A Compositional Semantics for Normal Open Programs

Sandro Etalle
D.I.S.I, Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
sandro@disi.unige.it

Frank Teusink
Center for Mathematics and Computer Science
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
frankt@cwi.nl

Abstract

Modular programs are built as a combination of separate modules, which may be developed and verified separately. Therefore, in order to reason over such programs, *compositionality* plays a crucial role: the semantics of the whole program must be obtainable as a simple function from the semantics of its individual modules. In this paper we propose a compositional semantics for first-order programs. This semantics is correct with respect to the set of logical consequences of the program. Moreover, – in contrast with other approaches – it is always computable. Furthermore, we show how our results on first-order programs may be applied in a straightforward way to normal logic programs, in which case our semantics might be regarded as a compositional counterpart of Kunen's semantics. Finally we discuss and show how these results have to be modified in order to be applied to normal CLP.

1 Introduction

Modularity in Logic Programming. Modularity is a crucial feature of most modern programming languages. It allows one to construct a program out of a number of separate *modules*, which can be developed, optimized and verified separately. Indeed, the incremental and modular design is by now a well established software-engineering methodology which helps to verify and maintain large applications.

In the logic programming field, modularity has received a considerable attention (see for instance [6]), and has generated two distinct approaches: the first one is inspired by the work of O'Keefe [25] and is based on the consideration that module composition is basically a *metalinguistic* operation, in which the modular construct should be independent from the logic language being used; the second one originated with the work of Miller [23, 24], and is obtained by using a logical system richer than Horn clauses, thus providing a

linguistic approach.

In this paper we follow the first approach. Viewing modularity in terms of *meta-linguistic* operations on programs has several advantages. In fact it leads to the definition of a simple and powerful methodology for structuring programs which does not require to extend the underlying language's syntax. This is essential if we want to compose modules written in different languages. Furthermore, the typical mechanisms of the object-oriented paradigm, such as encapsulation and information hiding can be easily realized within this framework [2].

The Need for a Compositional Semantics. In order to deal with modular programs, it is crucial that the semantics we refer to is *compositional*, i.e. that the semantics of the whole program is a (simple) function of the semantics of its modules. The need for a compositional semantics becomes even more pressing if one wants to build applications in which logic modules are combined with modules that are not logic programs themselves, such as constraint solvers, imperative programs, neural networks, etc. In such a situation, compositionality enables one to reason about the logic module in isolation, while the reference to knowledge provided by other modules is maintained intact.

In logic programming, this need for a compositional semantics has been long recognized. For *definite* (i.e. negation-free) logic programs a few semantics have been proposed. To the best of our knowledge, the first papers to discuss various forms of compositional semantic characterizations of definite logic programs were the ones of Lassez and Maher [18, 20]. Further work has been done by Mancarella and Pedreschi [22] and Brogi et. al. [4]. In [12] Gaifman and Shapiro proposed a compositional semantics, which was further extended in [3] and – for CLP programs – in [11].

Compositionality vs. Non-Monotonicity. However, in the development of semantics for *normal* logic programs, (which employ the negation operator) compositionality has been widely disregarded. Notable exception to this are the papers by Maher [21] and Ferrand and Lallouet [9]. The reason of this disattention is that, because of the presence of the negation-as-failure mechanism, the semantics of normal logic programs is typically non-monotonic. Now, compositionality and non-monotonicity are (almost) irreconcilable aspects. Compositionality implies that the 'old knowledge' is maintained when new knowledge is added. Non-monotonicity is defined as exactly the opposite. Thus, it seems that one can have either compositionality or non-monotonicity, but not both. Still, we need both aspects. On the one hand, the non-monotonicity that arises from the use of negation as failure is something we want in our logic programming language, because it enables us to define relations in a natural and succinct manner. On the other hand, modularity, and therefore compositionality of the declarative semantics, is essential when one wants to use a logic programming language in real life applications.

Contribution of this Paper. In this paper we propose a semantics for

modular logic programs. This semantics is compositional while remaining non-monotonic to a certain extent. In essence, the semantics is compositional and monotonic on the level of composition of modules, while addition of clauses to modules remains a non-monotonic operation.

We carry out our task by first providing a compositional semantics for *first-order* programs, which extends the semantics given by Sato [27] (which in turn can be regarded as an extension to first-order programs of Kunen's [17] semantics). In a second stage we show how this can be naturally used to provide a compositional semantics for normal logic programs and normal CLP. In the end, the semantics we propose can also be regarded as a compositional extension of Kunen's semantics [17].

2 Preliminaries

We assume that the reader is familiar with the basic concepts of logic programming; throughout the paper we use the standard terminology of [1, 19]. Symbols with a $\sim$ on top denote tuples of objects, for instance $\tilde{x}$ denotes a tuple of variables $x_1, \ldots, x_n$, and $\tilde{x} = \tilde{y}$ stands for $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$.

