Università di Pisa

Dipartimento di Informatica

Technical Report : TR-97-02

Programming by Combining General Logic Programs

A. Brogi, S. Contiero, F. Turini

January 13, 1997

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Programming by Combining General Logic Programs

A. Brogi, S. Contiero, F. Turini

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
{brogi,contiero,turini}@di.unipi.it

Abstract

The program composition approach can be fruitfully applied to combine general logic programs, i.e. logic programs possibly containing negative premises. We show how the introduction of a basic set of (meta-level) composition operations over general programs increases the knowledge representation capabilities of logic programming for non-monotonic reasoning. Examples of modular programming, hierarchical reasoning, constraints and rules with exceptions will be illustrated. The semantics of programs and program compositions is defined in terms of three-valued logic [17]. The computational interpretation of program compositions is formalised by an equivalence preserving syntactic transformation of arbitrary program compositions into standard general programs.

Keywords: *Logic programming, program composition, non-monotonic reasoning, three-valued logic.*

1 Introduction

It is becoming more and more evident that the activity of building software is moving from writing lines of code in some programming language to writing gluing software in some meta-language to put existing programs at work together. We would like to instantiate this general principle in a setting where both the language and the meta-language for combining software components have a formal mathematical semantics, thus providing firm foundations for reasoning about programs and program compositions.

We choose logic programming as the basic language and a suite of operations on logic programs as the gluing (part of the) meta-language. The choice of logic programming is justified by two orders of reasons:

- *Generality*: Logic programming supports a wealth of programming styles developed in algorithmic programming, database programming, and artificial intelligence programming via a small number of powerful features (unification, recursion, and nondeterminism).

- *Formal semantics*: Logic programs can be formally characterised via three complementary and equivalent semantics (model-theoretic, proof-theoretic, and fixpoint).

In previous work [6, 8] we have shown how the program composition approach enhances the expressive power and knowledge representation capabilities of (definite) logic programs. The introduction of a suite of meta-level composition operations supports a wealth of applications ranging from software engineering to artificial intelligence applications. Notably the meaning of programs and program compositions can be formalised (in a compositional way) by means of the standard immediate consequence operator $T(P)$ for logic programs [9].

One of the striking features of logic programming is that it supports various forms of non-monotonic reasoning via the introduction of negation. Indeed in the applications of logic programming in the field of knowledge representation or deductive databases the need for some form of negation comes out cogently.

In this paper we apply the program composition approach to combine *general* programs, that is programs possibly containing negative premises. We consider three basic meta-level operations for composing logic programs: *union*, *intersection*, and *restriction*.

When considering general programs, the semantics of programs and program compositions cannot be defined in terms of the standard immediate consequence operator, as in the case of definite programs. Fitting [17] proposed to use a three-valued logic to formalise the meaning of general programs. We show that the three-valued immediate consequence operator is well-suited to model the semantics of program compositions in a compositional way. Namely the meaning of a composition of general programs is determined by the meaning of the separate programs.

The employment of meta-level operations over general programs brings further evidence to the applicability of logic programming to non-monotonic reasoning. We illustrate several examples from modular programming, hierarchical reasoning, constraints and rules with exceptions.

The computational interpretation of program composition operations is also presented. Instead of proposing an extended, *ad hoc* proof theory, we define a semantics preserving syntactic transformation to map arbitrary program compositions into standard general programs. The latter can thus be executed by any state-of-the-art interpreter for general programs, such as SLDNF-based ones.

The paper is organised as follows. Section 2 introduces three-valued logic for defining the semantics of general programs. A set of program composition operations is introduced in Section 3, where a syntactic transformation mapping program compositions into general programs is defined. Section 4 contains several programming examples to illustrate the actual usefulness of the proposed operations. The semantics of program compositions is formalised in Section 5, while Section 6 discusses their computational interpretation. Finally, Section 7 contains a discussion of related work as well as some concluding remarks.

2 Background: Three-valued semantics of logic programs

We now briefly recall the use of three-valued logic for defining the semantics of general logic programs, originally proposed by Fitting [17].

We will use the standard terminology and notation of logic programming [1]. A literal is an atom or its negation, and a (general) query is a finite conjunction of literals. A *general clause* is a clause of the form $A \leftarrow \overline{L}$ where A is an atom and $\overline{L}$ is a query. A *general program* is a finite set of general clauses. Following [2], we consider a universal (infinite) language in which all programs and queries are written. We denote by $\mathcal{B}$ the Herbrand base of the language and by $ground(P)$ the fully instantiated version of a program P.

Three-valued logic models the idea that a query can yield *three* possible outcomes: It may succeed, fail, or also diverge. The third value is meant to capture the last possibility. We now recall the definition of the three-valued semantics for general programs proposed by Fitting [17], following the presentation given by Apt and Bol [2].

Fitting [17] based his approach on a logic due to Kleene [19], in which three values are assumed: 0 representing *false*, 1 representing *true*, and $\frac{1}{2}$ representing *unknown.* Assuming a mapping $|\ |$ from $\mathcal{B}$ to $\{0, \frac{1}{2}, 1\}$, the meaning of (ground) formulae can be defined as follows:

$$\begin{aligned} |\neg A| &= 1 - |A| \\ |A \wedge B| &= min(|A|, |B|) \end{aligned}$$

When a ground formula evaluates to 1, we say that it is *true relative to* $|\ |$, and when it evaluates to 0, we say that it is *false relative to* $|\ |$. The mapping $|\ |$ can be conveniently presented in the form of a three-valued interpretation.

Definition 1 *A pair $I = (I^+, I^-)$, with $I^+, I^- \subseteq \mathcal{B}$, is called a* three-valued Herbrand interpretation. *I^+ are atoms assumed true, and I^- are atoms assumed false. All other atoms (in $\mathcal{B}$ but not in I) are assumed unknown. Moreover:*

- *I is* total *if $I^+ \cup I^- = \mathcal{B}$,*
- *I is* consistent *if $I^+ \cap I^- = \emptyset$.*

Note that every (two-valued) Herbrand interpretation I can be identified with the three-valued, total, consistent interpretation $(I, \mathcal{B} - I)$.

The following natural ordering (also called *information* ordering) can be defined on three-valued interpretations:

$$I \sqsubseteq J \iff (I^+ \subseteq J^+ \ \wedge \ I^- \subseteq J^-)$$

where $\subseteq$ denotes set-theoretic inclusion. Consistent three-valued interpretations form a *complete partial ordering*, that is a partial ordering in which the limits of growing chains exist.

Following Fitting [17], we now introduce the three-valued immediate consequence operator Φ.

Definition 2 *Let P be a general logic program and let I be a three-valued Herbrand interpretation. We define:*

$$\Phi(P)(I) \;=\; (T, F)$$

where

$$T = \{A \mid \; \exists\, \overline{L} : \; (A \leftarrow \overline{L}) \in ground(P) \text{ and } \overline{L} \text{ is true in } I \; \}$$

$$F = \{A \mid \; \forall\, \overline{L} : \; (A \leftarrow \overline{L}) \in ground(P) \;\; \text{implies} \;\; \overline{L} \text{ is false in } I \; \}$$

The relevant properties of the Φ operator are summarised by the following lemma.

