MODULARITY IN LOGIC PROGRAMMING

MICHELE BUGLIESI, EVELINA LAMMA AND PAOLA MELLO

▷ The research on modular logic programming has evolved along two different directions during the past decade. Various papers have focused primarily on the problems of programming-in-the-large. They have proposed module systems equipped with compositional operators for building programs as combinations of separate and independent components. Other proposals have instead concentrated on the problem of programming-in-the-small in an attempt to enrich logic programming with abstraction and scoping mechanisms available in other programming paradigms.

The issues that arise in the two approaches are substantially different. The compositional operators of the former allow one to structure programs without any need to extend the theory of Horn clauses. The scoping and abstraction mechanisms of the latter are modeled in terms of the logical connectives of extended logic languages.

In this paper we provide a uniform reconstruction of the above approaches and we show, wherever this is possible, how the object-level logical connectives of the latter can be mapped onto the compositional operators of the former. ◁

1. INTRODUCTION

The interest in modular logic programming has motivated a considerable research effort over the past decade and it has been the subject of an active and still open

Address correspondence to Michele Bugliesi, Dipartimento di Matematica Pura ed Applicata, Università di Padova, Via Belzoni 7, 35131 Padova, Italy, e-mail: michele@goedel.unipd.it

Evelina Lamma, Paola Mello, Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy, e-mail: {evelina,paola}@deis33.cineca.it

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

debate. The need for a modular extension to logic programming has been always widely agreed upon. It was in fact acknowledged that relations provide a too fine-grained unit of abstraction for the design of large programs, and that having flat composition of clauses as the only mechanism at disposal leaves the programmer with rather poor tools for structuring programs.

However, as to the question of what modular extension should be adopted, there seems to be as yet no final answer. There are at least two measures for evaluating the adequacy of any such proposal. In fact, if from a programming language point of view, any extension can be justified to the extent that it implements useful programming features, it is nevertheless reasonable – from a *logic* programming point of view – to evaluate it also in terms of the logic (if there is any) it encompasses.

A further and important issue that should be addressed in the design of a modular language is related to the ability of the underlying abstraction mechanisms to provide an effective support for both the programming disciplines which are sometimes qualified as *programming-in-the-large* and *programming-in-the-small*.

The design of a principled modular extension of logic programming should therefore address both these issues and satisfy several properties [75]. A modular language should allow rich forms of abstraction, parametrization and information hiding; it should ease the development and maintenance of large programs as well as provide adequate support for re-usability and separate and efficient compilation; it should finally encompass a non-trivial notion of program equivalence to make it possible to justify the replacement of equivalent components. At the same time, we should also expect that these features do not undermine the declarativity of logic programming as it stands, and therefore that the logical foundations on which the extension relies be as firm and well-established as those of the underlying language.

The interest in the aforementioned two dimensions of programming inspired the two orthogonal lines of research the study of modularity has evolved along, over the past ten years.

Various proposals have focused primarily on the issue of programming-in-the-large. This research was inspired by the work of O'Keefe in [86]. His idea was to give a formal account of one of the fundamental principles of the software engineering view of programming: programs should be developed incrementally by defining several units and their interfaces and then by composing those units. This led him to propose an approach to modularity based on the notion of program composition. He formalized this idea by interpreting logic programs as elements of an algebra and by modeling their composition in terms of the operators of the algebra. The distinguishing property of this approach is that the modular extension of logic programming takes place without any need to extend the language of Horn clauses. In fact, module-composition is inherently a *meta-linguistic* mechanism: modules are viewed as sets of Horn clauses and their algebraic composition is modeled in terms of various operations on the component clauses: union, deletion, closure and combination of the above. The compositional frameworks of Mancarella and Pedreschi [61], Gaifman and Shapiro [39], Bossi *et al.* [8] and of Brogi *et al.* [15, 12] can in fact be seen as different formulations of this idea.

Information hiding and encapsulation can also be accounted for in this framework quite elegantly. Algebraic program composition can be made more selective so as to distinguish, within a module, predicates to be imported from other modules and/or predicates to be exported to other modules. This idea has been exploited by Gaifman and Shapiro in [39] and by Brogi *et al.* in [16] for defining a variety of powerful composition mechanisms. Similar mechanisms have also been adopted by prototypical implementations such as the Gödel system described in [46] and, coupled with additional cross-referencing facilities between modules, by some of the existing commercial Prolog systems such as Quintus [87] and SICStus [90].

An alternative approach to developing a principled modular (logic) language arose in the attempt to instrument logic programming with *linguistic* abstraction mechanisms richer than those offered by Horn clauses. The idea was to provide a richer support for programming-in-the-small and then to tailor those mechanisms to attack the problems of programming-in-the-large.

This approach originated with the work of Miller in [71], and was inspired by the observation that logical systems richer than Horn clauses could be employed to provide a natural support for modular programming. His idea was to model the operators for building and composing modules directly in terms of the logical connectives of a language defined as an extension of Horn Clause Logic. This led him to propose in [71] a modular language based on the use of implication goals in the body of clauses. A language with the same structural properties was then proposed by Giordano *et al.* in [42] and [43] to model visibility rules more refined than those effecting the language of [71]. On similar grounds, a *non-monotonic* interpretation of implication led Monteiro and Porto to introduce the *context extension* operator of [78] as the foundation for Contextual Logic Programming. Similarly, messages were proposed [62, 13, 19] as a way to achieve a logical reinterpretation for some of the distinguishing features of the Object-Oriented programming paradigm. Later extensions to the framework of [71] led Miller and his colleagues [70, 85] and, independently, Shapiro and Moscowitz [82], to study other (higher-order) logical frameworks where different notions of scope over clauses and program constants could be modeled.

Outline. In this paper we survey the existing literature on this area. Both the aforesaid lines of research will be taken into account. One of the points of the survey is actually to address, whenever possible, the connections between the two approaches and to point them out.

The first part of the survey (section 2) is dedicated to the study of the algebraic approach in the different formulations it has been proposed. The second part is instead dedicated to the study of the different modular languages defined as linguistic extensions of Horn clauses. Section 3 studies the operational characterization of these languages whereas section 4 explores their logical foundations. Section 5 is devoted to describing their implementation. The final section contains some concluding remarks.

Throughout the paper we will concentrate only on definite programs, i.e. on definite *Horn* programs in section 2, and on programs without negation for the modular languages disussed in the remaining sections. Under this assumption, we will be able to draw a picture where most of the theoretical work which has been done in the area can be discussed in a uniform manner. Also, this choice is faithful to the current status of the research in the field which, with few notable exceptions that will be pointed out in the paper, has concentrated primarily on the case of definite programs.

2. MODULAR PROGRAMMING AS ALGEBRAIC PROGRAM COMPOSITION

We start off our analysis with the study of the algebraic approach to modularity introduced by O'Keefe in [86]. As already anticipated, the fundamental idea behind the work of O'Keefe is that a logic program should always be understood as part of a system of programs. Having taken this view, he argues that new programs can be designed by combining the components of that system and possibly by defining new ones. He formalizes this idea in terms of an algebraic approach where a logic program is viewed as an element of an algebra and the operators for composing programs as operators over that algebra.

Viewing modularity as algebraic program composition offers several advantages. Firstly, program composition is a powerful tool for structuring programs without any need to extend the theory of Horn clauses. Secondly, it supports naturally the re-use of the same program within different composite programs and, when accompanied by an adequate notion of program-equivalence, the replacement of equivalent components. It is also highly flexible, for new composition mechanisms can be accounted for by simply introducing a corresponding operator in the algebra or by combining the existing ones. Finally, when coupled with mechanisms for specifying the interfaces between components (the import/export lists mentioned in the introduction are an example of these interfaces) it also allows one to model powerful forms of encapsulation and information hiding.

In the next subsection we will introduce three operators, called *union*, *overriding-union* and *closure*. Following the guidelines of [14], we will show that these operators suffice to model a rich set of mechanisms for program composition. We will then discuss the notions of program equivalence that arise for the different operators and finally describe the extension of this framework with import/export facilities.

Other operators, such as *intersection* [61] and *deletion* [15], which have been considered in the literature will not be taken into account in this survey since less relevant to the study of modular programming.

2.1. The Algebra of Programs and its Operators

The Algebra. The language L for the programs of the algebra is defined by fixing ahead a signature Σ of function and constant symbols (constants are viewed as usual as nullary functions) and a set Π of predicate symbols. We will henceforth call Σ-term any term built over Σ, and Σ-formula any formula built over Σ and Π.

All the programs in the algebra have the same Herbrand Base built over the signature of L. We denote with B the Herbrand Base and with $P(B)$ the power-set of B. Programs in the algebra are ordinary logic programs.

Following the style of [71], we will use $\top$ to denote the distinguished formula *true* and three metalinguistic variables – A, D and G – to stand respectively for atomic formulas, definite clauses and goals. Using these conventions the structure of Horn clauses can be expressed in terms of the following syntax:

$$\begin{array}{rcl} G & ::= & \top \mid A \mid G \wedge G \mid \exists x G \\ D & ::= & A \mid D \wedge D \mid \forall x D \mid G \supset A \end{array}$$

For reasons of notational convenience, we will often adopt the conventional Prolog syntax for clauses and write $A \leftarrow G_1, \ldots, G_n$ as an alternative notation for $(G_1 \wedge$

$\cdots \wedge G_n) \supset A$. In doing so, we will also assume that $A \leftarrow G_1, \ldots, G_n$ is in normal form, i.e. that all the variables occurring in it are universally quantified.

According to the above definition, a logic program P can be viewed as a conjunctive D-formula or equivalently as a set of D-formulas.

Operators. Program composition is modeled in terms of the operators in the algebra of programs. As already mentioned, we will consider three algebraic operators: *union* ($\cup$), *closure* (*) and *overriding-union* ($\lhd$). Their composition will be denoted with the extension formulas defined by the following productions:

$$E ::= P \mid E \cup E \mid E^* \mid E \lhd E$$

Here P is the name of a logic program, and we abuse the notation writing P to denote a program as well as its name in the algebra. To account for a formal semantics for the composition operators as well as for the programs in the algebra, we will take the immediate consequence operator as the denotation of a program.

Given a program P, the denotation $[\![P]\!]$ of P is defined as $[\![P]\!] = T_P$. T_P is the standard [91] continuous operator over the lattice $(P(B), \subseteq)$.

$$T_P(I) \quad = \quad \{A \mid A \leftarrow A_1, A_2 \ldots, A_N \in [P]_\Sigma \text{ such that } \{A_1, A_2, \ldots A_N\} \subseteq I\}$$

The application of T_P corresponds to a one-step deduction, using P, of ground atoms from ground atoms. The notation $[P]_\Sigma$ is used here to stand for the set of D-formulas obtained from P by closure under conjunction and instantiation. Formally, $[P]_\Sigma$ is the smallest set satisfying the following conditions:

$$\begin{array}{rcl} P \subseteq [P]_\Sigma & & \\ D_1 \wedge D_2 \in [P]_\Sigma & \Rightarrow & D_1 \in [P]_\Sigma \ \wedge \ D_2 \in [P]_\Sigma \\ \forall x D \in [P]_\Sigma & \Rightarrow & D[x/t] \in [P]_\Sigma \text{ for all the } \Sigma\text{-terms } t \end{array}$$

In the following we will also make use of the following definitions. Given any function $f : P(B) \mapsto P(B)$, the *ordinal powers* of f are defined as follows:

$$\begin{array}{rcl} f \uparrow 0 & = & \emptyset \\ f \uparrow \alpha & = & f(f \uparrow \alpha - 1) \\ f \uparrow \alpha & = & \cup_{\beta \leq \alpha} f \uparrow \beta \ \text{ if } \alpha \text{ is a limit ordinal} \end{array}$$

For any f as above, we also define the iteration operator $^\omega$ on f as $f^\omega(X) = \cup_{i=0}^{\infty} f^i(X)$ for any $X \subseteq B$. Note that T_P^ω and $T_P \uparrow \omega$ denote different objects: respectively, a function over $P(B)$, and an element of $P(B)$ – the least fixed point of T_P. It is also easy to see that $T_P \uparrow \omega = T_P^\omega(\emptyset)$.

The choice of T_P as the denotation of a program was discussed by Mancarella and Pedreschi in [61]. As in that case, it will allow us to have an elegant *homomorphic* semantics for our algebra. In fact, we will be able to show that the equality $[\![P \circ Q]\!] = [\![P]\!]\, \sigma(\circ)\, [\![D]\!]$ holds for a suitable choice of a homomorphism σ which maps the operator $\circ$ over programs onto the corresponding operator $\sigma(\circ)$ over the programs' denotation.

It may be argued that the choice $[\![P]\!] = T_P$ does not provide a very useful semantics for reasoning about programs since the notion of program-equivalence it induces has a too strong operational flavour. The objection is admittedly reasonable and we will discuss the choice of other and more abstract semantics in section 2.2. We now turn, instead, to the study of the different operators.

2.1.1. Union. The union of programs is the simplest operator in the algebra. Taking the union of two programs amounts simply to taking the set-theoretic union of their clauses (actually any program can be viewed as the union-composition of all its clauses). The denotation of the union $P \cup Q$ of two programs P and Q is immediately obtained by setting $[\![P \cup Q]\!] = T_{P \cup Q}$ where:

$$T_{P \cup Q}(I) = \; T_P(I) \cup T_Q(I)$$

Using this definition, Mancarella and Pedreschi [61] proved the identity $[\![P \cup Q]\!] = [\![P]\!] \cup [\![Q]\!]$ thus showing that the invariant $[\![\cdot]\!]$ is homomorphic with respect to $\cup$ (notice that here "$\cup$" stands for two different operators defined over programs, on the left of the previous equation, and over their denotation, on the right).

From a programming language perspective, this type of composition implements a form of dynamic scope, since each reference to a predicate in a program refers to a different definition depending on the composition that program is part of. The composition by union has also been shown to be well-suited for implementing forms of knowledge assimilation, where knowledge is dynamically updated as new information becomes available. Each program can in fact be naturally interpreted as *open* with respect to (compositions with) other programs. This corresponds to view an open program as an incomplete description of some knowledge domain. The composition of open programs may increase the degree of completeness of the description, as something which does not hold in one program can hold in another one, and the former can exploit the latter (and vice-versa) to derive new knowledge.