Throughout the paper we will work with three valued logic: the truth values are then **t**, **f** and **u**, which stand, respectively, for *true*, *false* and *undefined*. We adopt the Kleene's truth tables of [16].

Three valued logic allows us to define connectives that do not exist in two valued logic. In particular in the sequel we use the symbol $\Leftrightarrow$ corresponding to the operator of "having the same truth value": $a \Leftrightarrow b$ is **t** if a and b are both **t**, both **f** or both **u**; in any other case $a \Leftrightarrow b$ is **f**. As opposed to it, the usual $\leftrightarrow$ is **u** when one of its arguments is **u**. In most cases we restrict our attention to formulas which we consider "well-behaving" in the three valued semantics. A logic connective $\diamond$ is *allowed* iff the following property holds: when $a \diamond b$ is **t** or **f** then its truth value does not change if the interpretation of one of its argument is changed from **u** to **t** or **f**. A first order formula is *allowed* iff it contains only allowed connectives.

Notice that any formula containing the connective $\Leftrightarrow$ is not allowed, while formulas built with the three-valued counterpart of the "usual" logic connectives are allowed. Allowed formulas can be seen as monotonic functions over the lattice on the set $\{\mathbf{u}, \mathbf{t}, \mathbf{f}\}$ which has **u** as bottom element and **t** and **f** are not comparable. Finally, in what follows we always assume the equality symbol $=$ to be part of the language of the programs and modules we deal with, so – in some cases – in order to avoid confusion we will use $\equiv$ to denote equality at the meta-level.

First-Order Programs and Modules. Let us now recall the definition of a modular logic program. Intuitively, a modular logic program consists of a number of logic modules, each of which consists of a number of predicate definitions. The *definition (of a predicate p)* is a formula of the form

$$p(\tilde{x}) \Leftrightarrow \phi[\tilde{x}]$$

where $\tilde{x}$ is a tuple of distinct variables, and $\phi[\tilde{x}]$ is a first order (allowed) formula whose free variables are exactly the variables of $\tilde{x}$ (the notation $\phi[\tilde{x}]$ is used to emphasize this fact). $p(\tilde{x})$ and $\phi[\tilde{x}]$ are usually referred to as the *head* and the *body* of the definition.

Modules are defined within the context of a fixed *base language* $\mathcal{L}_B$, which contains all the constants and function symbols which may occur in the module itself, and the predicate symbols of those relations which have some predefined meaning. We assume that $\mathcal{L}_B$ always contains the equality symbol and (with a harmless overload of notation), three predicative constants $\mathbf{t}$, $\mathbf{f}$, $\mathbf{u}$, corresponding to the truth values $\mathbf{t}$, $\mathbf{f}$, $\mathbf{u}$. The primitive predicate symbols in $\mathcal{L}_B \setminus \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ are assumed to be defined in a fixed first-order consistent *base theory* Δ. Typical choices for Δ are for example the set of equality axioms together with Clark's equality theory, the domain closure axiom, or axioms defining arithmetic primitives. A relation we will always assume being part of the language is equality ($=$); its meaning may be either the identity over the domain of discourse or – if one prefers – it may be given by a suitable *complete* theory, in which case it is assumed to be incorporated in Δ.

Then, a *module* M on a base language $\mathcal{L}_B$ is a collection of predicate definitions such that each predicate is defined at most once, and none of the predicates in $\mathcal{L}_B$ is defined in M.

We define $Def(M)$ to be the set of predicates that are defined in M, and $Open(M)$ to be the set of predicates which are in neither $Def(M)$ nor $\mathcal{L}_B$ (of course we assume that $Def(M) \cap \mathcal{L}_B = \emptyset$). Predicates in $Open(M)$ are supposed to be *imported*, i.e. defined in some other – maybe unspecified – module M'. Those predicates are also referred to as the are the *open* predicates of M. If $Open(M)$ is empty then the module is said to be *closed*. A closed module corresponds to a classical first-order program. Also, we define $Pred(M)$ as $Def(M) \cup Open(M)$.

Semantics. A three-valued *structure* Σ for the language $\mathcal{L}_B$ is a pair $\langle D, I \rangle$, where D (the *domain*) is a non-empty set, and I is an interpretation over D, which is *two valued* for the predicates in $\mathcal{L}_B \setminus \{\mathbf{u}\}$, and three valued for the other predicate symbols. We also assume $\mathbf{t}$, $\mathbf{f}$ and $\mathbf{u}$ always take the value *true*, *false* and *undefined*. Given a sentence S, we use the notation $Val(S, \Sigma)$ to denote the truth value of S in Σ. Furthermore we say that Σ is a *model* of the set of sentences Γ if for each sentence $S \in \Gamma$ we have that $Val(S, \Sigma) = \mathbf{t}$; consequently, the three-valued logical consequence relation $\models$ is defined as follows: $\Gamma \models F$ iff $Val(F, \Sigma) = \mathbf{t}$ for every model Σ of Γ.