Proposition 1 *Let P be a general logic program.*

1. *If I is a consistent three-valued interpretation then $\Phi(P)(I)$ is a consistent three-valued interpretation,*
2. *Φ is monotonic,*
3. *In general Φ is not continuous.*

The powers of Φ are defined as usual:

$$\begin{aligned} \Phi(P)(I) \uparrow 0 &= (\emptyset, \emptyset) \\ \Phi(P)(I) \uparrow (n+1) &= \Phi(P)(\Phi(P)(I) \uparrow n) \end{aligned}$$

We will use $\Phi(P) \uparrow i$ as a shorthand for $\Phi(P)(\emptyset, \emptyset) \uparrow i$.

Fitting [17] proposed the use of three-valued logic for defining the semantics of program completion [11]. Following [2], we write $I \models_3 \phi$ to indicate that a formula ϕ is true in an arbitrary three-valued interpretation I, and we write $comp(P) \models_3 Q$ to indicate that a query Q is true in all three-valued models of the completion $comp(P)$ of a program P.

An important result by Kunen [21] established that the truth of a formula w.r.t. $comp(P)$ can be determined by iterating a *finite* number of times the operator Φ starting from the empty three-valued interpretation $(\emptyset, \emptyset)$.

Lemma 2 *Let ϕ be a first-order formula not containing $\leftarrow$ and $\leftrightarrow$. Then:*

$$comp(P) \models_3 \phi \;\; \textit{iff} \;\; \exists n \; \textit{finite} : \;\; \Phi(P) \uparrow n \models_3 \phi.$$

3 General program expressions

We now introduce three basic meta-level operations for composing general logic programs: *union* (denoted by $\cup$), *intersection* (denoted by $\cap$) and *restriction* (denoted by $\prec$). Starting from a collection of (named) general logic programs, these operations define a class of *general program expressions*. Syntactically such a class is defined as follows:

$$E \ ::= \ P \ | \ E \cup E \ | \ E \cap E \ | \ E \prec E$$

where P is (the name of) a general program.

As shown in [8], any arbitrary composition of *definite* programs can be transformed into an equivalent definite program. Namely, there exists a syntactic transformation τ that maps any definite program expression E into a definite program $\tau(E)$ which is equivalent to E, that is such that for any interpretation I: $T(E)(I) = T(\tau(E))(I)$.

We now introduce the syntactic transformation τ extended to deal with general programs. The transformation is defined inductively on the structure of (general) program expressions.

Definition 3 *The transformation τ mapping general program expressions into general programs is defined as follows:*

$$\begin{array}{lll}
\tau(P) & = P & \textit{if } P \textit{ is a general logic program} \\
\tau(E \cup F) & = \{A \leftarrow \overline{L} & | \ A \leftarrow \overline{L} \in \tau(E) \ \vee \ A \leftarrow \overline{L} \in \tau(F)\} \\
\tau(E \cap F) & = \{A \leftarrow \overline{L} & | \ \exists A', A'', \overline{L'}, \overline{L''}, \vartheta : \\
& & \quad A' \leftarrow \overline{L'} \in \tau(E) \ \wedge \ A'' \leftarrow \overline{L''} \in \tau(F) \ \wedge \ \vartheta = mgu(A', A'') \ \wedge \\
& & \quad A = A'\vartheta \ \wedge \ (\overline{L} = \overline{L'}\vartheta \cup \overline{L''}\vartheta)\} \\
\tau(E \prec F) & = \{A \leftarrow \overline{L} & | \ A \leftarrow \overline{L} \in \tau(E) \ \wedge \ name(A) \notin defs(\tau(F))\}
\end{array}$$

where $name(A)$ denotes the predicate symbol of atom A, and $defs(P)$ denotes the set of predicate symbols defined in program P.

The transformation τ merely depends on the syntactic structure of the given expression, namely on the operations and on the syntax of the general programs forming it. When the program expression is already a general program then τ is just the identity function. The *union* of two programs corresponds to the set-theoretic union of the clauses of the separate programs. The *intersection* operation combines two programs by merging pair of clauses with unifiable heads into clauses having the (duplicate-free) conjunctions of the bodies of the original clauses as a body. The net effect is that the two programs act as sets of constraints one upon the other. Consider, for instance, the following two simple programs:

P :
$Likes(x, y) \leftarrow \neg Bitter(y).$
$Hates(x, y) \leftarrow Sour(y).$

and

$Q:$
$Likes(Bob, y) \leftarrow Sour(y).$

Then the program $\tau(P \cap Q)$ consists of the clause:

$Likes(Bob, y) \leftarrow \neg Bitter(y), Sour(y).$

which is obtained by joining the first clause of P with the clause of Q. Notice that the program $\tau(P \cap Q)$ does not contain a definition for predicate $Hates$, since the latter is not defined in Q.

The *restriction* operation allows one to filter the set of clauses of a program by eliminating all the clauses defining predicates which are defined in another program. Namely, given two programs P and Q, the program $\tau(P \prec Q)$ contains all the clauses of P which define predicates for which no definition is present in Q. Considering the previous two sample programs P and Q we have that the program $\tau(P \prec Q)$ consists of the clause:

$Hates(x, y) \leftarrow Sour(y).$

where the first clause of P has been filtered out.

Before defining the three-valued semantics of the program composition operations, we will present some programming examples that motivate their introduction.

4 Motivating examples

In this section we show how the operations of *union* ($\cup$), *intersection* ($\cap$) and *restriction* ($\prec$) over general logic programs actually enhance the expressive power and knowledge representation capabilities of the logic programming paradigm. We will provide evidence of this by illustrating a number of programming examples.

Modular composition of programs
As discussed in [6, 8], *union* is the most basic composition operation over logic programs. Indeed every logic program can be viewed as the (set-theoretic) union of its clauses. The union operator supports a simple modular construction of programs by putting together separate existing pieces of code. Consider for instance the program $Even$ defining even numbers:

$Even(Zero).$
$Even(S(S(x))) \leftarrow Even(x).$

The relation Odd defining odd numbers can be defined separately by suitably extending program $Even$ above. For instance, one may define program Odd as follows:

$Odd(x) \leftarrow \neg Even(x).$

and then compose the two programs by means of the program expression $Even \cup Odd$.

The general program $\tau(Even \cup Odd)$ obtained by applying the transformation τ to the expression $Even \cup Odd$ consists of the set-theoretic union of the clauses of the two programs:

$$Even(Zero).$$
$$Even(S(S(x))) \leftarrow Even(x).$$
$$Odd(x) \leftarrow \neg Even(x).$$

More sophisticated forms of modular programming, such as information hiding or import/export relationships, are supported by the program composition approach, as thoroughly illustrated in [8].