2.1.2. Closure. There are situations in which it might be useful to view a program as *closed* (as opposed to *open*) with respect to possible compositions with other programs. This situation is discussed by Brogi *et al.* in [12], where the authors introduce a *closure* operator which enforces a form closed world assumption on the programs of their algebra. Roughly, the application of the *closure* operator to a program makes that program visible to other programs only in terms of its logical consequences (its extensional knowledge) whereas it encapsulates the program's intensional knowledge (its clauses). In our algebra, we will denote the closure of a program P with P^*. P^* corresponds to the program consisting of the atomic consequences of P; hence, its denotation will be simply given by the constant transformation which returns the least Herbrand model of P for any possible interpretation I. This yields the following homomorphic definition for the closure operator:

$$[\![P^*]\!] = [\![P]\!] \uparrow \omega$$

From a programming language point of view, the result is a mechanism for defining modules with an associated *local* and closed policy of scope. Consider a composite program obtained taking the union of programs which have been encapsulated by means of the * operator. The formulas which can be proved in such compositions are all the formulas which can be proved separately in the component programs.

Example 2.1. Let P and Q be the two programs:

$$P \;=\; \{p(X) \leftarrow q(X)\} \qquad\qquad Q \;=\; \left\{ \begin{array}{l} p(1) \\ q(2) \\ q(X) \leftarrow p(X) \end{array} \right\}$$

Here $P^* = \emptyset$ (the empty set of clauses) and the composition $P^* \cup Q^*$ is the program consisting of the three atomic clauses $\{p(1),\ q(2),\ q(1)\}$.

2.1.3. Overriding-union. The third operator of the algebra allows us to model a different and, in a way, hierarchical type of program composition. Indeed, the union $P \cup Q$ of two programs can be itself thought of as a form of hierarchical composition where P is composed with Q to *extend* the set of definitions contained in Q. Needless to say, as long as we take the union of two programs the order of the components doesn't matter. There are situations, however, in which we may wish to model a mechanism of specialization where the order *does* matter: a typical case is given by inheritance-based systems, where the definitions (methods) in a class override the corresponding definitions provided by the super-class(es). The operator of overriding-union captures precisely this type of behaviour.

Given two logic programs P and Q, $P \lhd Q$ stands for the composition of P and Q into a hierarchy where Q is P's immediate ancestor. Since an overriding semantics is assumed, if both P and Q contain a definition for same predicate, then the definition in P overrides the one found in Q.

The overriding-union of two programs can be defined in terms of union and restriction over the clauses of the component programs. Say that a predicate p is *defined* by a program P, if P contains a clause whose head's predicate symbol is p. Let $\delta(P)$ be the set of predicate symbols defined by P and let $Pred(A)$ denote the predicate symbol of any given atom A. Then, $P \lhd Q$ denotes the program obtained as the union of the clauses of P with the clauses of Q which do not define any of the predicates in $\delta(P)$. Formally:

$$P \lhd Q \quad = \quad P \cup \{A \leftarrow G \in Q : Pred(A) \notin \delta(P)\}$$

Example 2.2. Consider the two programs

$$P \;=\; \{p(2)\} \qquad\qquad Q \;=\; \left\{ \begin{array}{l} p(1) \\ q(X) \leftarrow p(X) \end{array} \right\}$$

Their composition $P \lhd Q$ is a new program which contains definitions for both $p/1$ and $q/1$, and where Q's definition of $p/1$ is overridden by the corresponding definition of P.

$$\left\{ \begin{array}{l} p(2) \\ q(X) \leftarrow p(X) \end{array} \right\}$$

Note that the operator of overriding-union is inherently *non-monotonic*: formulas derivable in Q may be no longer derivable in the composition of $P \lhd Q$. For the two programs above, $p(1)$ is derivable from Q whereas it is not so from $P \lhd Q$.

The $\lhd$-composition of two programs can be expressed in terms of the programs' denotation using the following operator introduced in [19].

Definition 2.1. Let π be an arbitrary set of predicate symbols and let S_1 and S_2 be two elements of $P(B)$. Then the function $\diamond_\pi : P(B) \times P(B) \mapsto P(B)$ is defined as follows:

$$S_1 \diamond_\pi S_2 = S_1 \cup \{t \in S_2 \mid Pred(t) \notin \pi\}$$

This definition is used in [19] to show that $\diamond_\pi$ is continuous over $(P(B), \subseteq)$ for any given π, and that, for any $I \subseteq B$:

$$T_{P \triangleleft Q}(I) = T_P(I) \diamond_{\delta(P)} T_Q(I)$$

Hence, a homomorphic definition for $[\![P \triangleleft Q]\!]$ can be obtained by lifting the definition of $\diamond$ at the function level and setting $[\![P \triangleleft Q]\!] = [\![P]\!] \diamond_{\delta(P)} [\![Q]\!]$

2.1.4. Operator Composition. Various proposals for modular logic programming adopt scope policies which are more complex than those we have discussed so far: typical examples are the operators for nested composition of [5, 78, 43] and the different forms of inheritance-based compositions discussed in [11] and [19, 79].

Programs can be composed in a nested fashion by accommodating them in a stack. Let $S = [P_n, ..., P_i, ..., P_1]$ be a stack where $P_1, \ldots, P_n$ are the component programs and P_n is the top of S. The behaviour of a nested composition is defined by specifying how the evaluation of a goal should be carried out in the stack. The idea is that the clauses used for reducing the goal in S are selected searching the components of S from top to bottom. If one such clause is selected from P_i, then the body of that clause in evaluated in the sub-stack $[P_i, ..., P_1]$. In other words, the definitions contained in P_i are visible only to the programs which have been added to the stack after P_i. This type of composition can be formalized in our setting by means of a composition where $\cup$ and * are suitably alternated. A stack $[P_n, ..., P_i, ..., P_1]$ can be expressed as the composition $(P_n \cup (\ldots \cup (P_i \cup (\ldots \cup P_1^*) \ldots)^* \ldots)^*)^*$: definitions in P_i are accessible to P_{i+1} but not the other way around.

Example 2.3. Consider again programs P and Q of example 2.1. The program we obtain by taking the hierarchical composition $(P \cup Q^*)^*$ has one additional unit clause. Namely:

$$(P \cup Q^*)^* \;=\; \{p(2), p(1), q(2), q(1)\}$$

In [86], O'Keefe introduces a similar scope rule in terms of an operator named *composition* and denoted by $\circ$. Given M_1 and M_2, two programs in the algebra, their composition $M_1 \circ M_2$ corresponds to a program where the definitions of M_2 are accessible to M_1, but not the other way around. O'Keefe uses the (more abstract) denotation $[\![P]\!]_{O'Keefe} = (T_P + Id)^\omega$ for his programs, where "+" denotes the addition of two functions $((f + g)(X) = f(X) \cup g(X))$ and Id is the identity function. Under this definition[1], he is able to model the meaning of $M_1 \circ M_2$ homomorphically as:

$$[\![M_1 \circ M_2]\!]_{O'Keefe} = [\![M_1]\!]_{O'Keefe} \circ [\![M_2]\!]_{O'Keefe}$$

The notable difference with respect to our framework is that $M_1 \circ M_2$ denotes a function over $P(B)$ whereas the denotation of $(M_1 \cup M_2^*)^*$ is a subset of B. In fact, it's immediate that $[\![(M_1 \cup M_2^*)^*]\!] = [\![M_1 \circ M_2]\!]_{O'Keefe}(\emptyset)$.

[1] Actually O'Keefe uses for his programs the different denotation $[\![P]\!]_{O'Keefe} = T_P^\omega$, but all the properties he attributes to this invariant imply that the definition he is really employing is the one we have introduced here.

Inheritance-based Compositions. The $\lhd$-composition of programs can be used to model both the forms of overriding inheritance which are qualified by Reddy in [88] as *static* inheritance, a lá SIMULA67, and *dynamic* inheritance, a lá SMALLTALK.

The difference between the two mechanisms can be explained as follows. Let *HP* be the hierarchy $P_n \ isa \ \cdots \ isa \ P_i \ isa \ \cdots \ isa \ P_1$, where P_i is P_{i+1}'s immediate ancestor, and let G be a goal to be evaluated in *HP*. Assume that the evaluation of G selects a clause in P_i: now, if *isa* is interpreted as static inheritance, then the body B of that clause will be evaluated in the sub-hierarchy $P_i \ isa \ \cdots \ isa \ P_1$. If, on the contrary, *isa* is interpreted as dynamic inheritance, B will be evaluated in the complete hierarchy *HP*. In both cases the overriding semantics of *isa* is captured by the fact that the clauses used for evaluating each (sub) goal are selected only from the top-most component, in the current hierarchy, which contains a definition for that goal.

Dynamic inheritance (with overriding) can be modeled in our framework by connecting the components of the *isa* hierarchy directly through the $\lhd$ operator. More specifically, the hierarchy $P_n \ isa \ \cdots \ P_i \ isa \ \cdots \ P_1$, can be expressed in terms of the extension formula $P_n \lhd \ldots\ldots \lhd P_i \lhd \ldots \lhd P_1$. Example 2.2 illustrates this point: in the composition $P \lhd Q$ the call to $p/2$ in the definition of $q/1$ refers to the definition contained in P.

Static inheritance, instead, can be modeled in terms of compositions where the application of $\lhd$ and * is alternated: $(P_n \lhd (\ldots \lhd (P_i \lhd (\ldots \lhd P_1^*)\ldots)^* \ldots)^*)^*$. For the two programs of example 2.2, the composition $(P \lhd Q^*)^*$ would now consist of the two unit clauses $\{p(2), q(1)\}$.

Obviously, the different forms of inheritance with extension or overriding mode (as well as the other composition mechanisms) are allowed to co-exist within the same inheritance tree as illustrated by the following example.

Example 2.4. The following inheritance tree [14] represents the knowledge that tigers and elephants are animals, that Kahan is a tiger and that Dumbo is an elephant.

$$\begin{array}{cccc} & \swarrow \ \mathit{Tiger} & \longleftarrow & \mathit{Kahan} \\ \mathit{Animal} & & & \\ & \nwarrow \ \mathit{Elephant} & \not\longleftarrow & \mathit{Dumbo} \end{array}$$

Each node contains a set of clauses stating the properties that hold at that node. We assume that *Dumbo* overrides the corresponding properties of a generic elephant. An inheritance tree like this is expressed in terms of the operators of our algebra by the the extension formula:

$$(\mathit{Animal} \cup \mathit{Tiger} \cup \mathit{Kahan})^* \cup (\mathit{Dumbo} \lhd (\mathit{Elephant} \cup \mathit{Animal})^*)$$

Notice the use of $\lhd$ to obtain dynamic inheritance with overriding and the use of * to avoid exchanging of knowledge between the two inheritance chains.

2.2. Compositionality and Full-Abstraction

We mentioned earlier in this section that the choice of T_P as the denotation of a program may not yield a satisfactory semantics for the algebra. Indeed, the property of being homomorphic is very basic as it simply gives us the ability to define the meaning of a composite program in terms of the meaning of its components.

As noted by Maher in [77], this represents probably the very least we should expect from a semantic characterization of a modular language. In fact, what is also reasonable to expect is that the notion of equivalence induced by the choice of the denotation be "useful" for reasoning about the computational behaviour of programs. From a practical point of view, this concept of equivalence is crucial for the development and the maintenance of large programs since it allows one to identify when two modules can be safely substituted with one another without affecting the global behaviour. It should be noted, in this regard, that the (potential) interactions between different components yields a fairly rich notion of equivalence. In fact, two components which are computationally equivalent if considered as stand-alone programs are likely to exhibit a completely different behaviour when viewed as part of a context. Consider for instance the two (equivalent) modules $P = \{\}$ and $Q = \{p \leftarrow q\}$ and contrast them with the (non-equivalent) programs obtained by taking the union of P and Q with the program $R = \{q\}$. Thus, in a modular language, two modules are to be considered equivalent if they can be interchanged in any context without affecting the visible results of the computation. More precisely:

> "two modules P and Q are observationally congruent ($P \equiv_{obs} Q$) iff for every context $C[\cdot]$, $C[P]$ and $C[Q]$ exhibit the same observational behaviour [68]."

Needless to say, reasoning about programs (and their computational equivalence) in terms of their denotation will be worthwhile only when the notion of semantic equivalence is a "good approximation" of the relation of computational equivalence.

These intuitive arguments can be formalized as suggested in [39] and [68], in terms of the two notions of *compositionality* and *full abstraction*. Compositionality ensures that two semantically equivalent programs are also observationally equivalent; full abstraction guarantees that any distinction made at the semantic level has also an observational counterpart. Formally:

> "a semantics is compositional if semantic equality implies observational congruence. It is fully abstract if semantic equality coincides with $\equiv_{obs}$ [68]."

We make these ideas precise following the style and terminology of [39]. For any denotation (or invariant) $[\![\cdot]\!]$, the definition of semantic equivalence $\equiv_{[\![\cdot]\!]}$ is standard: it states that two programs are equivalent if and only if their denotations coincide. Let now P denote a class of programs and Com a class of composition operators. We say that:

An equivalence relation $\equiv$ over P *preserves* Ob if and only if

$$P \equiv Q \;\Rightarrow\; Ob(P) = Ob(Q)$$

An equivalence $\equiv$ is a *congruence* for Com if for every $f \in Com$

$$P_i \equiv Q_i \;\; i = 1, \ldots, n \;\;\Rightarrow\;\; f(P_1, \ldots, P_n) \equiv f(Q_1, \ldots, Q_n)$$

For any notion of observable Ob, and any $op \in Com$, we can define the *equivalence induced by* (Ob, op) as follows:

$$P \equiv_{Ob,op} Q \quad\iff\quad \text{for all } R \left\{ \begin{array}{l} Ob(R\ op\ P) = Ob(R\ op\ Q) \\ Ob(P\ op\ R) = Ob(Q\ op\ R) \end{array} \right.$$

We say that the invariant $[\![\cdot]\!]$ is (Ob, op)-compositional if $\equiv_{[\![\cdot]\!]}$ preserves Ob and is a congruence for *op*. $[\![\cdot]\!]$ is fully abstract if $\equiv_{[\![\cdot]\!]}$ coincides with $\equiv_{Ob,op}$.

Given two equivalences $\equiv_i$ and $\equiv_j$, we will henceforth say that $\equiv_i$ is *finer* or *stronger* than $\equiv_j$ (dually $\equiv_j$ *weaker*, or *coarser*, than $\equiv_i$) whenever $\equiv_i$ implies $\equiv_j$.

As for the notion of observables, there are of course several possible choices. Here we will rely on the following definition:

$$Ob(P) \;=\; \{A \mid P \models A \text{ and } A \in B\}$$

This definition is essentially equivalent to the standard definition of observables based on the notion of success set. As noted by Maher in [77], if we assume that the domain of B contains infinitely many constant symbols, then the equivalence associated with Ob coincides with the equivalence that results by taking as observables the set of ground and *non-ground* atomic consequences. However, if we are to consider only the composition of programs belonging to the same algebra, we don't need this generality and the above definition suffices (see [77] for a fuller discussion about this point).