2.1 The unfolding operator

The semantics we are going to give is based on the unfolding operation, therefore we start with recalling its definition.

Definition 2.1 (Unfolding) *Let* $cl : p(\tilde{x}) \Leftrightarrow \phi[\tilde{x}]$ *and* $d : q(\tilde{y}) \Leftrightarrow \psi[\tilde{y}]$ *be two predicate definitions (which we assume to be standardized apart). Let*

$q(\tilde{t})$ be an atomic subformula of $\phi[\tilde{x}]$. Then, by unfolding *$q(\tilde{t})$ in cl (*via *d) we mean substituting $q(\tilde{t})$ with $\psi[\tilde{t}/\tilde{y}]$ in cl. In this case cl is called the* unfolded *definition while d is the* unfolding *one.* □

If M and N are modules, by *unfolding* M with N, $M \circ N$, we naturally mean applying the unfolding operation (in parallel) to all the atoms in the bodies of the definitions of M which are defined in N, using clauses of N as unfolding clauses. As usual, we associate the $\circ$ operator to the left. Thus, $M \circ N \circ O$ should be read as $(M \circ N) \circ O$. Now, for a module M, we adopt the following notation:

$$M^n \equiv \begin{cases} \{p(\tilde{x}) \Leftrightarrow p(\tilde{x}) \mid p \in \mathit{Def}(M)\} & \text{, if } n = 0 \\ M^{n-1} \circ M & \text{, otherwise} \end{cases}$$

So, intuitively, M^n is obtained from M by unfolding n times all its atoms (using the definitions of M as unfolding definitions). Notice that $M \equiv M^1 \equiv M \circ M^0 \equiv M^0 \circ M$.

The unfolding operation, when applied to a closed module is *correct*, in the sense that it maintains the set of (allowed) logical consequences. This is the content of the following Lemma, which is due to Sato [27].

Lemma 2.2 *Let M be a closed module on the base language $\mathcal{L}_B$. Suppose that M' is obtained from M by (repeatedly) applying the unfolding operation, using the definitions of M as unfolding definitions. Then, for any allowed formula ϕ, we have that $M \cup \Delta \models \phi$ iff $M' \cup \Delta \models \phi$.* □

3 A Compositional Semantics

Following the original paper of R. O'Keefe [25], the approach to modular programming we consider here is based on a *meta-linguistic* programs composition mechanism. In this framework, logic programs are seen as elements of an algebra and the composition operation is modeled by an operator on the algebra.

Viewing modularity in terms of *meta-linguistic* operations on programs has several advantages. In fact it leads to the definition of a simple and powerful methodology for structuring programs which does not require to extend the underlying language's syntax. This is not the case if one tries to extend programs by *linguistic* mechanisms, an approach which originated with the work of Miller [23, 24]. Moreover, *meta-linguistic* operations are quite powerful. For instance, the compositional systems of Mancarella and Pedreschi [22], Gaifman and Shapiro [12], Bossi et. al. [2] and Brogi et. al. [4, 5] can be seen as different instances of this idea. Furthermore, the typical mechanisms of the object-oriented paradigm, such as encapsulation and information hiding, as well as more complex form of composition mechanisms – in which we may distinguish between imported, exported, and local (hidden) predicates – can be easily realized within this framework. These mechanisms

are implemented – for instance – in the language Gödel [13], in Quintus Prolog [26] and in SICStus Prolog [7]. For a more detailed analysis we refer to the survey of Bugliesi et. al. [6].

3.1 Module Composition

To compose first-order modules we follow the same approach of [3] and use a simple program union operator.

Definition 3.1 (Module Composition) *Let M_1 and M_2 be modules on the base language $\mathcal{L}_B$. We define*

$$M_1 \oplus M_2 = M_1 \cup M_2$$

provided that $Def(M_1) \cap Def(M_2) = \emptyset$. Otherwise $M_1 \oplus M_2$ is undefined. □

This definition extends in a straightforward way to the case of several modules: $M_1 \oplus \ldots \oplus M_k$ is defined naturally as $(M_1 \oplus \ldots \oplus M_{k-1}) \oplus M_k$. Note that, in the definition we use, we require $Def(M_i) \cap Def(M_j) = \emptyset$, for all distinct i and j. This condition allows us to circumvent a number of unnecessary technicalities, and, in particular, to keep modules composition a *monotonic* operation. At first, the condition seems rather restrictive, in that it prevents one from refining a predicate p in a module M, by composing it with some module M' also containing a definition for p. However, part of the problem can be easily solved by the use of some renaming and an additional 'interface' module. Consider a predicate p, defined in both N_1 and N_2. We rename p to p_1 (resp. p_2) in the head of the definition of p in N_1 (resp. N_2), resulting in a module N_1' (resp. N_2'). Additionally, we define an interface module $I \equiv \{p(\tilde{x}) \Leftrightarrow p_1(\tilde{x}) \vee p_2(\tilde{x})\}$ Now observe that $I \oplus N_1' \oplus N_2'$ is well-defined (provided there are no other name clashes) and behaves exactly the way we would expect $N_1 \oplus N_2$ to. Finally, it is worth noticing that mutual recursion among modules *is allowed.*

3.2 Expressiveness of Modules

Now, we have to give a formal definition to the abstract concept of (semantical) *expressiveness* of modules, for this we have to take into account the fact that modules are meant to be composed together. In the rest of this section, we *always* assume that all the modules are given on the same fixed base language $\mathcal{L}_B$, and that the meaning of the predicates and functions in $\mathcal{L}_B$ is provided by a fixed base theory Δ.