Hierarchical composition of programs

Program composition operations can be combined together to express more sophisticated forms of program composition. The definition of hierarchies in object-oriented programming is one of such examples. A well-known form of inheritance between classes is the so-called *is-a* relation. In a logic programming setting, objects may be viewed as theories (programs) [25], and inheritance relations as composition operations over them.

For instance, given two programs P and Q, the meaning of the relation P *is-a* Q can be defined as follows: P inherits all the predicate definitions from Q except for the predicates that are re-defined in P. The *is-a* relation can be expressed by means of the *union* and *restriction* operations as follows:

$$P \text{ is-a } Q \;=\; P \cup (Q \prec P).$$

The above expression states that program P is extended with the definitions of predicates in Q which are not overridden by any definition in P.

Consider for instance the following program $Person$ which roughly describes the class of persons:

$$Eats(Meat).$$
$$Eats(Fish).$$
$$Eats(Vegetables).$$
$$Lives\_in(x) \leftarrow Works\_in(x).$$

The fact that the individual $John$ is a vegetarian working in London may be represented by the program $John$:

$$Eats(Vegetables).$$
$$Works\_in(London).$$

The expression $John$ is-a $Person$ will then extend program $John$ with the predicate definitions in $Person$ which are not overridden by $John$. According to the definition of *is-a*, the program $\tau(John$ is-a $Person)$ is then:

$Eats(Vegetables).$
$Works\_in(London).$
$Lives\_in(x) \leftarrow Works\_in(x).$

Constraints

Another interesting example of programming by composing object logic programs is the following. Suppose that we have a database on which we would like to impose some further constraint. For instance suppose that we have a database DB representing connections between cities of the form:

$Path(x, y) \leftarrow Connected(x, y).$
$Path(x, y) \leftarrow Connected(x, z), Path(z, y).$
$Connected(x, y) \leftarrow Flight(x, y).$
$Connected(x, y) \leftarrow Highway(x, y).$
$Connected(x, y) \leftarrow Railway(x, y).$
$Status(Pisa, Town).$
$Status(Florence, City).$
$Status(Rome, Metropolis).$
$\ldots$
$Flight(Pisa, Rome).$
$\ldots$

The above database DB can be queried, for instance, by means of queries of the form $Path(Pisa, w)$ in order to find all cities reachable from $Pisa$.

Suppose now that we are interested in determining whether there is a path between two cities such that no metropolis is encountered during the travel. Such a constraint can be expressed by means of the *intersection* operation by constraining the definition of predicate $Connected$ so as to consider in the path only connections between non-metropolis. This may be expressed by the program expression $DB \cap C$, where C is the program:

$Connected(x, y) \leftarrow \neg Status(y, Metropolis)$

It is easy to observe that the program expression $DB \cap C$ only contains a constrained definition of predicate $Connected$, while the definitions of all other predicates in DB are lost. These can be kept in the final program by suitably exploiting the *restriction* operation.

The operation *constrains* can be hence defined as follows:

$$C \text{ constrains } DB \;=\; (DB \cap C) \cup (DB \prec C).$$

The corresponding general program $\tau((DB \cap C) \cup (DB \prec C))$ is:

$Path(x,y) \leftarrow Connected(x,y).$
$Path(x,y) \leftarrow Connected(x,z), Path(z,y).$
$Connected(x,y) \leftarrow Flight(x,y), \neg Status(y, Metropolis).$
$Connected(x,y) \leftarrow Highway(x,y), \neg Status(y, Metropolis).$
$Connected(x,y) \leftarrow Railway(x,y), \neg Status(y, Metropolis).$
$Status(Pisa, Town).$
$Status(Florence, City).$
$Status(Rome, Metropolis).$
$\ldots$
$Flight(Pisa, Rome).$
$\ldots$

Rules and exceptions

An interesting application of the derived *constrains* operator is the treatment of rules and exceptions in knowledge representation problems. Consider for instance the following set of rules defining flying birds:

$Fly(x) \leftarrow Bird(x).$
$Bird(x) \leftarrow Ostrich(x).$
$Bird(x) \leftarrow Penguin(x).$
$Bird(Tweety).$
$Ostrich(Ozzy).$

As suggested in [20], clauses with positive conclusions represent general rules, while clauses with negative conclusions represent exceptions. For instance, the knowledge that ostrichs and penguins do not fly can be expressed by the following clauses:

$\neg Fly(x) \leftarrow Ostrich(x).$
$\neg Fly(x) \leftarrow Penguin(x).$

Following [20], rules and exceptions can be transformed into general logic programs by means of a program transformation that introduces *abnormality* predicates. For instance, the above sets of rules can be transformed into the following general program:

$Fly(x) \leftarrow Bird(x), \neg Abnormal(x).$
$Bird(x) \leftarrow Ostrich(x).$
$Bird(x) \leftarrow Penguin(x).$
$Bird(Tweety).$
$Ostrich(Ozzy).$
$Abnormal(x) \leftarrow Ostrich(x).$
$Abnormal(x) \leftarrow Penguin(x).$

It is worth observing that such a compilation of rules and exceptions can be entirely described in terms of program composition operations in a concise way. Let *Rules* be a (general) program defining a set of general rules. Consider now a generic exception of the form:

$$\neg P(\bar{t}) \leftarrow Condition(\bar{t}).$$

defining some exception for a predicate P in the case $Condition(t)$ holds. Each exception of the above form is converted into a clause of the form:

$$Abnormal\_P(\bar{t}) \leftarrow Condition(\bar{t}).$$

where the new predicate *Abnormal_P* defines the situations of abnormality for predicate P. This conversion produces a program *Abnormalities* containing the set of clauses obtained in this way. Moreover, for each predicate *Abnormal_P*, a clause of the form:

$$P(x) \leftarrow \neg Abnormal\_P(x).$$

is introduced, and all such clauses are grouped in a program *Ab_Constraints*. Rules and exceptions are then combined together by means of the program expression:

$$(Ab\_Constraints \text{ constrains } Rules) \cup Abnormalities$$

that is, unfolding the definition of constrains:

$$((Rules \cap Ab\_Constraints) \cup (Rules \prec Ab\_Constraints)) \cup Abnormalities.$$

In the example, the treatment of the two exceptions

$$\neg Fly(x) \leftarrow Ostrich(x).$$
$$\neg Fly(x) \leftarrow Penguin(x).$$

generates the program *Abnormalities*:

$$Abnormal\_Fly(x) \leftarrow Ostrich(x).$$
$$Abnormal\_Fly(x) \leftarrow Penguin(x).$$

and the program *Ab_Constraints*:

$$Fly(x) \leftarrow \neg Abnormal\_Fly(x).$$

It is worth observing that the general program

$$\tau(((Rules \cap Ab\_Constraints) \cup (Rules \prec Ab\_Constraints)) \cup Abnormalities)$$

obtained by applying the transformation τ coincides with the program given in [20].