In the rest of this section, we will consider the different operators of our algebra and study the properties of compositionality and full abstraction for the set of invariants listed below (in increasing order of abstraction):

$[\![P]\!]_1 = T_P;$

$[\![P]\!]_2 = T_P + Id;$

$[\![P]\!]_3 = (T_P + Id)^\omega$

$[\![P]\!]_4 = T_P \uparrow \omega.$

These invariants, and the corresponding equivalences, have all been studied by Maher in [77]. The equivalence induced by $[\![\cdot]\!]_1$ (denoted by $\equiv_1$) coincides with *subsumption* equivalence. $[\![\cdot]\!]_2$ induces a form of *weak* subsumption equivalence ($\equiv_2$) which abstracts upon tautological clauses. The third invariant computes the set of logical consequences of a program (obtained in any number of steps), and induces a notion of equivalence ($\equiv_3$) which coincides with *logical equivalence*. Finally, $[\![\cdot]\!]_4$ is the standard semantics of logic programming which identifies two programs as equivalent ($\equiv_4$) if they have the same least Herbrand model.

Note that in the previous section we have already (implicitly) shown that $[\![\cdot]\!]_1$ is compositional for all the operators in the algebra. In fact, we have shown that $[\![\cdot]\!]_1$ has a homomorphic definition for all such operators. Then, for any composite program $C[\cdot]$, it follows that $[\![C[P]]\!]_1 = [\![C[Q]]\!]_1$ whenever the components P and Q have the same denotation. Since $[\![\cdot]\!]_1$ preserves the observable Ob, it follows that $[\![\cdot]\!]_1$ is compositional for all the operators. On the other hand, it is easy to verify that it is not fully abstract for any of them.

We next turn to the analysis of the remaining invariants. In [10] Brogi develops a similar analysis for a different set of operators, and shows that the sequence of invariants listed above coincides with the sequence of compositional and fully abstract equivalences for the subsets obtained dropping from the complete set of operators one operator at the time. As we show next, some – but not all – of these results carry over to our framework.

We start with $\equiv_4$ and proceed in decreasing order of abstraction.

2.2.1. Minimal-model Semantics: $\equiv_4$. The minimal model semantics is obviously compositional and fully abstract with respect to the *closure* operator. In fact, in this case, the notion of observable coincides with the chosen denotation and thus compositionality and full abstraction are an immediate consequence of the fact the closure of a program is itself is defined in terms of the least Herbrand model of that program. It is also immediate to verify that $\equiv_4$ is not compositional with respect to either $\cup$ or $\lhd$.

2.2.2. Logical Equivalence: $\equiv_3$. As it turns out, the invariant $[\![\cdot]\!]_3$ is $(Ob, \cup)$-compositional and fully abstract. Compositionality was first shown by Maher in [77] using the identity $[\![P \cup Q]\!]_3 = ([\![P]\!]_3 + [\![Q]\!]_3)^\omega$ proved by J.L. Lassez and Maher himself in [59]. On the other hand, full abstraction has been proved independently by several authors (see Maher [77], Gaifman and Shapiro [39] and Gabbrielli *et al.* [38]).

Here we present an equivalent proof based on the following result proved by Brogi *et al.* in [15]. Let P and Q be two programs; then, for any Herbrand interpretation M, it holds that:

$$M \models P \cup Q \Longleftrightarrow M \models P \text{ and } M \models Q \tag{1}$$

Hence, we have that any model M for the union of two programs is also a model for both the component programs. Furthermore, since two programs are $\equiv_3$-equivalent iff they have the same Herbrand models, from (1) above we have that:

$$[\![P \cup Q]\!]_3 = [\![P]\!]_3 \cap [\![Q]\!]_3$$

This result, in turn, serves as the basis for proving that $[\![\cdot]\!]_3$ is fully abstract as shown by the following

Theorem 2.1 (Full abstraction of $[\![\cdot]\!]_3$). Given two programs P and Q:

$$\forall R \; Ob(P \cup R) = Ob(Q \cup R) \Longrightarrow P \equiv_3 Q$$

PROOF. by contradiction. Assume that for every program R, $Ob(P \cup R) = Ob(Q \cup R)$ but $P \not\equiv_3 Q$. Then, there exists a model M of P which is not a model of Q. But then, there exists a ground instance $A \leftarrow B_1, ..., B_N$ of a clause in Q, such that $\{B_1, \ldots, B_N\} \in M$, and $A \notin M$. Let now R' be the program composed by the set of facts $B_1, \ldots, B_N$. Then $M \models P \cup R'$ since $M \models P$ and $B_1, \ldots, B_N \in M$. On the other hand, from $A \notin M$ and $M \models P \cup R'$, it follows that $P \cup R' \not\models A$ whence $A \notin Ob(P \cup R')$. But this is a contradiction since $Q \cup R' \models A$ and hence $A \in Ob(Q \cup R')$. □

The previous results do not apply to the operator of overriding-union. In fact, $[\![\cdot]\!]_3$ is not even $(Ob, \lhd)$-compositional.

Example 2.5. Let P and P' be the two following programs:

$$P = \left\{ \begin{array}{l} p(a) \leftarrow q(b) \\ q(b) \end{array} \right\} \qquad\qquad P' = \left\{ \begin{array}{l} p(a) \\ q(b) \end{array} \right\}$$

P and P' are obviously logically equivalent (they have the same models) and thus $P \equiv_3 P'$. Now, by taking $R = \{q(a)\}$, we have:

$$R \lhd P \;=\; \left\{ \begin{array}{l} p(a) \leftarrow q(b) \\ q(a) \end{array} \right\} \qquad\qquad R \lhd Q \;=\; \left\{ \begin{array}{l} p(a) \\ q(a) \end{array} \right\}$$

and these two program are no longer logically equivalent. Hence $\equiv_3$ is not a congruence for $\lhd$, and $[\![\cdot]\!]_3$ is not compositional.

It is shown in [20] that none of the invariants we have considered provides an adequate semantics for the type of program composition encompassed by $\lhd$. An interesting example is given by the two programs $\{p(a) \leftarrow p(b)\}$ and $\{p(b) \leftarrow p(a)\}$. These two programs are clearly not logically equivalent whereas they are indistinguishable with respect to $\equiv_{Ob,\lhd}$. Notice that the only *logical* semantics that would identify these two programs as equivalent seems to be the classical minimal Herbrand model semantics[2]. Yet, the choice of minimal models as invariant would then fall short of capturing the compositional properties of the union of clauses which is used to define the $\lhd$-composition.

We conclude by noting that the two remaining equivalences are strictly finer than $\equiv_3$ and thus that they cannot be fully abstract with respect to $\cup$.

2.3. Import and Export

In traditional modular languages [50], a module is defined as a collection of declarations and statements which constitute *closed* scope: identifiers *imported* from the external environment as well as identifiers to be *exported* must be explicitly declared. An identifier is imported by a module if it is used there but defined in some other module; it is exported by a module if it is defined there and used elsewhere. This way, the import/export designator attached to each identifier constrains the *visibility* of that identifier to the modules which import it, and hence, they allow forms of encapsulation and information hiding in the module system.

Obviously, these principles can be applied as well to modular *logic* languages. In this context, a designator can be thought of as attached to several syntactic entities: predicate names, constant/function names and data constructors in general. However, most of the currently published work on encapsulation in logic programming deals with the import/export of predicate names. Two notable exceptions are the module system of Sannella and Wallen [89] and the module facility provided by Gödel [46].

In this section we will emphasize the analysis of the import/export of predicate names and approach the case of constant and function symbols on more informal grounds[3].

---

[2] Another possible solution would be to consider the completed programs but, since the very idea of program composition is implicitly based on an *open* world assumption, the notion of completion in this context does not seem that reasonable.

[3] This is not to dismiss the latter case as less interesting or relevant than the former. To the contrary: the real point is that, while the idea of constraining the visibility of constant/function names seems natural, its rendition in our algebraic framework is not. In fact, it is in straight contrast with our initial assumption that the signature Σ of the constant and function symbols be fixed, and hence global and *visible*, for all the programs in the algebra.

Once the visibility of each predicate identifier has been established, there are two ways that the corresponding relation can be imported and/or exported within the module system. A module can either import/export *intension* of the relation, i.e. the clauses defining it, or its *extension*, – i.e. the tuples belonging to the relation or, equivalently, the atomic consequences of the clauses defining it. In section 2.1 we have already seen examples of both these mechanisms: the union operator can be used to implement a mechanism for importing/exporting all of a program's clauses; the closure operator as a mechanism for exporting all of the program's extensional knowledge (its atomic consequences).

The problem with these operators is that their granularity is too coarse for them to be used as effective software engineering tools. Following the approaches presented in [39, 14, 16, 86, 10], we show next that these operators can be generalized to build more sophisticated modular systems in which it is possible to specify more refined visibility rules as well as model import/export of a relation at the intensional and/or extensional levels.

2.3.1. Intensional View. An intensional mechanism for import/export has been studied by Gaifman and Shapiro in [39]. They interpret a logic module as a quadruple (P, Im, Ex, Int) where P is a logic program and $\{Ex, Im, Int\}$ is a partition of all the predicates of P into three disjoint sets such that no predicate of Im occurs in the head of a clause. Im defines the set of the module's imported predicates, Ex the set of exported predicates, while the internal predicates in Int are the predicates local to the module. The union of Im and Ex constitutes the *interface* of the module.

The composition of two logic modules $M_1 = (P_1, Im_1, Ex_1, Int_1)$ and $M_2 = (P_2, Im_2, Ex_2, Int_2)$ is a new logic module $M_3 = (P_3, Im_3, Ex_3, Int_3)$ such that:

- $P_3 = P_1 \cup P_2$,
- $Im_3 = (Im_1 \cup Im_2) \setminus (Ex_1 \cup Ex_2)$,
- $Ex_3 = (Ex_1 \cup Ex_2)$, and
- $Int_3 = (Int_1 \cup Int_2)$.

The two sets Ex_1 and Ex_2 are assumed to be disjoint. When this is not the case they are renamed to avoid any possible name clash. In [39], the authors present a compositional and fully abstract semantics for this type of module composition and show that the induced equivalence is an extension of logical equivalence.

2.3.2. Extensional View. Several other papers in the literature approach the problem at the extensional level. Here, we will survey some of these proposals (e.g. [14, 16, 10]), and discuss some further extensions.

We first need to extend the definition of the closure operator introduced in section 2.1.2, with two new arguments representing the set of atomic consequences which can be imported from/exported to other modules.

The new operator, denoted by $(P, Imp, Ex)^*$ models a selective form of closure whereby a module has visibility of (dually, makes visible) only those atoms whose predicate name belongs to Imp (dually Ex). Thus $(P, Imp, Ex)^*$ provides a very general and flexible operator that can be specialized in several ways. The complete encapsulation of a program is expressed by the formula $(P, \{\}, \{\})^*$, that declares that no predicate should be either imported or exported. On the other hand, open programs simply correspond to importing and exporting all formulas; their closure is given by $(P, \gamma(P), \gamma(P))^*$, where $\gamma(P)$ is the set of predicate symbols occurring in P. Moreover, $(P, \{\}, \gamma(P))^*$ corresponds to the * operator introduced in section 2.1.2, and $(P, Imp, \gamma(P))^*$ to the operator $closure(P, Imp)$ introduced in [14] which supports import declarations only.

A similar and essentially equivalent operator is studied by Fitting [32]. He introduces the enumeration operator $[P^{I_1,\ldots,I_n}_{O_1,\ldots,O_m}]$ as the formal counterpart of a module which has P as axioms, imports predicates $I_1, \ldots, I_n$ and exports predicates $O_1, \ldots, O_m$. According to our notation, Fitting's operator would be expressed as $(P, \{I_1, \ldots, I_n\}, \{O_1, \ldots, O_m\})^*$.

The denotation of the extended closure operator can be itself modeled in terms of the immediate-consequence operator. For any program P, the denotation of $(P, Imp, Ex)^*$ can be defined by the following equation:

$$[\![(P, Imp, Ex)^*]\!](I) = \Phi_{Ex}(\ (T_P + Id)^\omega(\Phi_{Imp}(I)))$$

The transformation Φ_S (where S is a set of predicate names) is a filter over Herbrand interpretations defined as follows:

$$\Phi_S(I) = \{A \in I \mid Pred(A) \in S\}$$

Thus Φ_{Imp} acts as an input filter by importing only those atoms whose predicate name belongs to Imp, and Φ_{Exp} acts as an output filter by exporting only those atoms whose predicate name belongs to Ex.

Similar import/export mechanisms are defined by O'Keefe in [86] where the elements of the algebra are called *breeze blocks* and *building bricks*. Breeze blocks correspond to import/export lists in conventional module systems, while building bricks correspond to separate logic programs. A breeze block is defined as a function over predicate symbols:

$$b : \Pi \mapsto \Pi \cup \perp$$

where $\perp$ represents falsity. An import/export declaration is defined in terms of the ad-hoc breeze block $include\{p_1, \ldots, p_n\}$. Here $p_1, \ldots, p_n$ are distinct predicate symbols and *include* is used for shutting out predicates that a module is not interested in, by naming only the predicates of interest. This behaviour is modeled by defining the breeze-block as follows:

$$include\{p_1, \ldots, p_n\} = \lambda p \begin{cases} p & \text{if } p \in \{p_1, \ldots, p_n\} \\ \perp & \text{otherwise} \end{cases}$$

This definition – lifted to interpretations – models an operator which corresponds directly to the filter function $\Phi_{\{p_1,\ldots,p_n\}}$. A program consisting of n modules, where module M_j imports predicates I_j and exports predicates E_j can be specified as follows:

$$\bigcup_{j=1}^{n} include\ E_j \circ M_j \circ include\ I_j$$

where $\circ$ is the composition operator discussed in section 2.1.2. For each program component, the first composition, $include\ E_j \circ M_j$, ensures that only exported predicates are made visible to other components. $M_j \circ include\ I_j$, in turn, makes sure that only the imported predicates are made visible within M_j. Hence, the denotation of each component can be modeled in terms of our closure operator by setting

$$[\![(include\ Ex) \circ P \circ (include\ Imp)]\!]_{O'Keefe}(I) = [\![(P, Imp, Ex)^*]\!](I)$$

for any Herbrand interpretation I.

A similar composition mechanism has been studied by Brogi in [10]. Again the idea is to couple encapsulation and information hiding with mechanisms for exporting predicate names. He defines a new binary operation, $P \prec Q$, which builds a module out of a pair of modules P and Q. P plays the role of the visible part of the module, Q the role of the hidden part.