Definition 3.2 *Let M and N be two modules such that $Def(M) = Def(N)$. We say that*

$$M \text{ is (compositionally) more expressive than } N,\ M \succeq N,$$

iff for any other module Q such that $M \oplus Q$ and $N \oplus Q$ are defined, we have that for any allowed formula ϕ, if $N \oplus Q \cup \Delta \models \phi$ then $M \oplus Q \cup \Delta \models \phi$. We also say that

$$M \text{ and } N \text{ are (compositionally) equivalent, } M \sim N$$

iff $M \succeq N \succeq M$. □

In other words, we say that two first-order modules are compositionally equivalent if they have the same set of logical consequences in every possible *context*. Therefore, – according to the notation of [6] – $\sim$ is actually a congruence relation. The following lemma states an obvious yet important property of $\succeq$.

Lemma 3.3 *Let M, N and Q be modules such that $M \oplus Q$ is defined. If $M \succeq N$ then $M \oplus Q \succeq N \oplus Q$.* □

3.3 A Compositional Semantics for First-Order Modules

We start by defining the *skeleton* of a module. For a module M, we denote $Dummy(M) \equiv \{p(\tilde{x}) \Leftrightarrow \mathbf{u} \mid p \in Def(M)\}$. Then, the *skeleton* of M is defined as

$$[M] \equiv M \circ Dummy(M)$$

Using the skeleton and the unfolding operator, we can generate an infinite chain as follows: $[M^0], [M^1], [M^2], \ldots$. Now, it can be proven that, independently from the base theory Δ, for any n, we have that $[M^n] \preceq [M^{n+1}]$. Thus the chain is of increasing expressiveness. Next theorem shows that the semantics of the chain converges to the semantics of the module.

Theorem 3.4 (Main) *Let $M_1, \ldots, M_k$ be first-order logic modules such that $M_1 \oplus \ldots \oplus M_k$ is defined. Then, for any allowed formula ϕ,*

- $M_1 \oplus \ldots \oplus M_k \cup \Delta \models \phi$ *iff* $\exists_n\ [M_1^n] \oplus \ldots \oplus [M_k^n] \cup \Delta \models \phi$ □

This theorem might be regarded as a compositional counterpart of [27, Theorem 3.3] (which, in turn, is the first-order version of [17, Theorem 6.3]). Notice that, if M is a module, then $[M^n]$ is a collection of formulae of the form $p(\tilde{x}) \Leftrightarrow \phi[\tilde{x}]$, where $\phi[\tilde{x}]$ is an allowed formula containing *only* open or base predicates (for instance, in $[M^n]$, recursion is impossible). In a way, we could say that each $[M^n]$ is an *elementary* module; using this notation the above theorem states that the semantics of a module M is given by the $\preceq$-increasing sequence of elementary modules $[M^0], [M^1], [M^2], \ldots$.

Let us now work out a small example. The following program verifies, given a directed graph, whether a certain node is *critical*, i.e. whether by removing that node from the graph, some other nodes in the network become disconnected. We assume that the graph is represented in a module M_g. This module defines only the predicate $arc/2$ in such a way that $arc(x, y)$ is $\mathbf{t}$ in M_g iff there is a (direct) link from x to y in the graph.

Further, we have a module M_p which, referring to $arc/2$ as an open predicate, defines the predicate $path/3$ as follows

$$path(x,z,a) \Leftrightarrow \begin{array}{l} arc(x,z) \vee \\ \exists_y\ arc(x,y) \wedge \neg member(y,a) \wedge path(y,z,[y|a]) \end{array}$$

Thus, $path(x,z,a)$ is true iff there exists an acyclic path from x to y that avoids all the nodes in a. The predicate $member/2$ is assumed to be defined in the usual way in a separate module M_m. Finally, we have a module M_c that defines the predicate $critical/1$; it contains the single definition

$$critical(x) \Leftrightarrow \exists_{y,z}\ x \neq y \wedge x \neq z \wedge path(y,z,[]) \wedge \neg path(y,z,[x])$$

which states that x is critical if we can find a path from some node y to some node z, both different from x, but we cannot find a path from y to z that avoids x. If we want to compute critical nodes of different graphs, we compose this module with different graph modules.