5 Three-valued semantics of program expressions

The standard three-valued immediate consequence operator Φ, given a general program and a three-valued interpretation, yields a three-valued interpretation. We now extend Φ so that its first argument is an arbitrary general program expression. Therefore Φ given a general program expression E and a three-valued interpretation yields the three-valued interpretation that defines the immediate consequences of E.

Notice that we properly extend the definition of Φ in the sense that when the program expression is a plain general program then the extended definition coincides with the standard definition of Φ for general programs. In the following, we will use the notation $\Phi^+(E)(I)$ and $\Phi^-(E)(I)$ to denote the positive part and the negative part, respectively, of the three-valued interpretation $\Phi(E)(I)$.

Definition 4 *Let E_1 and E_2 be two program expressions, and let I be a three-valued interpretation. Then:*

$$\begin{aligned}
\Phi(E_1 \cup E_2)(I) &= (\Phi^+(E_1)(I) \cup \Phi^+(E_2)(I), \quad \Phi^-(E_1)(I) \cap \Phi^-(E_2)(I)) \\
\Phi(E_1 \cap E_2)(I) &= (\Phi^+(E_1)(I) \cap \Phi^+(E_2)(I), \quad \Phi^-(E_1)(I) \cup \Phi^-(E_2)(I)) \\
\Phi(E_1 \prec E_2)(I) &= (\Phi^+(E_1)(I) - \Phi^+(E_2)(\mathcal{B},\mathcal{B}), \quad \Phi^-(E_1)(I) \cup \Phi^+(E_2)(\mathcal{B},\mathcal{B}))
\end{aligned}$$

The definition for the *union* operator ($\cup$) states that the atoms assumed true for the expression $E_1 \cup E_2$ and the interpretation I are the atoms assumed true for E_1 and I *or* for E_2 and I. In a dual manner, the atoms assumed false for the expression $E_1 \cup E_2$ and the interpretation I are the atoms which are assumed false for E_1 and I *and* for E_2 and I.

The duality of the *union* and *intersection* operations is reflected by the corresponding definition of the Φ operator. Indeed, in the case of the *intersection* operator ($\cap$), the atoms assumed true for the expression $E_1 \cap E_2$ and the interpretation I are the atoms assumed true for E_1 and I *and* for E_2 and I, while the atoms assumed false for the $E_1 \cap E_2$ and I are the atoms which are assumed false for E_1 and I *or* for E_2 and I.

As far as the *restriction* operation $\prec$ is concerned, the atoms assumed true for $E_1 \prec E_2$ and I are all the atoms assumed true for E_1 and I but those for which there exists a definition in E_2. Namely, the latter can be defined as the atoms which are assumed true in E_2 taking the whole Herbrand base $\mathcal{B}$ as the input interpretation for Φ. In a dual manner, the atoms assumed false in $E_1 \prec E_2$ and I are all the atoms assumed false for E_1 along with all the atoms for which there exists a definition in E_2.

Definition 4 provides a compositional semantics for general program expressions. The compositionality property is especially important in a program composition setting like ours, as we shall discuss further in Section 7. It is important to observe that all the properties of the standard Φ operator (Proposition 1 of Sect. 2) continue to hold for the extended definition.

Proposition 3 *Let E be a general program expression.*

1. *If I is a consistent three-valued interpretation then $\Phi(E)(I)$ is a consistent three-valued interpretation,*
2. *Φ is monotonic,*
3. *In general Φ is not continuous.*

Proof.
1. To show that $\Phi(E)(I)$ is a consistent interpretation, we show that:

$$(A \in \Phi^{-}(E)(I) \implies A \notin \Phi^{+}(E)(I)).$$

The proof is by structural induction on E.

Base case
(E plain general program) By Proposition 1 (Lemma 4.11 in [2]).
Inductive cases
($E = E_1 \cup E_2$)

$$\begin{array}{ll}
 & A \in \Phi^{-}(E_1 \cup E_2)(I) \\
\iff & \{ \text{ by definition of } \Phi^{-} \ \} \\
 & A \in (\Phi^{-}(E_1)(I) \cap \Phi^{-}(E_2)(I)) \\
\iff & \{ \text{ by definition of set-theoretic intersection } \} \\
 & A \in \Phi^{-}(E_1)(I) \ \wedge \ A \in \Phi^{-}(E_2)(I) \\
\implies & \{ \text{ by inductive hypothesis on } E_1 \text{ and } E_2 \} \\
 & A \notin \Phi^{+}(E_1)(I) \ \wedge \ A \notin \Phi^{+}(E_2)(I) \\
\iff & \{ \text{ by definition of set theoretic union } \} \\
 & A \notin (\Phi^{+}(E_1) \cup \Phi^{+}(E_2)(I)) \\
\iff & \{ \text{ by definition of } \Phi^{+} \ \} \\
 & A \notin \Phi^{+}(E_1 \cup E_2)(I)
\end{array}$$

($E = E_1 \cap E_2$)

$$\begin{array}{ll}
 & A \in \Phi^{-}(E_1 \cap E_2)(I) \\
\iff & \{ \text{ by definition of } \Phi^{-} \ \} \\
 & A \in (\Phi^{-}(E_1)(I) \cup \Phi^{-}(E_2)(I)) \\
\iff & \{ \text{ by definition of set-theoretic union } \} \\
 & A \in \Phi^{-}(E_1)(I) \ \vee \ A \in \Phi^{-}(E_2)(I) \\
\implies & \{ \text{ by inductive hypothesis on } E_1 \text{ and } E_2 \} \\
 & A \notin \Phi^{+}(E_1)(I) \ \vee \ A \notin \Phi^{+}(E_2)(I) \\
\iff & \{ \text{ by definition of set theoretic intersection } \} \\
 & A \notin (\Phi^{+}(E_1) \cap \Phi^{+}(E_2)(I)) \\
\iff & \{ \text{ by definition of } \Phi^{+} \ \}
\end{array}$$

$A \notin \Phi^+(E_1 \cap E_2)(I)$

($E = E_1 \prec E_2$)

$A \in \Phi^-(E_1 \prec E_2)(I)$

$\iff$ { by definition of Φ^- }

$A \in (\Phi^-(E_1)(I) \cup \Phi^+(E_2)(\mathcal{B}, \mathcal{B}))$

$\iff$ { by definition of set-theoretic union }

$A \in \Phi^-(E_1)(I) \ \vee \ A \in \Phi^+(E_2)(\mathcal{B}, \mathcal{B})$

$\Longrightarrow$ { by inductive hypothesis on E_1 }

$A \notin \Phi^+(E_1)(I) \ \vee \ A \in \Phi^+(E_2)(\mathcal{B}, \mathcal{B})$

$\iff$ { by definition of set-theoretic difference }

$A \notin (\Phi^+(E_1)(I) - \Phi^+(E_2)(\mathcal{B}, \mathcal{B}))$

$\iff$ { by definition of Φ^+ }

$A \notin \Phi^+(E_1 \prec E_2)(I)$

2. We now prove that Φ is monotonic. We show that for any general program expression E and for any three-valued interpretations I and J:

$$I \sqsubseteq J \implies \Phi(E)(I) \sqsubseteq \Phi(E)(J).$$

The proof is by structural induction on the program expression E.