In the composition $P \prec Q$, P is visible by other modules which thus are allowed to access its clauses. On the other hand, the hidden part Q cannot be accessed directly from the outside. The set of formulas which are provable in Q can be referred to only by the visible part P.

The intended semantics of the composition $P \prec Q$ is introduced in [10] in terms of an encapsulation operator. In practice, it can be written in our framework as follows:

$$[\![P \prec Q]\!](I) = [\![P]\!](I \cup [\![Q^*]\!])$$

Notice that $P \prec Q$ differs from the hierarchical composition $P \cup Q^*$ (section 2.1.2), since in this latter case the extensional knowledge of Q is not hidden from the outside.

As a final remark we should note that in most of the existing logic programming languages with modules, such as MProlog [60], Quintus Prolog [87], SICStus Prolog [90] the import and export facilities are typically introduced at the extensional level. This choice is, most probably, dictated by reasons of efficiency: importing (or exporting) the *extension* of a definition allows one to resolve statically (and thus compile) all the local references that occur in that definition.

2.3.3. Visibility Rules for Data Constructors. The module system for Prolog described by Sannella and Wallen in [89] is in several respects similar to that proposed by O'Keefe. *Structures* are the basic program components and play the same role as O'Keefe's bricks. However, O'Keefe's system is untyped and his scope mechanisms are applied only to the clauses of the bricks. Constant and function symbols are instead thought of as global. Conversely, in [89] Sannella and Wallen extend the scope rules of their language to apply also to the constant and function symbols of a module.

Gödel [46] shares several features with the module system of Sannella and Wallen. All symbols in Gödel are treated equally by the module system, thus Gödel supports import and export mechanisms for predicates names as well as for data structures and types. Each module is equipped with an *export* part and a *local* part. The export part specifies the module's interface. The export part begins with an `EXPORT` declaration and contains zero or more `IMPORT` declarations, together with other declarations for types and predicate signatures. The local part is not visible from the outside and models a form of encapsulation and information hiding. The local part begins with a `LOCAL` declaration and contains zero or more `IMPORT` declarations, together with other language declarations and statements.

Example 2.6. We borrow the following example of a module definition in Gödel from [46].

```
EXPORT          M.
IMPORT          Lists.
BASE            Day, Person.
CONSTANT        Monday, Tuesday, Wednesday, Thursday, Friday,
                Saturday, Sunday : Day;
                Fred, Bill, Mary : Person.
PREDICATE       Append3 : List(a)*List(a)*List(a)*List(a).

LOCAL           M
Append3(x,y,z,u) <- Append(x,y,w) & Append(w,z,u).
```

`M` has an export and a local part. The export part of `M` makes all the symbols it declares or imports available for use by other modules which import `M`. In the example, this is the case for the declarations of the bases (types) `Day` and `Person`, of the constants `Monday`, `Fred` etc. and of the predicate `Append3`. The declaration `IMPORT Lists` makes all the symbols exported by `Lists` visible in `M`. Any module which imports `M` automatically imports all the symbols exported by `Lists`. The local part of `M` contains the definition of predicate `Append3` which uses the definition of `Append` from `Lists`.

In a recent paper [45], Hill discusses an extension of the Gödel module system to account for parametrized modules. The main motivation is to increase re-usability: modules can be parametrized with respect to predicates and types, and different instances of a parametrized module can be obtained by importing it with different values for actual parameters. The module name which follows the keywords `EXPORT` and `LOCAL` consists now of an identifier with zero or *more symbols* as arguments.

Example 2.7. In the following module, `Trans` defines the transitivity relation over a generic type `Point` and a generic predicate `Connect`.

```
EXPORT          Trans(Point,Connect).
BASE            Point.
PREDICATE       Connect, Tr : Point*Point.

LOCAL           Trans(Point,Connect).
Tr(x,y) <- Connect(x,y).
Tr(x,y) <- Connect(x,z), Tr(z,y).
```

This module is *initial*: instances of `Trans(Point,Connect)` can be obtained by substituting new symbols for the parameters occurring in the module name. For example, the declaration `IMPORT Trans(Person, Parent)` where `Person` is a type and `Parent` a predicate defining the parent relation, imports an instance of the parametric module `Trans`.

3. MODULAR PROGRAMMING: BEYOND HORN CLAUSE LOGIC

For the development of large applications, the possibility to define separate program components and to combine them using mechanisms like those outlined in the previous section represents certainly a fundamental requirement. However, there are other properties that we should expect from a modular system. In fact, the composition operators we have outlined allow us to specify only the collection of modules that are to be used for evaluating a top-level goal. Once the modular configuration of the program has been set, there is no way that we can dynamically modify its structure and enforce the evaluation of a (sub)-goal to occur in a collection of modules different than the one associated with the top-level goal.

To get a richer notion of composition, what we need is to have the operators for building and composing modules act as built-in mechanisms which directly effect the language's evaluation procedure. This argument motivated the work of Miller in his seminal paper on this subject [69]. His idea was to consider languages which make stronger use of the logical connectives, and to use them as modular languages where the composition operators correspond directly to these connectives.

From a programming language point of view, the main challenge in this approach is to isolate, within the wide class of candidate languages at disposal, a language which exhibits the desired modular features and which is also amenable to efficient implementations. There is of course also the question of whether the semantics of the extended language can be defined without undermining the language's declarative reading. This latter aspect will be discussed thoroughly in section 4. In the present section we will instead concentrate upon a programming-oriented style of presentation. We will survey the different extensions explored in the literature and discuss their computational characterization as well as the programming paradigms they encompass.

3.1. Preliminaries

Proof Systems. The operational semantics of each language will be defined proof theoretically. The associated proof-relation will be presented in terms of a corresponding inference system in the sequent calculus.

It could be argued that a more direct presentation of the operational semantics could be given relying on a transformational approach as it has been done, for instance, in [79] and [82]. The semantics of the extended language could be defined in terms of a mapping from programs in the extended language to corresponding Horn-clause programs. We could then appeal to the theory of SLD-resolution to specify (and logically justify) the computational behaviour of the extended language. However, depending on the type of module-composition we are to model, this approach may or may not be adequate. In fact, as we will show later, a transformational (or static) semantics *is* inadequate to capture the dynamic flavour of some of the modular languages we will consider. For this reason, in the remainder of this section we will employ a dynamic presentation.

Sequents will be denoted as pairs of the form $\Delta \vdash \Gamma$ where the antecedent Δ and the succedent Γ stand for (possibly empty) sets of formulas. When $\Gamma \cap \Delta$ contains an atomic formula we will say that $\Delta \vdash \Gamma$ is an *initial* sequent. The intended interpretation of the sequent $\Delta \vdash \Gamma$ is that there exists a proof from the *antecedent* Δ to some of the formulas in the *succedent* Γ.

Proofs are defined constructively by composing *inference figures* (proof rules) of the form:

$$\frac{\textit{upper sequent}(s)}{\textit{lower sequent}}$$

When interpreted bottom up, these rules can be directly employed as the basis of a *goal-directed* proof system which uses them to rewrite each lower sequent into a corresponding set of upper sequents. Hence, we can think of the inference figures as instructions for an idealized interpreter: when fed with the sequent $\Delta \vdash \Gamma$, the interpreter will succeed if the sequence of rewritings leads either to an empty set of sequents or to a set of initial sequents; it will fail when no rule applies to a non-initial sequent.

This operational re-interpretation of proofs in the sequent calculus has been extensively studied by Miller *et al.* in [76]. The purpose of their research was to identify and isolate a (maximal) subset of this calculus that could be implemented in a programming language. The aforesaid property of *goal-directedness* is just one of the properties that a practical implementation of a proof-system should satisfy. The idea of goal-directed search can be in fact carried much further by imposing stronger requirements on the structure of a "good" proof: namely, that the inference rule applied at each sequent be uniquely determined by the top-level logical connective of (one of the formulas of) the succedent of the current sequent. This intuitive argument was formalized by Miller and his colleagues [76] in terms of the notion of *uniform proof*, "a *cut-free* proof in which *(i)* the succedent of each sequent is a singleton set of formulas and *(ii)* each occurrence of a sequent whose succedent contains a non-atomic formula is the lower sequent of the inference figure that introduces the formula's top level connective".

These general principles will provide the key for defining the operational meaning of the languages we will consider. Our sequents will have the simplified form $P \vdash G$, where P will denote the set of *program* formulas and G a *single* goal formula. The proof systems will satisfy the aforesaid principle of uniformity.

Sequent Proofs for Horn Clauses. Following the style introduced in section 2, we will describe the syntax of Horn Clauses by means of the three metalinguistic variables A, D and G defined by the productions we introduced there. The operational semantics of the language of Horn clauses can be described by means of the following proof system which we borrow from [76].

$$\text{(SUCCESS)}\quad \frac{}{P \vdash \top} \qquad\qquad \text{(AND)}\quad \frac{P \vdash G_1 \quad P \vdash G_2}{P \vdash G_1 \wedge G_2}$$

$$\text{(INSTANCE)}\quad \frac{P \vdash G[x/t]}{P \vdash \exists x G} \qquad\qquad \text{(BACKCHAIN)}\quad \frac{P \vdash G}{P \vdash A}$$

The proviso for (BACKCHAIN) is that there exists a closed instance $G \supset A$ of a clause in $[P]_\Sigma$. Here Σ denotes the set of constant and function symbols of the program and, as in section 2, $[P]_\Sigma$ denotes the closure of P under conjunction and instantiation over all the possible Σ-terms.

It is important to remark that the above rules only partially specify the course of action of a real interpreter for logic programming as they do not specify what the result of the computation should be. Note in particular that the proof of an existentially quantified goal does not produce a *witness* substitution, as it is customary in logic programming. Instead, it results into a potentially infinite non-deterministic or-branching where each branch corresponds to a particular witness *guessed* by the interpreter. The same remark applies as well to the definition of the (BACKCHAIN) rule. This substitution-free notion of derivation, which we inherit from [76] has two advantages: it is completely general, as it does not commit to any definition of unification, and, for this very reason, it allows us to delegate substitutions and unification to implementation issues which will be dealt with in more detail in section 5.

Fixed Points. An alternative and more abstract presentation of the operational semantics will also be given in terms of a fixed point reconstruction of the proof-theoretic definitions. This will help us identify some of the distinguishing features of the different composition mechanisms and clarify the relations between them. The formal framework employed for the fixed point presentation will be a *possible-world* semantics based on Kripke-like interpretations [55].

Kripke interpretations provide an ideal setting for modeling – semantically speaking – the dynamic type of modular composition underlying the languages we will consider. The link between the proof-theoretic and the fixed point approaches will be established by introducing a notion of *weak satisfiability* ($\Vdash$) for goals in the set-theoretic structures (the possible worlds) we will associate with our programs.

What is important to remark here is that both the proof-theoretic and the fixed point presentations are to be understood as inherently *operational* specifications. As in [71], we will not assume any "*a priori* relation between $\vdash$ and other logical notions of derivation or provability"; nor we will assume any relation between weak satisfiability and other notions of entailment in any logical system. The analysis of these relations will be approached and discussed later in section 4.

3.2. Embedded Implications: A Foundation for Modular Programming

We now wish to build on top of Horn Clauses and assume a more complex syntax for a G-formula where we now allow occurrences of implication goals. The new sets of D and G-formulas are defined by the following (now mutually recursive) definitions of the meta-linguistic variables D and G:

$$\begin{array}{lcl} D & ::= & A \mid D \wedge D \mid \forall x D \mid G \supset A \\ G & ::= & \top \mid A \mid G \wedge G \mid \exists x G \mid D \supset G \end{array}$$

A language with this structure has been originally studied in the literature by Gabbay and Reyle [37, 36] in the attempt to enrich logic programming with mechanisms for hypothetical reasoning[4]. They observed that the embedded implication $D \supset G$ can be read as the hypothetical statement asserting that the truth of the consequent G of the implication is subject to the truth of the antecedent D. On this basis, they proposed the following operational interpretation: querying a program P with the goal $D \supset G$ amounts to requesting that the proof of G be drawn from P by assuming D as a further hypothesis.

It was Miller, though, who firstly proposed a notion of modular programming based on this use of embedded implications. In [71], he formalized the operational

[4]In [36] Gabbay considers an extension of this language that allows negative G-formulas.

semantics of implication goals by extending the provability relation $\vdash$ for Horn Clauses with the following inference rule:

$$\text{(AUGMENT)} \qquad \frac{P \cup D \vdash G}{P \vdash D \supset G}$$

We will henceforth use the subscript $\supset$, and write $\vdash_{\supset}$ to denote the proof predicate obtained by extending $\vdash$ with the (AUGMENT) rule.

When D and G are closed formulas, the above rule provides a direct formalization of the deduction theorem: to prove $D \supset G$, assume D and prove G from $P \cup D$. However, the (AUGMENT) rule works just as well when D and G contain occurrences of free variables. Consider in fact the case of a sequent $P \vdash_{\supset} \exists x(D(x) \supset G(x))$ where x is free in $D(x)$ and $G(x)$. By virtue of the treatment of existential quantifiers introduced earlier, a proof of $\exists x(D(x) \supset G(x))$ will be constructed by first guessing a closed term t (non-deterministically) and then by attempting a proof of the new sequent $P \vdash_{\supset} D(t) \supset G(t)$. Note how the choice of a unification-free notion of derivation helps ease the definition of the (AUGMENT) rule. Consider in fact what would happen if we replaced the existentially quantified variable x with a *logical* variable, say X. Now, in the attempt to find a proof for $P \vdash_{\supset} D(X) \supset G(X)$, the clauses $D(X)$ would again be added to P, but then whenever we produced a substitution for X both the formulas in $D(X)$ and the goal $G(X)$ would need updating consistently.

The study of a substitution semantics and its implementation for the language of embedded implications (and its variations) will be studied in detail in section 5. Until then, we will appeal to it informally when presenting some of the following examples. Our next goal is in fact to show how embedded implications can be taken as the basis for implementing modules in a logic language.

3.2.1. Implementing Modules. When D is a set of universally quantified clauses, the implication goal $D \supset G$, in a program P, can be interpreted operationally as a request to load the clauses in D before attempting G, and then discard them after the derivation for G succeeds or fails. Note that this type of composition between D and P is the same as that encompassed by the algebraic composition of two programs by union (see section 2). The fundamental difference is that here D and P are composed *dynamically* as the result of evaluating $D \supset G$.