Now, let us see how these modules behave under unfolding. Consider module M_p. The following table shows the body of the definition of $path/3$ in M_p^0, in $M_p^1 (\equiv M_p)$ and in M_p^2.

n	body of $path/3$ in M_p^n
0	$\mathbf{u}$
1	$arc(x,z) \vee$ $\exists_y\ arc(x,y) \wedge \neg member(y,a) \wedge path(y,z,[y\|a])$
2	$arc(x,z) \vee$ $\exists_y (\ arc(x,y) \wedge \neg member(y,a) \wedge$ $(\ arc(y,z) \vee$ $\exists_{y'}\ arc(y,y') \wedge \neg member(y',[y\|a]) \wedge path(y',z,[y'[y\|a]])\ )\ )$

The definition of $path/3$ in $[M_p^0]$, in $[M_p^1]$ and in $[M_p^2]$ can simply be obtained by replacing with the constant $\mathbf{u}$ all the atoms in the above table which have $path$ as predicate symbol ($path$ is the only non-open predicate symbol occurring in M_p).

Finally, it is worth noticing that, since the body of the definition of $critical/1$ does not contain any non-open predicate, we have that, for all n, $M_c \equiv M_c^n \equiv [M_c^n]$.

4 Normal (Constraint) Logic Programs

In this section we show how the results provided in the previous section may be used in a straightforward way in order to provide a compositional semantics to normal logic programs (i. e. logic programs with negation). Normal modules are finite collections of *normal clauses*, $A \leftarrow L_1, \ldots, L_m$. where A is an atom and each L_i is a literal (i.e. an atom or a negated atom). We adopt the usual

logic programming notation that uses "," instead of $\wedge$, hence a conjunction of literals $L_1 \wedge \ldots \wedge L_n$ will be denoted by $L_1, \ldots, L_n$ or by $\tilde{L}$.

Completion for Normal Modules. Since negative information cannot follow from a set of clauses, in order to provide a sound semantics to a normal module we follow [8] and refer to the module's completion. This is a standard approach, and – among the "standard" approaches – it is the only one that allows one to remain within first-order logic. When dealing with three-valued logic the definition of completion is given using the operator $\Leftrightarrow$ instead of $\leftrightarrow$, as follows.

Definition 4.1 *Let M be a normal module and $p(\tilde{t}_1) \leftarrow \tilde{B}_1, \ldots, p(\tilde{t}_r) \leftarrow \tilde{B}_r$ be all the clauses which define the predicate symbol p in M. The* completed definition *of p is*

$$p(\tilde{x}) \Leftrightarrow \bigvee_{i=1}^{r} \exists \tilde{y}_i \ (\tilde{x} = \tilde{t}_i) \wedge \tilde{B}_i.$$

where $\tilde{x}$ are new variables and $\tilde{y}_i$ are the variables in $p(\tilde{t}_i) \leftarrow \tilde{B}_i$.

The completion of *M, $Comp(M)$ consists in the conjunction of the completed definition of all the predicates* defined *in M.* □

It is important to notice that here we depart from [8] in the fact that we *do not close those definitions which are not explicitly given in M.* In a modular context, these predicates need to remain open.

The completed definition of a predicate is a first order formula that contains the equality symbol; hence, in order to interpret "=" correctly, we also need an equality theory.

In particular, we will refer to $\mathrm{CET}_{\mathcal{L}}$, *Clark's Equality Theory for the language* $\mathcal{L}$, which consists of the following axioms:

- $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ for all distinct f and g in $\mathcal{L}$;
- $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \rightarrow (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \quad \forall f \in \mathcal{L}$;
- $x \neq t(x)$ for all terms $t(x)$ distinct from x in which x occurs;

together with the usual *equality axioms*, i. e. *reflexivity, symmetry, transitivity,* and $(\tilde{x} = \tilde{y}) \rightarrow (f(\tilde{x}) = f(\tilde{y}))$ for all functions symbols f in $\mathcal{L}$. Notice that that "=" is always interpreted as two valued. Obviously, $\mathrm{CET}_{\mathcal{L}}$ depends on the underlying language $\mathcal{L}$, which we assume to be fixed and to contain all the functions symbols occurring in all the modules we consider.

A known problem that semantics based on program completion face is that when $\mathcal{L}$ is finite (that is, when it contains only a finite number of functions symbols) $\mathrm{CET}_{\mathcal{L}}$ is not a complete theory. Typically, this problem is solved by adopting one of the following solutions: (a) adding to $\mathrm{CET}_{\mathcal{L}}$ some domain closure axioms which are intended to restrict the interpretation of the quantification to $\mathcal{L}$-terms (as in [28]), or (b) assuming that the language contains always an infinite set of function symbols (as in [17]) or (c) by considering only interpretations and models over a specific fixed domain D (as in [10]). This latter solution requires the adoption of axioms which are usually not first order (unless all the functions symbols are 0-ary, i.e. constants),

and consequently leads to a semantics which is (usually) noncomputable. For these reasons we adopt either solutions (a) or (b). Luckily, these two solutions yield basically the same semantics. For an extended discussion of the subject, we refer to [17, 28].