Base case

(E plain general program) By Proposition 1 (Lemma 4.11 in [2]).

Inductive cases

($E = E_1 \cup E_2$)

$\Phi(E_1 \cup E_2)(I)$

$=$ { by definition of Φ }

$(\Phi^+(E_1)(I) \cup \Phi^+(E_2)(I)\ ,\ \Phi^-(E_1)(I) \cap \Phi^-(E_2)(I)$

$\sqsubseteq$ { by inductive hypothesis on E_1 and E_2 }

$(\Phi^+(E_1)(J) \cup \Phi^+(E_2)(J)\ ,\ \Phi^-(E_1)(J) \cap \Phi^-(E_2)(J)$

$=$ { by definition of Φ }

$\Phi(E_1 \cup E_2)(J)$

($E = E_1 \cap E_2$) Analogous to the previous case.

($E = E_1 \prec E_2$)

$\Phi(E_1 \prec E_2)(I)$

$=$ { by definition of Φ }

$(\Phi^+(E_1)(I) - \Phi^+(E_2)(\mathcal{B}, \mathcal{B})\ ,\ \Phi^-(E_1)(I) \cup \Phi^+(E_2)(\mathcal{B}, \mathcal{B}))$

$\sqsubseteq$ { by inductive hypothesis on E_1 }

$(\Phi^+(E_1)(J) - \Phi^+(E_2)(\mathcal{B}, \mathcal{B})\ ,\ \Phi^-(E_1)(J) \cup \Phi^+(E_2)(\mathcal{B}, \mathcal{B}))$

$=$ { by definition of Φ }

$\Phi(E_1 \prec E_2)(J)$

3. To show that Φ in in general not continuous it suffices to refer to the case of plain general programs (Proposition 1). $\diamond$

6 Computing with general program expressions

We now discuss how general program expressions actually compute. Rather than defining an extended, *ad hoc* proof procedure for dealing with compositions of general programs, we will show how the general program expressions can be transformed into equivalent general programs and then executed by any existing interpreter for general programs.

Namely, we show that the transformation τ introduced in Section 3 preserves the three-valued semantics defined by Φ. More precisely, we prove that every general program expression E is equivalent w.r.t. Φ to the transformed general program $\tau(E)$, as stated by the following theorem.

Theorem 4 *Let E be a general program expression. Then for any three-valued Herbrand interpretation I:*

$$\Phi(E)(I) \ = \ \Phi(\tau(E))(I).$$

Proof.

We will exploit the following observation, which trivially holds by definition of τ.

Observation (*): Let E_1, E_2 be general program expressions. Then:

$$\begin{aligned}
ground(\tau(E_1 \cup E_2)) &= \{A \leftarrow \overline{L} \mid (A \leftarrow \overline{L}) \in ground(\tau(E_1)) \ \vee \ (A \leftarrow \overline{L}) \in ground(\tau(E_2))\} \\
ground(\tau(E_1 \cap E_2)) &= \{A \leftarrow \overline{L} \mid (A \leftarrow \overline{B}) \in ground(\tau(E_1)) \ \wedge \ (A \leftarrow \overline{C}) \in ground(\tau(E_2)) \\
&\qquad \wedge \ \overline{L} = \overline{B} \cup \overline{C} \\
ground(\tau(E_1 \prec E_2)) &= \{A \leftarrow \overline{L} \mid (A \leftarrow \overline{L}) \in ground(\tau(E_1)) \ \wedge \ name(A) \notin defs(\tau(E_2))\}
\end{aligned}$$

The proof of the theorem is by induction on the structure of E.

Base case

(E plain general program) Immediate since $\tau(E) = E$.

Inductive cases

($E = E_1 \cup E_2$)

$$\begin{array}{ll}
(i) & A \in \Phi^+(E_1 \cup E_2)(I) \\
\Longleftrightarrow & \{ \text{ by definition of } \Phi^+ \ \} \\
 & A \in (\Phi^+(E_1)(I) \cup \Phi^+(E_2)(I)) \\
\Longleftrightarrow & \{ \text{ by definition of set theoretic union } \} \\
 & A \in \Phi^+(E_1)(I) \ \vee \ A \in \Phi^+(E_2)(I) \\
\Longleftrightarrow & \{ \text{ by inductive hypothesis on } E_1 \text{ and } E_2 \ \} \\
 & A \in \Phi^+(\tau(E_1))(I) \vee A \in \Phi^+(\tau(E_2))(I) \\
\Longleftrightarrow & \{ \text{ by definition of } \Phi^+ \ \}
\end{array}$$

$(\exists \overline{L_1} :\ (A \leftarrow \overline{L_1}) \in ground(\tau(E_1)) \wedge \overline{L_1}$ is true in $I) \vee$
$(\exists \overline{L_2} :\ (A \leftarrow \overline{L_2}) \in ground(\tau(E_2)) \wedge \overline{L_2}$ is true in $I)$

$\iff$ { by observation (*) }
$\exists \overline{L} :\ (A \leftarrow \overline{L}) \in ground(\tau(E_1 \cup E_2)) \wedge \overline{L}$ is true in I

$\iff$ { by definition of Φ^+ }
$A \in \Phi^+(\tau(E_1 \cup E_2))(I)$

(ii) $A \in \Phi^-(E_1 \cup E_2)(I)$

$\iff$ { by definition of Φ^- }
$A \in (\Phi^-(E_1)(I) \cap \Phi^-(E_2)(I))$

$\iff$ { by definition of set theoretic intersection }
$A \in \Phi^-(E_1)(I)\ \wedge\ A \in \Phi^-(E_2)(I)$

$\iff$ { by inductive hypothesis on E_1 and E_2 }
$A \in \Phi^-(\tau(E_1))(I) \wedge A \in \Phi^-(\tau(E_2))(I)$

$\iff$ { by definition of Φ^- }
$(\forall \overline{L_1} :\ (A \leftarrow \overline{L_1}) \in ground(\tau(E_1)) \Longrightarrow \overline{L_1}$ is false in $I)\ \vee$
$(\forall \overline{L_2} :\ (A \leftarrow \overline{L_2}) \in ground(\tau(E_2)) \Longrightarrow \overline{L_2}$ is false in $I)$

$\iff$ { by observation (*) }
$\forall \overline{L} :\ (A \leftarrow \overline{L}) \in ground(\tau(E_1) \cup \tau(E_2))) \Longrightarrow \overline{L}$ is false in I

$\iff$ { by definition of Φ^- }
$A \in \Phi^-(\tau(E_1 \cup E_2))(I)$

($E = E_1 \cap E_2$)

(i) $A \in \Phi^+(E_1 \cap E_2)(I)$

$\iff$ { by definition of Φ^+ }
$A \in (\Phi^+(E_1)(I) \cap \Phi^+(E_2)(I))$

$\iff$ { by definition of set theoretic intersection }
$A \in \Phi^+(E_1)(I)\ \wedge\ A \in \Phi^+(E_2)(I)$