This dynamic form of composition supports naturally a modular approach to writing code. Modules can be introduced as named collections of clauses and programs can be structured as collections of modules each one dedicated to answer a specific class of queries. Cross-referencing between modules and module-composition can then be accounted for relying on the workings of embedded implications. If, in module M, the answer to a goal G requires that the clauses of module M_1 be loaded, then we will simply enforce the evaluation of G in the composition of M and M_1 by means of the implication goal $M_1 \supset G$.

This programming discipline permits also to model forms of encapsulation and scoping over the clauses contained in a program. Consider for instance the case of a conjunctive goal $(M \supset G_1) \wedge G_2$. Here the clauses in M are only available for evaluating G_1 whereas they are hidden during the evaluation of G_2. A more concrete example of the use of embedded implications as a scope mechanism is illustrated by the following definition of the list-reverse predicate which we borrow from [71].

$$\begin{array}{rcl} \forall x,y\ rev(x,y) & \leftarrow & \{\ \forall l\ rv_1([\,],l,l). \\ & & \quad \forall x,l_1,l_2,k\ rv_1([x|l_1],l_2,k) \leftarrow rv_1(l_1,l_2,[x|k]) \\ & & \}\ \supset rv_1(x,y,[\,]) \end{array}$$

This two-argument reverse works in linear time. As usual, it is defined in terms of an auxiliary predicate, rv_1, which uses a third argument as accumulator. The notable difference is that the auxiliary definition is now encapsulated in the embedded implication. The effect is that the clauses for rv_1 are local to the definition of rev which is now the only predicate which has access to them.

A notion of *parametric* module can also be accounted for in this framework. The fact that we allow embedded implications of the form $\exists x(D(x) \supset G(x))$ suggests that the clauses defining a module can contain free variables. Thus, as proposed in [71], modules can be referred to by names which have an arity and take arguments just as predicate names. If $D(\boldsymbol{x})$ denotes a set of clauses whose free variables are in the list $\boldsymbol{x}$, then $M_D(\boldsymbol{x})$ will be the module name used to refer to $D(\boldsymbol{x})$. The arguments for a module name designate the parameters of that module. Different instances will be then obtained by providing values for the module's parameters and, correspondingly, by instantiating the free variables of the associated set of clauses. An application of this idea has been already exemplified in section 2.3.3. Other examples will be described more fully in section 3.6, where we discuss an Object-Oriented extension of logic programming based on the use of embedded implications.

What we show next, instead, is how free variables in an embedded implication can be employed to model powerful forms of variable inheritance between nested scopes.

Example 3.1. The following program, proposed in [70], provides a refined version the previous definition of the list-reverse predicate:

$$\begin{array}{rcl} \forall x,y\ rev(x,y) & \leftarrow & \{\ rv_2([\,],y). \\ & & \quad \forall x,l_1,l_2\ rv_2([x|l_1],l_2) \leftarrow rv_2(l_1,[x|l_2]) \\ & & \}\ \supset rv_2(x,[\,]) \end{array}$$

Notice that y now occurs free in the first clause of rv_2. It is interesting to look at the behaviour of this program more closely. Assume that we query the program with the goal $rev(x,Y)$. If x is instantiated to a ground list and (the logical variable) Y is unbound, this goal triggers the evaluation of $rv_2(x,[\,])$. This, in turn, recursively traverses the list x and reverses it. Upon returning from the call $rv_2(x,[\,])$, the free variable y gets instantiated to the reversed list and this binding is finally propagated back up to instantiate Y in the original goal $rev(x,Y)$.

The nature of embedded implications as a scope mechanism will be discussed further in section 3.3. We now turn to consider an alternative and more abstract characterization of the operational workings of embedded implications. As already anticipated, the result will consist of a fixed point reconstruction of the proof-theoretic setting discussed so far.

3.2.2. A Fixed point Semantics for Embedded Implications The framework for this reconstruction is given by a Kripke-like semantics. The idea, due to Miller [69, 71], is to model the behaviour of embedded implications by viewing a program as a form of computation in a set of possible worlds. At this level, the dynamic evolution of the proof space, which is peculiar of embedded implications, is captured by having the possible worlds of a Kripke interpretation act as partial interpretations of a program. Each world will be used to interpret the set of clauses corresponding to the "image" of the program at a given stage of the computation.

The notion of Kripke-like interpretation employed by Miller is a special case of a more general definition which will be introduced in section 4. Assume we have fixed ahead the signature Σ, Π of the program, let B be the associated Herbrand base and $P(B)$ be its power-set. An interpretation is defined as a mapping $I : W \mapsto P(B)$ where W is the set of all possible programs and I is monotone on W: $\forall w_1, w_2 \in W, w_1 \leq w_2 \;\Rightarrow\; I(w_1) \subseteq I(w_2)$. Interpretations defined according to these principles will be referred to as $\Im$-interpretations to distinguish them from Herbrand interpretations and from general Kripke interpretations.

Associated with this notion of $\Im$-interpretation, Miller defines the following relation of *weak satisfiability* for a closed G-formula, in an $\Im$-interpretation I *at* program w:

$$
\begin{array}{llll}
I, w \models\!\models \top & & & \\
I, w \models\!\models A & \text{iff} & A \in I(w) & (A \text{ atomic}) \\
I, w \models\!\models G_1 \wedge G_2 & \text{iff} & I, w \models\!\models G_1 \text{ and } I, w \models\!\models G_2 & \\
I, w \models\!\models \exists x G & \text{iff} & I, w \models\!\models G[x/t] & \text{for a } \Sigma\text{-term } t \\
I, w \models\!\models D \supset G & \text{iff} & I, w \cup D \models\!\models G &
\end{array}
$$

This definition has a natural intuitive reading. An $\Im$-interpretation can be thought of as a collection of partial interpretations indexed by sets of program clauses. The relation $I, w \models\!\models G$ means that the goal G holds in the interpretation associated with the set of clauses w. The case of embedded implications parallels the corresponding proof-rule: to interpret an implication goal $D \supset G$ in I at w, we interpret G in the interpretation indexed by the *extended* program $w \cup D$.

The relation of weak satisfiability serves as the basis for establishing the link between the proof-theoretic and the fixed point descriptions. The goal of the latter is to compute an $\Im$-interpretation I^* such that G is operationally derivable from P ($P \vdash_{\supset} G$) if and only if $I^*, P \models\!\models G$. For this purpose, $\Im$-Interpretations are accommodated in a complete lattice $(\Im, \sqsubseteq)$ where $\sqsubseteq$ is defined as the following ordering:

$$I_1 \sqsubseteq_{\Im} I_2 \iff \forall w \in W \; I_1(w) \subseteq I_2(w).$$

The bottom element of this lattice is denoted by $I_\perp$ and defined by setting $I_\perp(w) = \emptyset$ for all $w \in W$. The join of two $\Im$-interpretations is the $\Im$-interpretation defined as $(I_2 \sqcup I_2)(w) = I_1(w) \cup I_2(w)$. The $\Im$-interpretation I^* with the desired properties is computed as the least fixed point of a continuous immediate-consequence operator on $\Im$-interpretations. The definition of this operator relies on the notion of weak satisfiability we have just introduced[5]:

$$T(I) \;=\; \lambda w \; \{A \mid A \leftarrow G \in [w]_\Sigma \text{ such that } I, w \models\!\models G\}$$

The notation $[w]_\Sigma$ is used here as in section 2, to stand for the set of D-formulas obtained from w by closure under conjunction and instantiation. The continuity of T, proved in [71], guarantees that there exists the least fixed point $lfp(T)$ and that $lfp(T)$ can be computed as the interpretation $T^\omega(I_\perp) = \sqcup_{k\in\omega} T^k(I_\perp)$. The equivalence between the fixed point and the proof-theoretic definitions of derivability is then established by the following result proved by Miller.

Theorem 3.1. [71] *For any program P and any closed (ground or existentially quantified) goal G:*

$$P \vdash_{\supset} G \quad \textit{if and only if} \quad T^\omega(I_\perp), P \models\!\models G$$

In section 4, we will present a stronger result for Miller's computational interpretation of embedded implications. What we will show there is that the proof relation encompassed by $\vdash_{\supset}$ is sound and complete with respect to the notions of provability and entailment in intuitionistic logic. Before doing so, however, we now move on to study other interpretations of embedded implications which have been proposed in the literature as extensions or variations of the one we have just surveyed.

3.3. Embedded Implications and Lexical Scoping

Although the previous characterization of implication appears to be quite natural, it is by no means the only possible one. In fact, from a programming language point of view, the notion of scope encompassed by the (AUGMENT) rule appears to be rather weak. Consider, in this regard, the evaluation of an atomic goal A in a program P containing the clause $A \leftarrow D \supset G$. After backchaining on that clause and reducing the embedded implication, we are left with the evaluation of G in $P \cup D$. Assume now that the next backchaining step selects a clause from P and let G' be the body of that clause. Notice that, by virtue of the definition of (AUGMENT), at this stage the clauses of D are still accessible to G'. Hence, in the proof of G' we will be able to use not only the clauses in P but also those provided by D. But this implies that the meaning of the clauses of the surrounding scope P depends on the definitions coming from the inner scope D. Furthermore, this dependency is inherently *dynamic*: each call to a predicate in P will be associated with different definitions depending on the sequence of embedded implications which leads to consider that call during the proof.

This observation led other authors to study alternative characterizations of embedded implications in the attempt to capture stronger notions of scope. The first proposal in this direction was due to Giordano *et al.* in [42] and in [43]. Later

[5] This definition suggests a deeper technical justification for using *weak* satisfaction in this context. The following observation was pointed out to us by Dale Miller [74]. The problem is that if the T operator is defined using a full possible-world notion of satisfaction, then T is not monotone. In particular, having $\models$ denote Kripke's S4-validity operator (see section 4.3) consider defining $T(I)(w) = \{A | G \leftarrow A \in [w] \text{ and } I, w \models G\}$. That this definition of T is not monotone is revealed by the following counter-example. Let $J_\perp$ be the S4-interpretation that attaches the empty set of atoms to all worlds. Now consider the two programs $w_0 = \{(p \supset q) \supset r\}$ and $w_1 = \{p, (p \supset q) \supset r\}$: we have $T(J_\perp)(w_0) = \{r\}$, $T(J_\perp)(w_1) = \{p, r\}$, and $T^2(J_\perp)(w_0) = \{\}$. Hence, $J_\perp \sqsubseteq T(J_\perp)$ but $T(J_\perp) \not\sqsubseteq T(T(J_\perp))$ and T is not monotonic.

work led then Miller himself [70] and, independently, Moscowitz and Shapiro [82] to obtain similar results by resorting to limited forms of higher order universal quantification over embedded implications. This second approach will be considered later in the paper (section 3.7). Here we concentrate on the solution proposed in by Giordano *et al.* in [42].

Their idea is to model a notion of *lexical* scope which allows one to determine the set of formulas available for reducing each goal by simply inspecting the syntactic structure of a program. The language they considered has the same structure as that proposed by Miller but the embedded implications are interpreted differently. The idea is similar to that underlying the use of the closure operator described in section 2: the difference, as for the dynamic scope-rule of Miller, is that here the composition occurs dynamically. The new proof rule for the sequent $P \vdash D \supset G$ composes D with the set of atomic formulas which are provable from P.

In analogy to what we have done in section 2, we denote with P^* the set of the atomic consequences of P:

$$P^* \quad = \quad \{A \mid A \text{ is atomic and } P \vdash A\}$$

The following "abstract" rule for embedded implications formalizes the previous intuition:

$$\frac{D \cup P^* \vdash G}{P \vdash D \supset G} \tag{1}$$

Notice how this is different from the semantics of embedded implications encompassed by (AUGMENT): the meaning of P, the set of its atomic consequences through $\vdash$, is computed before extending P with the new clauses coming from D. The remarkable consequence is that the dependency between P and D works now in one single direction: the body of a clause of D depends on P but not vice-versa. All of the references to a predicate in the outer scope P can thus be bound *lexically* to the definitions occurring in that scope. Hence, a lexical rule of scope can be accounted for quite elegantly in this framework.

The previous definition is admittedly idealized: to construct a proof for $P \vdash D \supset G$, an interpreter will have to "guess" all the atomic formulas provable from P which are needed to construct a proof for G. However, it is easy to show how the proof system defined by the rules (AND), (INSTANCE), (BACKCHAIN) and (1) can be implemented in terms of an equivalent (and more effective) proof system. The trick is to allow a more complex structure for sequents and to have the antecedent of a sequent be structured as a *stack* (to be contrasted with *set*) of clauses. The idea is due to Giordano *et al.* and the following definition of the proof predicate $\vdash_{\overline{\mathrm{stk}}}$ can be found in [42].

Let $P_1, \ldots, P_n$ denote sets of program clauses and let $S = P_n \mid \cdots \mid P_1$ denote a stack of programs (P_n being the top of the stack).

$$(\mathrm{AND}_{\mathrm{stk}}) \quad \frac{S \vdash_{\overline{\mathrm{stk}}} G_1 \quad S \vdash_{\overline{\mathrm{stk}}} G_2}{S \vdash_{\overline{\mathrm{stk}}} G_1 \wedge G_2} \qquad (\mathrm{INSTANCE}_{\mathrm{stk}}) \quad \frac{S \vdash_{\overline{\mathrm{stk}}} G[x/t]}{S \vdash_{\overline{\mathrm{stk}}} \exists x G}$$

$$(\mathrm{BACKCHAIN}_{\mathrm{stk}}) \quad \frac{P_i \mid \cdots \mid P_1 \vdash_{\overline{\mathrm{stk}}} G}{P_n \mid \cdots \mid P_1 \vdash_{\overline{\mathrm{stk}}} A} \qquad (\mathrm{AUGMENT}_{\mathrm{stk}}) \quad \frac{D \mid S \vdash_{\overline{\mathrm{stk}}} G}{S \vdash_{\overline{\mathrm{stk}}} D \supset G}$$

The proviso for ($\mathrm{BACKCHAIN}_{stk}$) is that the clause $G \supset A$ used to backchain on

A belongs to P_i, the top-most component of the stack in the upper sequent. The notion of initial sequent introduced in section 3.1 can be easily reformulated here by taking as initial any sequent $P_n \mid \cdots \mid P_1 \vdash_{\mathrm{stk}} \Gamma$ such that $(\cup_{i=1,\dots n} P_i) \cap \Gamma$ contains an atomic formula.