Let $\mathcal{L}$ be a finite language (i.e. a language with a finite set of predicate symbols). The *Domain Closure Axiom* for the language $\mathcal{L}$, $\mathrm{DCA}_{\mathcal{L}}$, is

$$\exists \tilde{y}_1 \ (x = f_1(\tilde{y}_1)) \vee \ldots \vee \exists \tilde{y}_r \ (x = f_r(\tilde{y}_r))$$

where $f_1, \ldots, f_r$ are all the function symbols in $\mathcal{L}$ and $\tilde{y}_i$ are tuples of variables of the appropriate arity. This axiom is also referred to as the *weak* domain closure axiom[1].

A Compositional Semantics for Normal Programs. It is now easy to see that in this context, the semantics for open normal logic modules finds a natural embedding in the one proposed for first order modules in Section 3 (the underlying language $\mathcal{L}_B$ contains only the equality predicate). Modules composition is defined exactly as for the case of first-order modules: if M_1 and M_2 are normal modules. We define $M_1 \oplus M_2 = M_1 \cup M_2$ provided that $Def(M_1) \cap Def(M_2) = \emptyset$ holds. Otherwise $M_1 \oplus M_2$ is undefined.

Corollary 4.2 *Let $M_1, \ldots, M_k$ be normal modules such that $M_1 \oplus \ldots \oplus M_k$ is defined. Then, for each allowed ϕ there exists an integer n such that the following statements are equivalent:*

1. $\mathrm{Comp}(M_1 \oplus \ldots \oplus M_k) \cup \mathrm{CET}_{\mathcal{L}} \models \phi$
2. $[Comp(M_1)^n] \oplus \ldots \oplus [Comp(M_k)^n] \cup \mathrm{CET}_{\mathcal{L}} \models \phi$

where we assume that, if $\mathcal{L}$ is finite, $\mathrm{CET}_{\mathcal{L}}$ incorporates $DCA_{\mathcal{L}}$. □

As an example, let us consider again the problem of deciding whether a node in a graph is critical. The program given in the previous section can also be written as a modular normal program composed by the modules defining *arc*, *member*, together with the following two modules.

$$\begin{array}{lrcl} N_p \equiv & path(x,z,a) & \leftarrow & arc(x,z). \\ & path(x,z,a) & \leftarrow & arc(x,y), \neg member(y,a), path(y,z,[y|a]). \\ N_c \equiv & critical(x) & \leftarrow & x \neq y, x \neq z, path(y,z,[]), \neg path(y,z,[x]). \end{array}$$

In fact it is easy to check that M_p and M_c coincide with the completion of N_p and N_c.

[1] As opposed to it, the *strong* domain closure axiom for the language $\mathcal{L}$ is $x = t_1 \vee x = t_2 \vee \ldots$ where $t_1, t_2, \ldots$ is the (usually infinite) sequence of all the ground $\mathcal{L}$-terms. This axiom is equivalent to choice (c) above, and determines uniquely the universe of the possible interpretation. Again, if $\mathcal{L}$ contains a non-constant function symbol then the above axiom is not a first order formula, and leads to a noncomputable semantics.

Compositionality vs. Non-Monotonicity. In the introduction, we referred to the fact that - to a certain extent - we manage to combine compositionality and non-monotonicity. More precisely, we have that *within a module* the addition of a clause is a non-monotonic operation, while at the *meta-level* module composition is a monotonic one. To illustrate this point, we give a simple example. Consider two normal modules $M \equiv \{q(a).\}$ and $Q \equiv \{p \leftarrow \neg q(b).\}$, with $\mathcal{L}_B$ consisting of equality and the constants a and b. Now, suppose we want to add to the database the fact that $q(b)$ holds. If we do this by simply adding a clause to M we have a non-monotonic behavior, which implies a defeat of compositionality. For instance we have that

$$\mathrm{Comp}(M) \oplus \mathrm{Comp}(Q) \cup \mathrm{CET}_{\mathcal{L}_B} \models p$$

while

$$\mathrm{Comp}(M \cup \{q(b).\}) \oplus \mathrm{Comp}(Q) \cup \mathrm{CET}_{\mathcal{L}_B} \not\models p$$

Now, it is important to notice that we are not allowed to add the clause $q(b)$ via a module composition operation. In fact $M \oplus \{q(b)\}$ is not defined, as the condition on name clashes is violated.