$\iff$ { by inductive hypothesis on E_1 and E_2 }
$A \in \Phi^+(\tau(E_1))(I)\ \wedge\ A \in \Phi^+(\tau(E_2))(I)$

$\iff$ { by definition of Φ^+ }
$(\exists \overline{L_1} :\ (A \leftarrow \overline{L_1}) \in ground(\tau(E_1)) \wedge \overline{L_1}$ is true in $I)\ \wedge$
$(\exists \overline{L_2} :\ (A \leftarrow \overline{L_2}) \in ground(\tau(E_2)) \wedge \overline{L_2}$ is true in $I)$

$\iff$ { by observation (*) }
$\exists \overline{L} :\ (A \leftarrow \overline{L}) \in ground(\tau(E_1 \cap E_2)) \wedge \overline{L}$ is true in I

$\iff$ { by definition of Φ^+ }
$A \in \Phi^+(\tau(E_1 \cap E_2))(I)$

(ii) $A \in \Phi^-(E_1 \cap E_2)(I)$

$\iff$ { by definition of Φ^- }
$A \in (\Phi^-(E_1)(I) \cup \Phi^-(E_2)(I))$

$\iff$ { by definition of set theoretic union }

$A \in \Phi^-(E_1)(I) \ \vee \ A \in \Phi^-(E_2)(I)$

$\Longleftrightarrow$ { by inductive hypothesis on E_1 and E_2 }

$A \in \Phi^-(\tau(E_1))(I) \ \vee \ A \in \Phi^-(\tau(E_2))(I)$

$\Longleftrightarrow$ { by definition of Φ^- }

$((\forall \overline{L_1}:\ (A \leftarrow \overline{L_1}) \in ground(\tau(E_1))) \Longrightarrow \overline{L_1}$ is false in $I) \ \vee$

$((\forall \overline{L_2}:\ (A \leftarrow \overline{L_2}) \in ground(\tau(E_2))) \Longrightarrow \overline{L_2}$ is false in $I)$

$\Longleftrightarrow$ { by observation (*) }

$\forall \overline{L}:\ (A \leftarrow \overline{L}) \in ground(\tau(E_1 \cap E_2)) \ \Longrightarrow \ \overline{L}$ is false in I

$\Longleftrightarrow$ { by definition of Φ^- }

$A \in \Phi^-(\tau(E_1 \cap E_2))(I)$

($E = E_1 \prec E_2$)

(i) $A \in \Phi^+(E_1 \prec E_2)(I)$

$\Longleftrightarrow$ { by definition of Φ^+ }

$A \in (\Phi^+(E_1)(I) \ - \ \Phi^+(E_2)(\mathcal{B},\mathcal{B}))$

$\Longleftrightarrow$ { by definition of set theoretic difference }

$A \in \Phi^+(E_1)(I) \ \wedge \ A \notin \Phi^+(E_2)(\mathcal{B},\mathcal{B})$

$\Longleftrightarrow$ { by inductive hypothesis on E_1 and E_2 }

$A \in \Phi^+(\tau(E_1))(I) \ \wedge \ A \notin \Phi^+(\tau(E_2))(\mathcal{B},\mathcal{B})$

$\Longleftrightarrow$ { by definition of Φ^+ }

$\exists \overline{L}:\ (A \leftarrow \overline{L}) \in ground(\tau(E_1)) \ \wedge \ \overline{L}$ is true in $I \ \wedge \ A \notin \Phi^+(\tau(E_2))(\mathcal{B},\mathcal{B}))$

$\Longleftrightarrow$ { by definition of Φ }

$\exists \overline{L}: (A \leftarrow \overline{L}) \in ground(\tau(E_1)) \wedge \overline{L}$ is true in $I \ \wedge \ name(A) \notin defs(\tau(E_2))$

$\Longleftrightarrow$ { by observation (*) }

$\exists \overline{L}: (A \leftarrow \overline{L}) \in ground(\tau(E_1 \prec E_2)) \wedge \overline{L}$ is true in I

$\Longleftrightarrow$ { by definition of Φ^+ }

$A \in \Phi^+(\tau(E_1 \prec E_2))(I)$

(ii) $A \in \Phi^-(E_1 \prec E_2)(I)$

$\Longleftrightarrow$ { by definition of Φ^- }

$A \in (\Phi^-(E_1)(I) \cup \Phi^+(E_2)(\mathcal{B},\mathcal{B}))$

$\Longleftrightarrow$ { by definition of set theoretic union }

$A \in \Phi^-(E_1)(I) \ \vee \ A \in \Phi^+(E_2)(\mathcal{B},\mathcal{B})$

$\Longleftrightarrow$ { by inductive hypothesis on E_1 and E_2 }

$A \in \Phi^-(\tau(E_1))(I) \ \vee \ A \in \Phi^+(\tau(E_2))(\mathcal{B},\mathcal{B})$

$\Longleftrightarrow$ { by definition of Φ^- }

$(\forall \overline{L}:\ (A \leftarrow \overline{L}) \in ground(\tau(E_1)) \Longrightarrow \overline{L}$ is false in $I) \ \vee \ A \in \Phi^+(\tau(E_2))(\mathcal{B},\mathcal{B})$

$\Longleftrightarrow$ { by definition of Φ }

$(\forall \overline{L}: (A \leftarrow \overline{L}) \in ground(\tau(E_1)) \Longrightarrow \overline{L}$ is false in $I) \ \vee \ name(A) \in defs(\tau(E_2))$

$\Longleftrightarrow$ { by observation (*) }

$(\forall \overline{L}: (A \leftarrow \overline{L}) \in ground(\tau(E_1 \prec E_2)) \Longrightarrow \overline{L}$ is false in $I)$

$\iff$ { by definition of Φ^- }
$A \in \Phi^-(\tau(E_1 \prec E_2))(I)$

$\diamond$

By virtue of the above equivalence result, the three-valued fixpoint semantics of general program expressions can be related to the operational semantics for general programs proposed in the literature.

SLDNF-resolution has been proved to be sound w.r.t. three-valued semantics (e.g., see [13, 28]). Let us consider the definition of SLDNF-derivation and SLDNF-tree introduced in [3] to amend a problem with the definitions given in [24].

By virtue of Lemma 2, the soundness result can be formulated as follows.

Theorem 5 *Let P be a general program and G a query. Then:*

1. *If ϑ is a computed answer substitution (c.a.s.) for G in P then there exists a finite n s.t. $\Phi(P) \uparrow n \models_3 \forall G\vartheta$.*

2. *If there exists a finitely failed SLDNF-tree for G in P then there exists a finite n s.t. $\Phi(P) \uparrow n \models_3 \forall \neg G$.*

Theorem 4 ensures that the transformation τ preserves the three-valued semantics of program expressions, and hence allows us to extend Theorem 5 to the case of general program expressions.