The program stack provides a dynamic representation of the scoped structure of the embedded implications occurring in the clauses of the program. The key to understand the workings of the proof predicate $\vdash_{\mathrm{stk}}$ is in the definition of (AUGMENT$_{stk}$) and (BACKCHAIN$_{stk}$). An application of (AUGMENT$_{stk}$) corresponds to a push of the new scope D on top of the current stack. D will be popped off the stack once G succeeds or fails. The selection of a matching clause for a goal A on (BACKCHAIN$_{stk}$) has a dual effect: it shrinks the program stack and reduces the definitions available for subsequent backchaining steps to the clauses which occur at the same nesting level as that of the clause used to backchain on A. This is how the lexical scope rule encompassed by $\vdash_{\mathrm{stk}}$ is captured.

The different behaviour of the proof predicates $\vdash_{\supset}$ and $\vdash_{\mathrm{stk}}$ is illustrated in the following simple example.

Example 3.2. Consider the program $P = \{p \leftarrow q\}$ and the goal $G = q \supset p$. It is easy to see that $P \not\vdash_{\mathrm{stk}} G$ whereas $P \vdash_{\supset} G$. The following steps show that $P \vdash_{\supset} G$.

$$\begin{array}{rcc} p \leftarrow q \vdash_{\supset} q \supset p & \text{only if} & \text{(AUGMENT)} \\ q,\, p \leftarrow q \vdash_{\supset} p & \text{only if} & \text{(BACKCHAIN)} \\ q,\, p \leftarrow q \vdash_{\supset} q & & \end{array}$$

The last sequent is proved by a further application of (BACKCHAIN) using the atomic clause q coming from the embedded implication. Conversely, for $\vdash_{\mathrm{stk}}$, the sequence of steps we obtain is

$$\begin{array}{rcc} p \leftarrow q \vdash_{\mathrm{stk}} q \supset p & \text{only if} & (\text{AUGMENT}_{stk}) \\ q \mid p \leftarrow q \vdash_{\mathrm{stk}} p & \text{only if} & (\text{BACKCHAIN}_{stk}) \\ p \leftarrow q \vdash_{\mathrm{stk}} q & \text{only if} & (\text{BACKCHAIN}_{stk}) \end{array}$$

and there is no clause for reducing q.

Example 3.3. The following example illustrates how the proof predicate $\vdash_{\mathrm{stk}}$ can be used to model visibility according to a lexical rule of scope. Consider the following program:

$$\begin{array}{lcl} anc(X,Y) & \leftarrow & parent(X,Y). \\ anc(X,Y) & \leftarrow & parent(X,Z), anc(Z,Y). \\ & & \\ parent(a,b). & & \\ parent(b,c). & & \\ parent(d,e). & & \\ & & \\ test(X,Y) & \leftarrow & \{parent(a,d).\} \supset anc(X,Y). \end{array}$$

The program is structured into two nested blocks: the enclosing one, containing the definitions for $anc/2$ and $parent/2$ and the inner one containing a definition for $parent/2$. Now consider querying this program with, say, $test(a, X)$. It is easy to verify that evaluating this goal produces only the two bindings X/b and X/c as the additional clause defining $parent/2$ in the inner block is not accessible to the corresponding calls in the outer block.

Lexically scoped Implications: fixed point semantics. In [42] the authors present a fixed point semantics for this scope mechanism using a construction similar to that proposed by Miller. The notable difference is that they employ a conventional Herbrand semantics in contrast to Miller's possible-world setting. Associated with a program, they define a mapping from the lattice of the program's Herbrand interpretations to itself. Given two Herbrand interpretations, I and X, and a program P, the mapping is denoted with $T_{P,I}$ and is so defined:

$$T_{P,I}(X) \quad = \quad I \cup \{A \mid A \leftarrow G \in [P]_\Sigma \ \text{ and } \ X \mathrel{|\!\approx} G\}$$

Here $\mathrel{|\!\approx}$ denotes the relation of weak satisfiability for a goal in a Herbrand interpretation defined below.

$$\begin{array}{llll}
X \mathrel{|\!\approx} \top & & & \\
X \mathrel{|\!\approx} A & \text{iff} & A \in X & (A \text{ atomic}) \\
X \mathrel{|\!\approx} G_1 \wedge G_2 & \text{iff} & X \mathrel{|\!\approx} G_1 \ \text{ and } \ X \mathrel{|\!\approx} G_2 & \\
X \mathrel{|\!\approx} \exists x G & \text{iff} & X \mathrel{|\!\approx} G[x/t] & \text{for a } \Sigma\text{-term } t \\
X \mathrel{|\!\approx} D \supset G & \text{iff} & T^{\omega}_{D,X}(\emptyset) \mathrel{|\!\approx} G &
\end{array}$$

The intuitive reading of the definition is the following. The set I in $T_{P,I}$ represents the interpretation associated with the outer scope of P, whereas X is the interpretation we are associating with P itself. The lexical form of scope attributed to embedded implications is reflected in the definition of satisfiability for implication goals. The interpretation X, which we initially associated with P, becomes the interpretation for the outer scope of D (P itself) and the empty set is the new interpretation we associate with D. Thus, informally, X provides the approximation of the set of the atomic consequences of the outer scope P and this approximation is used to compute (an approximation of) the interpretation associated with the nested scope D. The well-foundedness of this construction follows from the fact that, although mutually recursive, $T_{P,I}$ and $\mathrel{|\!\approx}$ are both defined inductively on the nested structure of a program.

The continuity of $T_{P,I}$, proved in [43], guarantees the existence of the least fixed point $lfp(T_{P,I})$, which is again computed as the limit $T^{\omega}_{P,I}(\emptyset) = \bigcup_{k<\omega}\{T^{k}_{P,I}(\emptyset)\}$. That this limit provides a sound and complete semantics is shown by the following theorem proved in [43].

Theorem 3.2. [43] *For any program P and any closed (ground or existentially quantified) goal G the following holds:*

$$P \vdash_{\overline{\text{stk}}} G \quad \textit{if and only if} \quad T^{\omega}_{P,\emptyset}(\emptyset) \mathrel{|\!\approx} G$$

We will show later (section 4.3) how this fixed point construction has been used in [42, 43] and [41] as an intermediate step to prove a soundness and completeness result for $\vdash_{\mathrm{stk}}$ with respect to entailment in S4-modal logic. What we show now, instead, is that an equivalent fixed point construction can be obtained using the possible-world setting proposed by Miller.

Let $(\Im, \sqsubseteq)$ be the lattice of $\Im$-interpretations as defined by Miller. Again, we denote with T a mapping from $\Im$ onto itself defined as follows:

$$T(I) \quad = \quad \lambda w \ \{A \mid A \leftarrow G \in [w]_\Sigma \text{ such that } I, w \models\!\!\models^* G\}$$

What is new here is the choice of the relation $\models\!\!\models^*$ of weak Kripke satisfiability. The new relation coincides with $\models\!\!\models$ for all the cases except (not surprisingly) for embedded implications. The truth of $D \supset G$ in I at w is now defined as follows:

$$I, w \models\!\!\models^* D \supset G \quad \text{iff} \quad I, I(w) \cup D \models\!\!\models^* G$$

This definition should be contrasted with the corresponding case of the weak Herbrand satisfiability $\approx$. $I(w)$ plays here the same role as X there: $I(w)$ represents an approximation of the atomic consequences provable from the outer scope w of D.

The proof of the continuity of the mapping T can be carried out in the exact same way as for the original operator defined by Miller. Hence, the least fixed point of T is well defined and can be again computed as the limit $\Im$-interpretation $T^\omega(I_\perp)$.

We show now that this fixed point construction is equivalent to that proposed by Giordano and her colleagues in [42]. To our knowledge, the proof of the following result has not been presented earlier in the literature.

For any Herbrand interpretation I, let I^* denote the corresponding program (consisting of the ground formulas of I).

Theorem 3.3. For any program P, closed goal G and Herbrand interpretation I,

$$T^\omega_{P,I}(\emptyset) \approx G \quad \textit{if and only if} \quad T^\omega(I_\perp), P \cup I^* \models\!\!\models^* G$$

PROOF. OUTLINE. The following two properties can be proved by induction on the level of nested occurrences of the connective $\supset$ in P and G:

(a) For any $\Im$-interpretation I and program w, $I(w) \approx G$ iff $I, w \models\!\!\models^* G$.

(b) For any Herbrand interpretation I, $T^\omega_{P,I}(\emptyset) = T^\omega(I_\perp)(P \cup I^*)$.

Now take $w = P \cup I^*$ and $I = T^\omega(I_\perp)$. Then, from (b), we have that $I(\omega) = T^\omega_{P,I}(\emptyset)$. Hence the claim follows by applying (a). □

The equivalence between the two semantics follows now as an immediate corollary of the previous result.

Corollary 3.1. For any program P and closed goal G

$$T^\omega_{P,\emptyset}(\emptyset) \approx G \quad \textit{if and only if} \quad T^\omega(I_\perp), P \models\!\!\models^* G$$

PROOF. Apply theorem 3.3 with $I = \emptyset$. □

3.4. From Open to Closed Scope Mechanisms

Now that we have a characterization of embedded implications as a lexical scope mechanism, it is easy to develop an even stronger notion of scope. Note, in this regard, that both the interpretations of an embedded implication we have outlined so far are inherently *open*: the meaning of the nested scope D introduced by the embedded implication $D \supset G$ depends on the (the meaning) of outer scope associated with the goal $D \supset G$ itself.

It should however be clear how to account for a notion of closed scope in this framework. We simply need to tailor the proof rule for embedded implications so as to break any dependency between nested scopes. The following definition satisfies this requirement:

$$\frac{D \vdash G}{P \vdash D \supset G}$$

Proving the embedded implication $D \supset G$ from P amounts now to prove G in a new program, D, which inherits no knowledge from P. This behaviour captures precisely the semantics of the *demo* predicate defined by Bowen and Kowalski in [9]. A similar reconstruction can be found in [49, 47] where Hodas and Miller use linear logic to partially account for the *demo* predicate.

In terms of a possible-world semantics, we have a corresponding new notion of weak satisfiability. The new definition reflects the change of context encompassed by last proof rule for $\supset$.

$$I, w \models D \supset G \quad \text{iff} \quad I, D \models G$$

The truth of $D \supset G$ in the $\Im$-interpretation I at program w is now tested by moving to a new interpretation, indexed by D, where no information about the outer scope of D is assumed available. This is how the behaviour of implication as a closed scope mechanism is captured in the fixed point framework.

3.5. Contextual Logic Programming: Implication and Overriding

There is an independent perspective wherefrom embedded implications have been studied in the context of modular logic programming. The idea, which led Monteiro and Porto in [78] to the definition of *Contextual Logic Programming* (CxLP in the following) is again inspired by the interpretation of implication as a mechanism of scope but differs from the previous characterizations substantially. According to its original definition, the interpretation of an implication goal in CxLP models again a "lexical" notion of scope: the novelty introduced by CxLP is that, in the evaluation of the implication goal $D \supset G$ (the *extension* goal $D \gg G$ according to the terminology of CxLP) the extension of the search space is non-monotonic. The new definitions coming from the nested scope D *override* the corresponding definitions provided by the outer scope.

The proof rule for the connective $\gg$ can be introduced abstractly using a construction similar to that followed for the static scope mechanism described in the previous section. Namely:

$$\frac{D \lhd P^* \vdash G}{P \vdash D \gg G} \tag{2}$$

This definition should be contrasted with the algebraic composition discussed in section 2 based on the operators of overriding-union $\lhd$ and closure *. By virtue of the non-monotonic extension of the search space associated with $\gg$, we obtain a new mechanism of scope: according to the above rule, the nested scope D depends on the outer scope P only for those definitions which are not local to D. The "visibility" rules affecting the behaviour of extension goals are thus more selective than those encompassed by the embedded implications of Giordano *et al.* in [42]: non-local definitions will be used for reducing a goal only when there is no definition for that goal in the local scope.

Not surprisingly, we can obtain a more effective characterization of the above rule by means of the same artifice used earlier to describe the proof predicate $\vdash_{\overline{\mathrm{s}}\mathrm{tk}}$. In fact, the provability relation for CxLP, which we will denote with $\vdash_{\gg}$, can be formalized as originally proposed by Monteiro and Porto in terms of the same proof rules used for $\vdash_{\overline{\mathrm{s}}\mathrm{tk}}$. The overriding semantics of $\gg$ is modeled by imposing a new – and much more stringent – proviso for the backchain rule. In

$$(\mathrm{BACKCHAIN}_{\gg}) \quad \frac{P_i \mid \cdots \mid P_1 \vdash_{\gg} G}{P_n \mid \cdots \mid P_1 \vdash_{\gg} A}$$

we impose that P_i be the top-most component of $P_n \mid \cdots \mid P_1$which contains a definition for the predicate symbol of A. Note the difference between this definition and the corresponding one given for $\vdash_{\overline{\mathrm{s}}\mathrm{tk}}$. In that case, the choice of P_i was non-deterministic: hence, all the definitions provided by $P_n \mid \cdots \mid P_1$ can be used for backchaining. To the contrary, here we commit to the top-most definition, thus modeling the effect of overriding.

The overriding semantics of $\gg$ can be used to model the typical lexical scope rule found in conventional programming language.

Example 3.4. This is how the second version of the list-reverse predicate of section 3.2 (example 3.1) would be written in Contextual Logic Programming.

$$\begin{array}{lcl} \forall x, y\ rev(x,y) & \leftarrow & (\ rev([\ ],y). \\ & & \quad \forall x, l_1, l_2\ rev([x|l_1],l_2) \leftarrow rev(l_1,[x|l_2]) \\ & &)\ \gg rev(x,[\ ]) \end{array}$$

Note that there is no need to rename the nested clauses defining $rev/2$ as we did in example 3.1. The overriding semantics of $\gg$ ensures that these clauses are the only available to evaluate the call $rev(x,[\ ])$.

In [78], Monteiro and Porto use their operator to model structuring mechanisms more general than the scope rule we have exemplified here. They give names to modules and allow the same module to have multiple occurrences in different extension goals. The effect is that, differently from [42], a reference to a predicate occurring in a module cannot be bound *statically* to the definitions occurring in the outer scope. Although conceptually (and semantically) equivalent, this usage of extension goals raises some interesting questions relative to their implementation. This and related issues will be discussed more fully in section 5.

Extensions of CxLP: overriding and dynamic scope. Motivated and inspired by the original definition of CxLP, an alternative semantics for extension goals is proposed by Mello *et al.* [67] in the attempt to couple a notion of dynamic scope with the overriding semantics proposed by Monteiro and Porto.