If we wanted to be able to add the knowledge $q(b)$ via a module composition operation (thus in a compositional way) we would have had to start with a modified version of M, namely with the following:

$$N \equiv \left\{ \begin{array}{l} q(a). \\ q(x) \leftarrow q'(x). \end{array} \right\}$$

Here the predicate q' is an open predicate which can be used to extend our knowledge on q. Now, $N \oplus \{q'(b).\}$ is defined and going from N to $N \oplus \{q'(b).\}$ we have a monotonic behavior. In fact

$$\mathrm{Comp}(N) \oplus \mathrm{Comp}(Q) \cup \mathrm{CET}_{\mathcal{L}_B} \not\models p$$

$$\mathrm{Comp}(N) \oplus \mathrm{Comp}(\{q'(b).\}) \oplus \mathrm{Comp}(Q) \cup \mathrm{CET}_{\mathcal{L}_B} \not\models p$$

Thus, the negation-as-failure mechanism can still be profitably employed in a non-monotonic manner, as long as the negated atom and its descendants in the proof tree are not open. This has to be so: the failure of proving an atom whose proof tree could be augmented by module's composition can not be taken as "sufficient evidence" for assuming *true* the negation of the atom itself (as usually done by negation as failure).

4.1 Normal CLP Modules

For obvious space limitations we only give a brief sketch of how the results of the previous section may be applied to CLP programs.

The *Constraint Logic Programming* paradigm (CLP for short) has been proposed by Jaffar and Lassez [14] in order to integrate a generic computational mechanism based on constraints with the logic programming framework. Such an integration results in a framework which – for programs

without negation – preserves the existence of equivalent operational, model-theoretic and fixpoint semantics. Indeed, as discussed in [21], most of the results which hold for *definite* (i.e. negation-free) constraint logic programs can be lifted to CLP in a quite straightforward way. We refer to the recent survey [15] by Jaffar and Maher for the notation and the necessary background material about CLP. A CLP clause is a formula of the form $A \leftarrow c \wedge L_1 \wedge \ldots \wedge L_k$ where A is an atom, $L_1, \ldots L_k$ are literals and c is a *constraint*, i. e. a first order formula in a specific language $\mathcal{L}_C$. Historically, the semantics of the constraints is determined in either one of the following two ways:

1. by providing a consistent *Theory*, that their interpretation has to satisfy (like Peano's arithmetic); or
2. by giving *structure* Σ over which they have to be interpreted, (for instance, the natural numbers).

It is clear that if we follow the first approach then the results of the previous section can be naturally used to provide a semantics to normal CLP. All we have to do is to incorporate in the base theory Δ the theory that provides a meaning to the constraints and to refer to the modules *completion* (which is defined exactly as in the case of normal logic programs). The rest is straightforward.

Regrettably, the second approach is certainly more popular in the CLP community (even though also the first one is considered standard [15]). The problem with this approach is that the given structure determines uniquely the universe of the models, and this – in presence of negation – leads to a semantics which is again usually noncomputable. As already done in [17, 27], we can avoid this problem by referring to some *elementary extension* of the structure itself. In the rest of this section we briefly sketch how this may be done. First, we have to establish some notation.

Let M be a first-order module on the base language $\mathcal{L}_B$. Let $\Sigma = \langle D, I \rangle$ be a structure *for* $\mathcal{L}_B$ We say that the first-order allowed formula ϕ *follows from* M *under the structure* Σ, we write $M \models_\Sigma \phi$, if $Val(\phi, \Sigma') = \mathbf{t}$ for every model $\Sigma' = \langle D', I' \rangle$ of M for which $D' = D \cup Pred(M)$ and $I'|_D = I$; i.e. if ϕ is true in all the models of M whose universe coincides with D, and whose interpretation of functions and predicates in $\mathcal{L}_B$ coincides with the one given by Σ. Now, let $\Sigma = \langle D, I \rangle$, $\Sigma' = \langle D', I' \rangle$, be two structures for $\mathcal{L}_B$, we say that Σ' is an *elementary extension* of Σ if $D' \supseteq D$ and, for any allowed formula $\phi[x]$ in $\mathcal{L}_B$, we have that $Val(\phi[t], \Sigma) = Val(\phi[t], \Sigma')$, for any $t \in D$. Thus, reasoning over Σ' is basically like reasoning over Σ.

We are now able to state the counterpart of Corollary 4.2.

Corollary 4.3 *Let $C_1, \ldots, C_k$ be CLP modules such that $C_1 \oplus \ldots \oplus C_k$ is defined. If $\mathcal{L}_C$ is the language of the constraints and Σ is a structure for $\mathcal{L}_C$, then there exists an elementary extension Σ' of Σ such that, for each allowed ϕ the following statements are equivalent*

1. $Comp(C_1 \oplus \ldots \oplus C_k) \models_{\Sigma'} \phi$
2. $\exists_n\ [Comp(C_1)^n] \oplus \ldots \oplus [Comp(C_k)^n] \models_{\Sigma'} \phi$ □

The need to refer to an enriched structure Σ' is shown by the following example. Consider the following CLP modules over the language of integer arithmetics

$$N_1 \equiv \{p \leftarrow \neg n(x).\}$$
$$N_2 \equiv \{n(0)., n(x) \leftarrow x = y + 1 \wedge n(y).\}$$