Corollary 6 *Let E be a general program expression and let G be a query. Then:*

1. *If ϑ is a c.a.s. for G in $\tau(E)$ then there exists a finite n s.t. $\Phi(E) \uparrow n \models_3 \forall G\vartheta$.*

2. *If there exists a finitely failed SLDNF-tree for G in $\tau(E)$ then there exists a finite n s.t. $\Phi(E) \uparrow n \models_3 \forall \neg G$.*

The completeness of SLDNF-resolution w.r.t. three-valued semantics has been established only for specific classes of general programs. Let us recall the definition of one of those classes, the class of *allowed* programs [23].

Definition 5

1. *A query $\overline{L}$ is called* allowed *if* every *variable of it occurs in a positive literal.*

2. *A clause $H \leftarrow \overline{L}$ is called* allowed *if $\neg H, \overline{L}$ is allowed.*

3. *A program is called allowed if all its clauses are allowed.*

Kunen [22] proved the following completeness result of SLDNF-resolution w.r.t. the three-valued semantics for the class of allowed programs.

Theorem 7 *Let P be an allowed general program and let G be an allowed query. Then:*

1. *If there exists a finite n s.t. $\Phi(P) \uparrow n \models_3 \forall G\vartheta$ then ϑ is a c.a.s. for G in P.*

2. *If there exists a finite n s.t. $\Phi(P) \uparrow n \models_3 \forall \neg G$ then G has a finitely failed SLDNF-tree in P.*

As in the case of the soundness result, Theorem 4 can be exploited to extend Theorem 7 to the case of general program expressions. To do this, it is first necessary to prove that the class of allowed programs is closed with respect to the program composition operations introduced.

Proposition 8 *Let E be a program expression over* allowed *general programs. Then $\tau(E)$ is an* allowed *general program.*

Proof.
The proof is by structural induction on E. We denote by $Vars(L)$ the set of variables occurring in a literal L. Moreover, we denote by $pos(\overline{L})$ and by $neg(\overline{L})$ the positive and negative literals of a conjunction of literals $\overline{L}$. The, by Definition 5, we have that $A \leftarrow \overline{L}$ is allowed iff $(Vars(A) \cup Vars(neg(\overline{L})) \subseteq Vars(pos(\overline{L}))$.

Base case
(E plain general program) Trivial, since $\tau(E) = E$.
Inductive cases
Suppose, by inductive hypothesis, that $\tau(E_1)$ and $\tau(E_2)$ are allowed general programs.
($E = E_1 \cup E_2$)

$$
\begin{array}{ll}
 & A \leftarrow \overline{L} \in \tau(E_1 \cup E_2) \\
\Longleftrightarrow & \{ \text{ by definition of } \tau \ \} \\
 & A \leftarrow \overline{L} \in \tau(E_1) \ \vee \ A \leftarrow \overline{L} \in \tau(E_2) \\
\Longrightarrow & \{ \text{ by inductive hypothesis on } E_1 \text{ and } E_2 \ \} \\
 & A \leftarrow \overline{L} \text{ is allowed}
\end{array}
$$

($E = E_1 \cap E_2$)

$$
\begin{array}{ll}
 & A \leftarrow \overline{L} \in \tau(E_1 \cap E_2) \\
\Longleftrightarrow & \{ \text{ by definition of } \tau \ \} \\
 & \exists A_1, A_2, \overline{L_1}, \overline{L_2}, \vartheta : \\
 & \quad A_1 \leftarrow \overline{L_1} \in \tau(E_1) \ \wedge \ A_2 \leftarrow \overline{L_2} \in \tau(E_2) \ \wedge \ \vartheta = mgu(A_1, A_2) \ \wedge \\
 & \quad A = A_1\vartheta \ \wedge \ (\overline{L} = \overline{L_1}\vartheta \cup \overline{L_2}\vartheta) \\
\Longrightarrow & \{ \text{ by inductive hypothesis on } E_1 \text{ and } E_2 \} \\
 & \exists A_1, A_2, \overline{L_1}, \overline{L_2}, \vartheta :
\end{array}
$$

$$(Vars(A_1) \cup Vars(neg(\overline{L_1}))) \subseteq Vars(pos(\overline{L_1})) \wedge$$
$$(Vars(A_2) \cup Vars(neg(\overline{L_2}))) \subseteq Vars(pos(\overline{L_2})) \wedge$$
$$\vartheta = mgu(A_1, A_2) \ \wedge \ A = A_1\vartheta \ \wedge \ (\overline{L} = \overline{L_1}\vartheta \cup \overline{L_2}\vartheta)$$

$\Longrightarrow$ $\{$ since $\forall\vartheta : \ (Vars(C) \subseteq Vars(D) \Longrightarrow Vars(C\vartheta) \subseteq Vars(D\vartheta))$ $\}$

$$\exists A_1, \overline{L_1}, \overline{L_2}, \vartheta :$$
$$A = A_1\vartheta \ \wedge \ (\overline{L} = \overline{L_1}\vartheta \cup \overline{L_2}\vartheta) \wedge$$
$$(Vars(A_1\vartheta) \cup Vars(neg(\overline{L_1\vartheta})) \cup Vars(neg(\overline{L_2\vartheta})))$$
$$\subseteq \ (Vars(pos(\overline{L_1\vartheta})) \cup Vars(pos(\overline{L_2\vartheta})))$$

$\Longrightarrow$ $\{$ since $A = A_1\vartheta$ and $(\overline{L} = \overline{L_1}\vartheta \cup \overline{L_2}\vartheta)$ $\}$

$A \leftarrow \overline{L}$ is allowed

($E = E_1 \prec E_2$)

$A \leftarrow \overline{L} \in \tau(E_1 \prec E_2)$

$\Longrightarrow$ $\{$ by definition of τ $\}$

$A \leftarrow \overline{L} \in \tau(E_1)$

$\Longrightarrow$ $\{$ by inductive hypothesis on E_1 $\}$

$A \leftarrow \overline{L}$ is allowed

$\diamond$

Corollary 9 *Let E be an allowed general program expression and let G be an allowed query. Then:*

1. *If there exists a finite n s.t. $\Phi(E) \uparrow n \models_3 \forall G\vartheta$ then ϑ is a c.a.s. for G in $\tau(E)$.*

2. *If there exists a finite n s.t. $\Phi(E) \uparrow n \models_3 \forall \neg G$ then G has a finitely failed SLDNF-tree in $\tau(E)$.*

7 Concluding remarks

We have shown how the program composition approach can be fruitfully applied for combining general logic programs. More precisely, we have shown how the introduction of three basic meta-level composition operations over general programs (*union*, *intersection* and *restriction*) enhances the knowledge representation capabilities of logic programming for non-monotonic reasoning. We have provided general program expressions with a compositional three-valued semantics, based on the immediate consequence operator introduced by Fitting [17].