As a matter of fact, the formal framework we have developed so far is rich enough to provide a formal definition of this new interpretation of extension goals. The corresponding semantics can in fact be described in terms of the following proof rule which couples the dynamic flavour of the (AUGMENT) rule with the overriding semantics of CxLP.

$$\frac{D \lhd P \vdash G}{P \vdash D \gg G}$$

Here, as in our first definition of Miller's proof predicate $\vdash_{\supset}$, P denotes a set (as opposed to a *stack*) of clauses. The new proof relation, denoted with $\vdash_{+\!\gg}$, is obtained from the definition of $\vdash_{\supset}$ by replacing (AUGMENT) with the above rule. The extension of the search space associated with $\gg$ is non-monotonic as in CxLP but it retains the dynamic flavour of (AUGMENT). The new set of clauses used in the proof of G is the result of the *overriding-union* (to be contrasted with the *union*) of D and P. Hence, the dependency between D and P is again bi-directional, as for (AUGMENT), but constrained by virtue of the restricted form of union provided by $\lhd$. An example of how this mechanism can be used to implement useful forms of dynamic scope will be discussed in the next subsection.

In [67], the authors introduce an equivalent proof system for $\vdash_{+\!\gg}$ by extending the stack-based proof system for $\vdash_{\gg}$. The definition we have used here appears to be more elegant and concise. However, the proof system of [67] is more general since it makes it possible to integrate into a unique framework all the mechanisms of scope we have outlined so far.

From the previous proof-theoretic presentation, it should by now be clear how the two proof relations $\vdash_{\gg}$ and $\vdash_{+\!\gg}$ can be expressed in terms of two corresponding definitions of weak satisfiability in $\Im$-interpretations. We simply need to tailor the interpretation of the connective $\gg$ so as to mimic the overriding semantics underlying the composition operator $\lhd$. This can be accomplished by having $I, w \models$ $\models D \supset G$ be defined respectively as:

$$I, w \VDash_{\gg} D \supset G \;\text{ iff }\; I, D \lhd I(w) \VDash_{\gg} G$$

to model the lexical scope rule encompassed by $\vdash_{\gg}$ and

$$I, w \VDash_{+\!\gg} D \supset G \;\text{ iff }\; I, D \lhd w \VDash_{+\!\gg} G$$

for the dynamic scope rule $\vdash_{+\!\gg}$.

What should be discussed here is whether or not the non-monotonic extension of the search space encompassed by the operator $\gg$ affects the continuity of the associated mapping T on $\Im$-interpretations. As it turns out, in the dynamic case (and similarly for $\vdash_{\gg}$) the continuity proof for T goes through in the exact same way as the corresponding proof for the original operator defined by Miller (the continuity of essentially the same operators was proved in [18] and similarly in [80]). Hence, we will again be able to compute the least fixed point of T as the limit $\Im$-interpretation $T^{\omega}(I_{\perp})$ and to use it to establish the desired soundness and completeness result for the corresponding proof procedures. The proof of the following theorem is similar to the corresponding one found in [71] and is therefore omitted.

Theorem 3.4. Where $\VDash_{\gg}$ and $\VDash_{+\!\gg}$ denote the two satisfiability relations defined above, and $\vdash_{\gg}$ and $\vdash_{+\!\gg}$ the two corresponding proof predicates, the following equivalences hold true for any program P and closed goal G.

$$P \vdash_{\gg} G \quad \textit{if and only if} \quad T^{\omega}(I_{\perp}), P, \VDash_{\gg} G$$
$$P \vdash_{+\!\gg} G \quad \textit{if and only if} \quad T^{\omega}(I_{\perp}), P, \VDash_{+\!\gg} G$$

In a recent paper [81], Monteiro and Porto present a similar semantics for a language that extends Contextual Logic Programming with constructs for parametric modules, import mechanisms and encapsulation. The framework for their fixed point construction is again a possible-world semantics – a *situation semantics* according to the terminology of [81]. The notable difference is that in [81] the construction has a *finitary* character, in that the structure used to interpret a program has finitely many modules (as opposed to the infinitely many programs needed for Miller's semantics), and it is homomorphic in the sense that the denotation of a context is determined in a compositional way from the denotations of the component modules.

3.6. Inheritance and message-passing: Object-Oriented Logic Programming

In section 3.2 we briefly alluded to the use of embedded implications to model an Object-Oriented (OO) extension of logic programming.

The study of the integration of the OO and logic programming aradigms has been approached from two different perspectives. The first dates back to the work of Aït-Kaci and Nasr on LOGIN [2], a logic language with built-in inheritance. In LOGIN classes and objects are represented as compound terms whose arguments designate the objects' attributes. A labeling schema over terms is employed to logically link objects into *isa* hierarchies. Attribute inheritance is then achieved by overloading unification to take the term-hierarchy into account when attempting to match two terms. The work on LOGIN inspired a number of other proposals pursuing this idea and extending it with higher-order features to accommodate methods within complex terms (see [26] and [54] for examples).

The second approach, which we consider here, is based on the idea of representing an object as a *first-order* logic theory. This view inspired McCabe's *Class Template Language* [62] and has been adopted by many other authors in the literature ([35, 34, 40, 66, 48] among others). In his Class Template Language McCabe proposes a logical reconstruction of the OO paradigm where objects are interpreted as sets of axioms defining the objects' attributes and methods. The same idea can be exploited to model an OO extension of logic programming using embedded implications.

Classes are introduced as parametric modules whose parameters act as (stateless) instance variables in conventional OO languages. As pointed out by McCabe, a parametric module is interpreted declaratively as the denotation of the (infinitely many) instances obtained by substituting corresponding terms for the module's parameter.

Message passing can then be accounted for directly using embedded implications. A message-sent $O : G$, requesting that G be evaluated in object O, is modeled by having ":" cause a change of context from the current object (module) to the object O. The interpretation of embedded implications discussed in section 3.4 works just as well here. If O is the current object, the proof rule for $\overline{O} : G$ is simply

$$\frac{\overline{O} \vdash_{oo} G}{O \vdash_{oo} \overline{O} : G}$$

Inheritance Class hierarchies can be modeled by combining algebraic composition and message-sents as proposed, for instance in [13] and [19]. The super-class relation *isa* is expressed there as a *meta-level* axiom O_1 *isa* O_2 stating that O_2 is O_1's immediate ancestor in the *isa* hierarchy. Accordingly, the message-sent $O : G$ causes the evaluation of G to take place not simply in O but in the program obtained by the (algebraic) composition of O with all of its ancestors in the object hierarchy. Depending on the type of algebraic composition associated with the *isa* relation, all the different forms of inheritance discussed in section 2.1.3 can be accommodated in this context.

Consider the hierarchy O_n *isa* $O_{n-1} \ldots$ O_2 *isa* O_1 and the message-sent $O_j : G$. A system with dynamic (overriding) inheritance is modeled by means of the following proof rule (H is the current object hierarchy):

$$\frac{O_j \lhd O_{j-1} \lhd \ldots \lhd O_1 \vdash_{oo} G}{H \vdash_{oo} O_j : G}$$

Static inheritance can be accounted for by simply changing the above definition to:

$$\frac{(O_j \lhd (O_{j-1} \lhd (\ldots \lhd O_1^*) \ldots)^*)^* \vdash_{oo} G}{H \vdash_{oo} O_j : G}$$

In the style of the conventional OO systems, McCabe models the behaviour of dynamic inheritance by using explicit references to $self$. The message-sent $self : G$ requests that G be evaluated in the hierarchy associated with the object that received the last message-sent. Monteiro and Porto [79] resort instead to a clever use of parametric modules. Here, as suggested by Mello in [65] we can rely on the workings of the $\lhd$-composition we have outlined in section 2.1.3.

Example 3.5. Consider the following example written using McCabe's Class Template Language.

bird	isa	*animal*
tweety	isa	*bird*
$human(S, A)$	isa	*animal*
peter	isa	$human(male, 30)$
mary	isa	$human(female, 42)$

$human(S, A)$ is the name of a parametric class. *bird isa animal* states that *bird* is a sub-class of *animal* and similarly *peter isa* $human(male, 30)$ states that *peter* is an instance of $human(S, A)$. When we say that *bird* is a sub-class of *animal* (or that $human(S, A)$ is a sub-class of *animal*) we are stating that whatever holds for animals, and is not overridden, holds also for birds (respectively, for humans). In other words, theory *bird* inherits from theory

$animal$. A possible definition for the above classes and instances is the following:

$$animal: \left[\begin{array}{l} mode(walk). \\ mode(run) \leftarrow self : no\_of\_legs(2). \\ mode(gallop) \leftarrow self : no\_of\_legs(4). \end{array}\right.$$

$$bird: \left[\begin{array}{l} mode(fly). \\ no\_of\_legs(2). \\ covering(feather). \end{array}\right.$$

$$human(S, A): \left[\begin{array}{l} sex(S). \\ age(A). \\ no\_of\_legs(2). \\ likes(logic) \leftarrow sex(male), age(Age), less\_than(Age, 40). \\ likes(logic) \leftarrow sex(female). \end{array}\right.$$

$$tweety: \quad [no\_of\_wings(2).$$

$$peter: \quad [likes(music).$$

$$mary \quad [likes(painting).$$

The call $self : g$ realizes the self-reference mechanism we have discussed before: it causes the proof of g to be performed in the tip node of the current hierarchy independently of the class where the call occurs. In our framework, the above hierarchy can be realized in terms of the following compositions:

$peter \lhd human(male, 30) \lhd animal$
$mary \lhd human(female, 42) \lhd animal$
$tweety \lhd bird \lhd animal$

Messages to $self$ need not be explicit thanks to the workings of the $\lhd$ composition. Thus $animal$ can be simply defined as follows:

$$animal: \left[\begin{array}{l} mode(walk). \\ mode(run) \leftarrow no\_of\_legs(2). \\ mode(gallop) \leftarrow no\_of\_legs(4). \end{array}\right.$$

If we now ask whether $peter$ likes logic, by evaluating the message-sent $peter : likes(logic)$, the goal $likes(logic)$ gets evaluated in the composition:
$peter \lhd human(male, 30) \lhd animal$.
The answer will be "no" since $peter$ redefines $likes/1$ to state that he only likes music.

Objects with state The above characterization of objects as logic theories does not account for any notion of state. In [62], McCabe suggests that the change of state for an instance can be simulated by creating new instances. Other proposals [34] simulate the changes of state by means of *assert* and *retract* but this approach lacks any logical foundation. A refined solution has been proposed by Chen and Warren in [27], where *intensional variables* are introduced to keep track of state changes without side effects. In other proposals [35, 66] state change is simulated by means of unification and recursion within a concurrent logic programming framework. In

[3] and [28], multi-headed clauses are used for similar purposes whereas a (goal) continuation-passing style of programming is used in [48, 49, 47]. The interested reader should refer to these references for a fuller description of these issues and of the different proposals.

3.7. Lexical Scoping as Universal Quantification

All of the modular extensions we have considered so far rely on the (stringent) assumption that all the free variables of an embedded implication be existentially quantified. In this section we consider a language which allows G-formulas (and forcefully, embedded implications) to be universally quantified and study the scope mechanisms that arise by virtue of this extension.

Uniformity and First-Order Universal Quantification. Let's first consider the extent to which the combination of the logical connectives at our disposal should be allowed in the language. We first note that, as discussed by Miller *et al.* [76], an unrestricted use of these connectives should be prevented in order for the language to satisfy the computational requirement of uniformity we have introduced at the outset. Consider in fact the class of D-formulas defined by the productions:

$$D \quad ::= \quad A \mid D \wedge D \mid \forall x D \mid B \supset A$$

where B denotes an arbitrary first-order formula and A, as before, stands for an atomic formula. The formulas in this class are known as *Harrop Formulas* for they satisfy the condition, introduced by Harrop [44], that they contain no strictly[6]positive occurrence of disjunctions or existential quantifiers. Harrop formulas enjoy an important computational property proved by Harrop: if a formula B, arbitrary, follows (intuitionistically) from a set of Harrop formulas P, then there exists a sequent proof for $P \vdash B$ whose last step introduces the top-level connective of B. In [76], the authors observe that this property can be exploited to make a proof involving these formulas, we quote, "uniform at the root" but not uniform. The problem arises from the fact that one such proof might contain sequents whose antecedents are not sets of Harrop formulas. In fact, if we allow an arbitrary formula to occur in the body of a clause, then that body might contain an embedded implication $B \supset G$ with B again arbitrary. Hence, an application of the augment rule on $P \vdash B \supset G$, will generate $P, B \vdash G$ as its upper sequent and B might not be a Harrop formula.

This consideration motivated Miller and his colleagues [76, 85] to consider a subclass of the Harrop Formulas, the class of *Hereditary Harrop Formulas*, which are so defined as to ensure that the antecedent of an embedded implication be itself a Harrop Formula[7]. This guarantees that the Harrop property is satisfied by any sequent introduced by an application of the augment rule and, consequently, that the proofs involving these formulas can be made uniform. The set of G and D formulas which meet these conditions were introduced in [85] by means of the

[6]Here *strictly* refers only to the top-level occurrences: in the formula $((a \wedge b) \supset p) \supset r$, the disjunction has a positive occurrence that is not strictly positive.

[7]In [85] and [76] they also consider a higher-order extension of these formulas which will be discussed in section 3.7.2. Until then, the language we consider is to be understood as strictly first-order.

following definition.

$$\begin{array}{lcl} G & ::= & \top \mid A \mid G \wedge G \mid \exists x G \mid \forall x G \mid D \supset G \\ D & ::= & A \mid D \wedge D \mid \forall x D \mid G \supset A \end{array}$$

A language which shares the same structural properties for D- and G-formulas has also been considered by McCarty[8][64] in his approach to *clausal intuitionistic logic*[9]and by Bonner *et al.* in [7].

The quantificational extension to the language N-Prolog of Gabbay and Reyle [37] falls also in this class. They all show that this extended logic can be used for modeling powerful forms of knowledge-representation and common-sense reasoning. Our interest here is, instead, in the use of Hereditary Harrop Formulas in the context of modular programming.

Sequent Proofs for Universally Quantified Goals. There are two ways that we can interpret a universal quantifier, either intensionally or extensionally. Correspondingly, we have two possible ways to attempt a proof for a universally quantified goal $\forall x G(x)$. Interpreting the quantifier extensionally amount to attempting a proof for $\forall x G(x)$ by showing that G holds for every element in any given domain. In contrast, if we assume an intensional interpretation, then a proof for $\forall x G(x)$ will be constructed by first instantiating $G(x)$ with a "new" object, say c, and then attempting a proof for the new goal $G(c)$.

This constructive flavour of the intentional interpretation appeals implicitly to the intuitionistic definition of proof. The relations between the intuitionistic and the intensional interpretations will be addressed more fully in section 4. We next show how the intensional interpretation can be formalized proof-theoretically in the sequent calculus.

