as in the relations defined by G. Wagner in [Wag91] and [Wag93]), *not* is defined as negation as *finite* failure. Because, in P_3, $\sim q$ does not fail finitely (there is a cyclic dependency between q and $\sim q$), in this system *not* $\sim q$ *should* not be derivable. A solution to this problem could be, to define a conservative derivability relation in which *not* stands for negation as (possibly infinite) failure. In such a case, we would get $P_3 \vdash q$ and $P_3 \vdash$ *not* $\sim q$. We are quite confident that such a modified system for conservative reasoning can be given, and that for such a system and for consistent extended logic programs P, we can prove soundness and completeness of conservative derivability with respect to three-valued completion of $\widehat{P}$.

With respect to such a modified conservative derivability relation, the *cr* transformation would no longer be complete. However, we can refine the transformation by omitting the consistency check for those predicates for which consistency can be proven.

Example 6.3 Consider program P_3. It is clear that the definition of q is consistent. Therefore, a consistency check on q is superfluous. So, we refine P_{3cr} to

$$\begin{array}{l} q^p \leftarrow \\ q^n \leftarrow not\ q \\ q \leftarrow q^p \\ \sim q \leftarrow q^n \end{array}$$

Clearly, q is a consequence of the completion of this program. ∘

So, we could improve the behaviour of the transformed program by analyzing the extended logic program and removing superfluous consistency checks in the transformed program.

As a final remark on this problem, we would like to stress that we do not advocate the use of the *cr* transformation for program that are *known* to be consistent. Instead, we are concerned with extended logic programs for which it is not possible or practical to prove consistency beforehand.

Apart from a mismatch between the two translations with respect to three-valued completion semantics, there is also a problem with using floundering SLDNF-resolution.

Example 6.4 Consider the extended logic program P_4:

$$q(x) \leftarrow$$

P_{4cr} is the general logic program

$$\begin{array}{l} q^p(x) \leftarrow \\ q(x) \leftarrow q^p(x), not\ q^n(x) \\ \sim q(x) \leftarrow q^n(x), not\ q^p(x) \end{array}$$

Now, consider the query $q(x)$. For P_4, this is a very simple query, which simply should be answered by *yes*. But SLDNF-resolution on P_{4cr} flounders. ∘

Note that, although in this example P_4 is a general logic program, the problem also occurs in extended logic programs that are not general logic programs.

This problem can be solved by using a form of constructive negation, instead of SLDNF-resolution. For instance, W. Drabent presented SLDFA-resolution, which uses a form of constructive negation, in [Dra92] and proved that this proof procedure is sound and complete with respect to three-valued completion semantics. So, we can use the program transformation together with SLDFA-resolution as a sound and complete proof procedure for extended logic programs.

7 Conclusion

In this paper we presented a transformation from extended logic programs to general logic programs. For this transformation we have proven that, for extended logic programs without function symbols, the three-valued completion semantics of a transformed program is sound and complete with respect to conservative derivability in the original extended logic program. As a result, we can use arbitrary proof procedures for general logic programs, as long as they are sound with respect to three-valued completion semantics. For instance, using the transformation together with SLDNF-resolution, we get a proof procedure for extended logic programs that is sound with respect to conservative derivability and using SLDFA-resolution we get a proof procedure which is sound and complete with respect to conservative derivability.

The advantage of using a transformation from extended logic programs to general logic programs, is that it gives us access to all results concerning proof procedures for general logic programs. For instance, we do not need to redo work on termination of goals.

The soundness and completeness result are restricted to extended logic programs without function symbols. The reason for this is, that the notion of conservative derivability is only defined for programs without function symbols. We believe that the notion of conservative derivability can be extended to programs with function symbols, and that with such an extended definition, we will be able to generalize the soundness and completeness result to extended logic programs with function symbols.

Aside from extending the conservative derivability relation to programs with function symbols, it might be interesting to solve the second problem mentioned in section 6, i.e. define a notion conservative reasoning in which *not* stands for negation as (possibly infinite) failure. Once we have such a system, we could use consistency analysis on the extended logic program to optimize the transformed program by omitting superfluous consistency checks, without losing soundness of the optimized general program w.r.t. conservative derivability on the original extended logic program.

Acknowledgements

This paper was supported by a grant from SION, a department of NWO, the Netherlands Organization for Scientific Research. I would like to thank Krzysztof Apt for suggesting to me to look at proof procedures for extended logic programs, for proof reading this paper and for giving valuable suggestions for improvement. Furthermore, I am grateful to the referees for their valuable comments.

References

[AD92] Krzysztof R. Apt and Kees Doets. A new definition of SLDNF-resolution. Technical Report CS-R9242, Centre for Mathematics and Computer Science, 1992. To appear in JLP.

[Dra92] Wlodzimierz Drabent. What is failure? an approach to constructive negation. Updated version of a Technical Report LITH-IDA-R-91-23 at Linkoping University, 1992.

[Fit85] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[GL90] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.

[Kun87] Kenneth Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[MNR92] V. Wiktor Marek, Anil Nerode, and Jeffrey B. Remmel. The stable models of a predicate logic program. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 446–460, 1992.

[Prz90] Teodor C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *Proceedings of the International Conference on Logic Programming*, pages 459–477, 1990.

[Wag91] Gerd Wagner. Ex contradictione nihil sequitur. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 538–543, 1991.

[Wag93] Gerd Wagner. Neutralization and preemption in extended logic programs. Technical Report Bericht Nr. 20/93, Freien Universität Berlin, 1993.