The computational interpretation of program compositions has been formalised by means of an equivalence preserving syntactic transformation τ, which maps arbitrary program compositions into standard general programs. The latter can be then executed by any state-of-the-art interpreter for general programs. In Section 6, we have shown how existing soundness and completeness result of SLDNF-resolution w.r.t. three-valued logic can be extended to the case of general program expressions. It is worth noting that, as the transformation τ maps program expressions into standard general programs, the completeness result given in Corollary 9 may be generalised to larger classes of programs like, for instance, the class proposed by Cavedon and Decker in [10], where (slightly) more general floundering-preventing conditions are introduced. Similarly, the strong completeness result by Drabent [14] may be inherited in our setting.

We have shown that Fitting's three-valued immediate consequence operator is well-suited for defining compositional semantics of general program expressions. It is worth observing that such a property does not hold for other existing semantics for general programs. For instance, supported models [4] require the object program to be stratified. It is easy to see that the union of two stratified programs is not stratified in general, that is the class of stratified programs is not closed under the operation of union of programs. Similar considerations apply to the case of perfect model semantics and locally stratified programs [26].

Other well-known semantics, such as the stable models semantics [18] or the well-founded models semantics [29], do not address compositionality. For instance, it is easy to see that the stable models of a general program cannot be determined by the stable models of its clauses. Consider the two propositional programs $A \leftarrow$ and $A \leftarrow \neg B$ which have the same (unique) stable model $\{A\}$. If these programs are extended with the clause $B \leftarrow$ we obtain two programs which have different stable models ($\{A, B\}$ and $\{B\}$, respectively). Similar considerations apply to the case of the well-founded models semantics.

It is worth noting that the compositionality of Fitting's three-valued semantics has been established in terms of the three-valued immediate consequence operator Φ used to compute the models of a program, rather than in terms of the three-valued models themselves. The need of resorting to second order semantics for defining compositional semantics for logic programs was already pointed out for the case of definite programs. Brogi and Turini [9] showed that the equivalence relation induced by the immediate consequence operator $T(P)$ is the fully abstract compositional equivalence relation for an algebra of logic programs containing several composition operations over definite programs. In this perspective the compositionality of stable models or well-founded models could be studied in terms of second order functions used for computing the models of programs. For instance, one could consider the fixpoint operator used for computing the well-founded model of a program. However, differently from the case of Fitting's Φ operator, there is no simple composition operation on fixpoint operators which is homomorphic to the operation of union on programs.

Some compositionality results for extended logic programs, including general programs, were established by Brogi et al. in [7]. The authors showed that many semantics for general programs (such as stable models, well-founded models, stationary expansions [27], complete scenaria [15]) can be defined in terms of the Herbrand models of (the positive version of) a program. The compositionality of Herbrand models w.r.t. the union of programs then induces general compositionality results for the various semantics considered. While these results can be applied to different semantic frameworks, they consider only the operation of union between programs. Unfortunately, as shown for instance in [9], the Herbrand model semantics is not compositional for other operations, like the *intersection* or *restriction* operations considered in this paper.

To the best of our knowledge, not many other efforts have been devoted to investigate the composition of general logic programs. A related work is the paper by Barbuti et al. [5], where a set of operations for composing logic programs were studied in the context of *intensional negation*. While some of the operations are the same (namely *union* and *intersection*) the approach presented in [5] is quite different from ours. In [5] negation is handled in a constructive way by transforming programs into pairs of definite programs. Then union and intersection are defined over these pairs, and the semantics is given with respect to the completion of the original general programs.

Recently, Etalle and Teusink [16] proposed a compositional semantics for general programs by considering the union of programs as the only composition operation. Differently from our case, where separate programs can fully cooperate via the union operation, a quite harsh composition policy is considered in that the definition of a predicate cannot be spread over different programs.

Future work will be devoted to extend the three-valued semantics proposed in this paper to include other composition operations in the lines of [6], and to study other properties of the semantics in the lines of [9, 12]. Another interesting research direction is to further investigate the compositionality of other semantics for general programs, and their usability in a program composition setting.

References

[1] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[2] K. R. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[3] K. R. Apt and K. Doets. A New Definition of SLDNF-resolution. *Journal of Logic Programming*, 18:177–190, 1994.

[4] K.R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Alto, CA, 1988.

[5] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A Transformational Approach to Negation in Logic Programming. *Journal of Logic Programming*, 8:201–228, 1990.

[6] A. Brogi. *Program Construction in Computational Logic.* PhD thesis, University of Pisa, 1993.

[7] A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A unifying view for logic programming with non-monotonic reasoning. *Theoretical Computer Science*, 1997. To appear.

[8] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[9] A. Brogi and F. Turini. Fully abstract compositional semantics for an algebra of logic programs. *Theoretical Computer Science*, 149(2):201–229, 1995.

[10] L. Cavedon and H. Decker. Generalizing allowedness while retaining completeness of SLDNF-resolution. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *Proceedings of the Third Workshop on Computer Science Logic*, number 440 in LNCS, pages 98–115. Springer Verlag, 1990.

[11] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum, 1978.

[12] J. Dix. Semantics of logic programs: Intuition and formal properties. In A. Furhmann and K. Rott, editors, *Logic, Action and Information.* De Gruyter, 1993.

[13] H.C. Doets. Leviatonis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.

[14] W. Drabent. Completeness of SLDNF-resolution for Non-floundering Queries. *Journal of Logic Programming*, 27(2):89–106, 1994.

[15] P.M. Dung. Negation as hypotheses: An abductive foundation for logic programming. In K. Furukawa, editor, *Proc. 8th International Conference on Logic Programming*, pages 3–17. The MIT Press, 1991.

[16] S. Etalle and F. Teusink. A Compositional Semantics for Normal Open Programs. In M. Maher, editor, *JICSLP 96 Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 468–482, 1996.

[17] M. Fitting. A Kriple-Kleene semantics for general logic programs. *Journal of Logic Programming*, 4:295–312, 1985.

[18] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming, Seattle, 1988*, pages 1070–1080. The MIT Press, 1988.

[19] S.C. Kleene. *Introduction to Metamathematics.* van Nostrand, New York, 1952.

[20] R.A. Kowalski. Problems and Promises of Computational Logic. In J.W. Lloyd, editor, *Computational Logic, Symposium Proceedings*, pages 1–36. Springer-Verlag, 1990.

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

[22] K. Kunen. Signed data dependencies in programs. *Journal of Logic Programming*, 7:231–246, 1989.

[23] J. W. Lloyd and R. W. Topor. A Basis for Deductive Database Systems II. *Journal of Logic Programming*, 3(1):55–67, 1986.

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

[25] F.G. McCabe. *Logic and Objects.* PhD thesis, Imperial College of Science and Technology, November 1988.

[26] T. Przymusinski. On the declarative semantics of logic programs with negation. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, Los Alto, CA, 1988.

[27] T. Przymusinski. Stationary semantics for normal and disjunctive logic programs. In C. Delobel, M. Kifer, and Y. Masunagar, editors, *Proceedings of DOOD'91.* Springer-Verlag, 1991.

[28] J.C. Shepherdson. Negation in logic programming for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, Los Alto, CA, 1988.

[29] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.