If the interpretation of the constraint is determined by the standard structure **Nat**, with the set $\mathbb{N}$ of natural numbers as universe, then we have that $Comp(N_1) \oplus Comp(N_2) \models_{\mathsf{Nat}} \neg p$, while for no natural n we will have that $[Comp(N_1)^n] \oplus [Comp(N_2)^n] \models_{\mathsf{Nat}} \neg p$. This shows the need of extending the extend the structure **Nat**. Further, in our opinion, p should not be considered false in the semantics of $N_1 \oplus N_2$: firstly because if we take *any* non-trivial extension **Nat'** of **Nat**, $Comp(N_1) \oplus Comp(N_2) \not\models_{\mathsf{Nat}'} \neg p$, so the falsehood of p depends in a way from the limits of the universe of **Nat**, and, secondly, because the falsehood of p is in any case not computable (one would need $\omega + 1$ computation steps in order to calculate it).

5 Conclusions

In this paper we propose a semantics for first order programs which is compositional with respect to the $\oplus$ (module composition) operator. This semantics is built via a first-order unfolding operator and allows to characterize (compositionally) the set of logical consequences of the module in three valued logics. Further, we have shown how our results may be applied to modular normal programs and normal CLP. The semantics we have proposed may be regarded as a compositional counterpart of Kunen's semantics for normal programs [17] and its first-order version due to Sato [27].

Another recent proposal for a compositional semantics for logic programs is the one of G. Ferrand and A. Lallouet [9]. In this paper, Ferrand and Lallouet propose two compositional semantics, one based on Fitting semantics and one based on well-founded semantics. The notion of program unit they use is similar to the notion of (open) module. The differences between their approach and ours stem mostly from the kind of models that are considered. In both Fitting semantics and well-founded semantics for normal logic programs, interpretations are only considered over a fixed universe (typically, the Herbrand universe of the program). As a result, these semantics cannot be axiomatized within first-order logics. Consequently, – and we think this is even more important – these semantics are in general noncomputable (they may require more than ω iterations in order to be built). In contrast, our semantics for modular normal and first-order logic programs is based upon

arbitrary three-valued models and characterized by a countably infinite sequence of approximations, and is thus recursively enumerable.

In [21] Maher presents a transformation system for normal programs with respect to a compositional version of the perfect model semantics, which is defined in the same paper. From the point of view of modularity the main difference between this paper and [21] is that in [21] modules are also required to have a hierarchical calling pattern. For instance mutual recursion among modules is prohibited (this can be seen as a consequence of the fact that the Perfect Model Semantics itself requires the program to be stratified). From the purely semantics point of view the differences between this paper and [21] may be assimilated to the differences between the perfect model semantics and Kunen's semantics (the first is based on two-valued logics, imposes some syntactic restriction on the syntax of modules (stratification, or local stratification), and, in particular, it is usually not computable).

Acknowledgments. This research was supported in part by the Italian "Comitato naz.le Scienze e Tecnologia dell'Informazione" under the project "Programmazione Logica".

References

[1] K.R. Apt. Logic programming. In L. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 494–574. Elsevier Science Publishers B.V., 1990.

[2] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential logic programming. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[4] A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1):1–21, 1992.

[5] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition Operators for Logic Theories. In J. W. Lloyd, editor, *Proc. Symposium on Computational Logic*, pages 117–134. Springer-Verlag, Basic Research Series, 1990.

[6] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19-20:443–502, 1994.

[7] M. Carlsson. *SICStus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[8] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[9] Gérard Ferrand and Arnaud Lallouet. A compositional proof method of partial correctness for normal logic programs. In John Lloyd, editor, *Proceedings of the International Logic Programming Symposium*, pages 209–223, 1995.

[10] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2:295–312, 1985.

[11] M. Gabbrielli, G.M. Dore, and G. Levi. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

[12] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.

[13] P. M. Hill and J. W. Lloyd. *The Gödel programming language.* The MIT Press, 1994.

[14] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[15] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[16] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, 1995.

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

[18] J.-L. Lassez and M. J. Maher. Closures and Fairness in the Semantics of Programming Logic. *Theoretical Computer Science*, 29:167–184, 1984.

[19] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer-Verlag, 1987. Second, extended edition.

[20] M. J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[21] M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, March 1993.

[22] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1006–1023. MIT Press, 1988.

[23] D. Miller. A Theory of Modules for Logic Programming. In *Proceedings IEEE Symposium on Logic Programming*, pages 106–114, 1986.

[24] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

[25] R. A. O'Keefe. Towards an Algebra for Constructing Logic Programs. In *Proc. IEEE Symp. on Logic Programming*, pages 152–160, 1985.

[26] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[27] T. Sato. Equivalence-preserving first-order unfold/fold transformation system. *Theoretical Computer Science*, 105(1):57–84, 1992.

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