The idea is to enrich the structure of a sequent so as to make it explicit which is the domain over which the individual variables should range. The new sequents will have the form $\Sigma; P \vdash G$ where Σ represents the signature of the current domain, and P and G respectively a set of D-formulas (clauses) and a goal formula over the signature Σ. The definition of a proof system for Hereditary Harrop formulas is obtained by simply attaching the signature Σ to the sequents occurring in the proof rules. An additional rule will be needed to handle the case of universally quantified goals. The following definition is adapted from the corresponding one proposed in [70].

$$(\text{AND}_\forall)\ \frac{\Sigma; P \vdash_\forall G_1 \quad \Sigma; P \vdash_\forall G_2}{\Sigma; P \vdash_\forall G_1 \wedge G_2} \qquad (\text{INSTANCE}_\forall)\ \frac{\Sigma; P \vdash_\forall G[x/t]}{\Sigma; P \vdash_\forall \exists x G}$$

$$(\text{BACKCHAIN}_\forall)\ \frac{\Sigma; P \vdash_\forall G}{\Sigma; P \vdash_\forall A} \qquad (\text{AUGMENT}_\forall)\ \frac{\Sigma; P \cup D \vdash_\forall G}{\Sigma; P \vdash_\forall D \supset G}$$

$$(\text{GENERIC}_\forall)\ \frac{\Sigma + \{c\}; P \vdash_\forall G[x/c]}{\Sigma; P \vdash_\forall \forall x G}$$

There are some important remarks. In the definition of (BACKCHAIN$_\forall$), we are assuming not only that there exists in P a clause of the form $G \supset A$, but also that

[9]In [64] McCarty considers an extension of this language that allows *negation rules* of the form $p \leftarrow \neg Q$ and $\neg p \leftarrow Q$.

this clause be a Σ-formula. Similarly, in (INSTANCE$_\forall$), we require that the term t which replaces x in G, be a Σ-term. In contrast, in (GENERIC$_\forall$), the constant c is *new*, i.e. it is required to not be an element of Σ.

The effect of (GENERIC$_\forall$) on the signature Σ parallels that of (AUGMENT$_\forall$) on the program P: it causes the extension of the current signature with a new and fresh constant symbol. It is important to note, in this regard, that the choice of a unification-free definition of the above rules helps substantially ease the treatment of existentially quantified variables. Suppose we replace the existential variable x in the lower sequent of (INSTANCE$_\forall$) with the logical variable X and delegate the construction of the witness t to the unification algorithm. Then, we must also ensure that the term t which will eventually instantiate X be a term over the signature Σ associated with the sequent that introduced X. This might be a non-trivial task for, at the time X gets bound, the signature Σ might have been augmented by several applications of (GENERIC$_\forall$). Unification should therefore be instructed to not instantiate X with any term containing the new constants (*eigen-variables*) introduced by (GENERIC$_\forall$) after X itself has been introduced. The description of how this can be accomplished is contained in [70, 73] and similarly in [63, 31, 83]. Details can be found in section 5. In the remaining of this section we will present, instead, two applications of universally quantified embedded implications for modeling powerful forms of scope over the constants and the clauses of a program.

3.7.1. Local Constants and Abstract Data Types In section 3.2 we have seen that the free variables occurring in the clauses introduced by an embedded implication can be used to exchange values between nested scopes. We now show that universally quantified embedded implications provide a way of introducing constants with local scope. This use of universal quantifiers was firstly addressed by Miller and Nadathur in [85] and then further studied by Miller in [70]. Consider the sequent $P \vdash_{\forall} \exists x \forall y(D(y) \supset G(x))$. A proof for this sequent would first substitute a Σ-term t for x, then introduce a new constant c for y and finally prove $G(t)$ in the $\Sigma + c$ program $P \cup D(c)$. If we replace x with a logical variable, say X, a proof of $\forall y(D(y) \supset G(X))$ would be required to bind X to a term which does not include the new constant introduced for y. Thus, that constant would be local and encapsulated into $D(c)$. Note also that when y does not occur free in $G(x)$, $\exists x \forall y(D(y) \supset G(x))$ is (intuitionistically and classically) equivalent to $\exists x(\exists y(D(y)) \supset G(x)$. Thus a local constant can be formally thought of (and implemented as well) as a "variable that is existentially quantified over the clauses in a module [85]". This property of existentially quantified variables can be exploited to account for data abstraction and encapsulation in ways similar to those based on existential types in the quantified λ-calculus of [24]. The following example was proposed by Miller in [70].

Example 3.6. A stack can be implemented as an abstract data type by means of the following definition:

$$\begin{array}{ll} \exists e \exists st & empty(e) \\ & \forall x, s(push(x, s, st(x, s))) \\ & \forall x, s(pop(x, st(x, s), s)) \end{array}$$

Here "e" and "st" act as the stack constructors and are hidden to the programs using this definition of the stack data type. A typical example of how this implementation would be used is the following. Suppose we are to write a program for evaluating expressions in reverse polish notation. An easy way to write that program is to use a stack for storing the intermediate values that arise during the computation. A possible definition would be:

$$\begin{array}{lcl} eval(expr, val) & \leftarrow & empty(s), ev(expr, val, s). \\ ev([\,], val, s) & \leftarrow & pop(val, s, \_). \\ ev([e|r], val, s) & \leftarrow & push(e, s, ns), \cdots \end{array}$$

If *stack* is the name for the stack definition, we can evaluate any expression by calling $stack \supset eval(expr, val)$. Note that the stack constructors are hidden to the call whereas the stack elements are visible to it. Hence the value returned by $ev/3$ will be bound to val as we expect.

3.7.2. Local Predicates: A Higher-Order view of Lexical Scope. We conclude this section with a brief outline of the higher-order use of universal quantification proposed by Miller and Nadathur [70, 85] and by Moscowitz and Shapiro [82] to model a lexical scope rule over the clauses of a program.

The extension of Miller and Nadathur was based on the language of *Higher-Order Hereditary Harrop Formulas* whereas Moscowitz and Shapiro defined it by introducing the notion *disjunctive lexical clauses.* The two languages share the same restriction on the higher-order use of universal quantification. Universal quantifiers are permitted over the predicate symbols that occur in the embedded implications, but the top-level symbol of a clause is required to be a constant. Thus, they both allow clauses of the form $\forall x\ p(x) \leftarrow \forall q\ (q(x) \supset r(x))$, and disallow clauses like $\forall p, x\ p(x) \leftarrow q(x) \supset r(x)$ whose head's predicate symbol is universally quantified[10].

During a proof, universally quantified predicates can be treated in much the same way as universally quantified variables. The proof of a goal $\forall p\ p(\cdots)$ is attempted by first generating a new predicate symbol, say p_c, and then trying to prove the goal $p_c(\cdots)$. This allows the realization of a lexical scope rule with an overriding behaviour which is similar to that encompassed by the operator $\gg$ introduced in Contextual Logic Programming.

Example 3.7. Consider again the program of example 3.1. The way this program would be written using a universally quantified embedded implication has been shown by Miller in [70]:

$$\begin{array}{lll} \forall x, y\ rev(x, y) \leftarrow \forall rv & (& (rv([\,], y). \\ & & \quad \forall x, l_1, l_2\ rv([x|l_1], l_2) \leftarrow rv(l_1, [x|l_2]) \\ & &) \supset rv(x, [\,])\ ) \end{array}$$

Now, a proof $rev([a, b, c], r)$, would first generate a new symbol for rv, say p, load the corresponding clauses for p, and finally prove the goal $p([a, b, c], r)$ in the extended program. This definition of reverse should be contrasted with the corresponding definition that uses the $\gg$ operator of CxLP (cf. example 3.4).

[10] As noted in [76] this restriction avoids them the need of full higher-order unification and, as such, it keeps this set of formulas still amenable to efficient implementations.

The difference is that here the overriding semantics associated with $\gg$ is captured by generating a new name for the universally quantified predicate rv.

4. LOGICAL FOUNDATIONS FOR MODULAR PROGRAMMING

The modular extensions presented in the previous section have been studied primarily in terms of their operational semantics. We have described the different impact of each composition mechanism and outlined the programming features that can be accounted for in the different languages. The goal of this section is to explore the logical foundations of the extensions, i.e. to study whether the operational characterizations have an equivalent formulation in terms of corresponding *logical* notions of provability and entailment. This correspondence represents a well-known result in logic programming: the proof of equivalence between SLD-resolution and entailment in classical logic dates back to the seminal papers of Apt, Kowalski and van Endem [91, 4].

Here, the first question that arises is whether we can still appeal to classical logic to attempt a logical reconstruction of the extensions. Not surprisingly the answer depends on the extension we consider. In [82], Moscowitz and Shapiro formalize the semantics of their lexical logic programs by interpreting them as higher-order intuitionistic formulas. However, they also present an equivalent *classical* first-order semantics by defining a transformation ψ mapping lexical logic programs onto corresponding (and equivalent) logic programs. Monteiro and Porto used a similar technique in [79] to justify the proof rules they introduce for describing the workings of their inheritance system.

A different approach has been considered for most of the remaining proposals. In [71] and [76] Miller and his colleagues use a proof-theoretic argument to show that the operational semantics of Hereditary Harrop formulas finds its logical counterpart in the intuitionistic proof theory. A similar result is then obtained by Miller [72] in terms of entailment in intuitionistic logic. The intuitionistic model theory is also at the basis of the logical reconstruction proposed by Gabbay [36], by McCarty [64] and by Bonner *et al.* [7]. Finally, Giordano and Martelli [41] show that the semantics of lexically scoped embedded implications can be equivalently expressed in terms of entailment in S4-modal logic.

In this section we will survey the intuitionistic and modal reconstructions outlined above. The interested reader should refer to the work of Moscowitz and Shapiro and of Monteiro and Porto for more details about the transformational approach. Also, in the following discussion, we will emphasize the description of the model-theoretic frameworks and approach the proof theoretic reconstruction of [76] only on intuitive grounds.

4.1. Intuitionistic Proof Theory for Modular Logic Programming

Miller's choice of intuitionistic logic was initially motivated by the observation that "classical provability is too strong for specifying the behaviour of the (AUGMENT) rule [71]". The following example, which we borrow from [71], provides a convincing argument in favour of the previous statement. If $\vdash_{\supset}$ were equivalent to classical provability, then an interpreter implementing it, should be able to construct a proof for p starting from the D-formula $(p \supset q) \supset p$. That this should be the case follows by observing that p is classically entailed by $(p \supset q) \supset p$ as $(p \supset q) \supset p$ holds only when p does. Conversely, it is immediate to see that there is no way to prove the sequent $(p \supset q) \supset p \vdash p$ using a proof system based on the (AUGMENT) rule.

In [71], Miller proved that the operational notion of derivability encompassed by $\vdash_{\supset}$ corresponds to provability in intuitionistic logic. More precisely, where $\vdash_I$ denotes the intuitionistic proof predicate, he proved that:

Theorem 4.1. [71] *For any program P and closed goal G, $P \vdash_{\supset} G$ iff $P \vdash_I G$*

This result relies on the close correspondence between the two notions of derivation. As a matter of fact, the soundness part of the theorem is proved easily by noting that the proof-rules defining $\vdash_{\supset}$ are a subset of the intuitionistic sequent calculus. As for the opposite direction, the completeness result is established in [71] by showing that any intuitionistic sequent-proof can be turned into an equivalent proof that uses only the inference rules defining $\vdash_{\supset}$.

In [76] a corresponding result was proved for the language of Hereditary Harrop Formulas. Here the equivalence between $\vdash_{\forall}$-derivability and intuitionistic provability can be justified intuitively by means of the following observation. The intensional interpretation of universal quantifiers encompassed by $\vdash_{\forall}$ appeals implicitly to the intuitionistic definition of proof for a universally quantified formula. The intensional reading of $\forall x\, G(x)$ corresponds in fact to the question of whether G holds for *any* possible object; to prove this we construct a generic witness, c, on which we make no assumption, not even that it belongs to the domain of interest. Correspondingly, given a domain D and a formula $A(x)$ stating a property on the elements of D, an intuitionistic proof for $A(\alpha)$, where $\alpha \in D$, is a constructive mapping – a *method* – $\pi(\alpha)$ which transforms the hypothesis $\alpha \in D$ into the thesis $A(\alpha)$. An intuitionistic proof for $(\forall x \in D)A(x)$ is then defined as a proof $\pi(\alpha) : \alpha \mapsto A(\alpha)$ which is intuitionistic and, most importantly, *variable-free*, that is such that, for any other element $\beta \in D$, $\pi(\alpha/\beta)$ is the constructive mapping $\pi(\alpha/\beta) : \beta \in D \mapsto A(\beta)$. The equivalence between the intensional and the intuitionistic interpretations should now be clear: the "newness" requirement on the constant c in the former corresponds to the "freeness" requirement on α in the latter.

4.2. Intuitionistic Model Theory for Modular Logic Programming

We start by introducing the intuitionistic notions of satisfaction and model. The first modeling structure for first-order intuitionistic logic was proposed by Kripke in [55]. Here we will briefly outline the basics of this approach following the notation introduced by Fitting in [33] and used also by Bonner *et al.* and Miller respectively in [7] and [72].

An intuitionistic structures is a quadruple of the form $M = \langle S, \leq, \phi, Dom\rangle$ where S is a non-empty set, the set of *worlds* (or *substates*), $\leq$ is a reflexive and transitive relation on S, ϕ is a mapping from elements of S to sets of ground atomic formulas (the facts that are true in the associated substate) and Dom is a domain function mapping each substate to the set of terms over the signature associated with that substate. The mapping ϕ is required to be monotone, i.e. $\phi(s_1) \subseteq \phi(s_2)$ whenever

$s_1 \leq s_2$ and, for any $s \in S$, $\phi(s)$ is assumed to contain only formulas built over the signature of $Dom(s)$.

In the remainder of this section we will assume that the signature associated with each substate be the set of constant and function symbols occurring in the terms associated to that substate.

Truth in an intuitionistic structure is defined relative to its substates. We will therefore consider the truth value of a formula ψ at a particular state s of some intuitionistic structure M and write $s, M \models_I \psi$ if ψ is satisfied in M at s. In general the statement $s, M \models_I \psi$ is taken to be false if ψ contains symbols which do not belong to the signature of $Dom(s)$. This proviso applies to all the cases of the following definition which we borrow from [7].

Definition 4.1. [Intuitionistic Satisfiability] Let M be an intuitionistic structure and s be a substate of M. Let ψ denote an arbitrary first order formula (containing no occurrences of negation and disjunction).

$$
\begin{array}{llll}
s, M \models_I \top & & & \\
s, M \models_I A & \text{iff} & A \in \phi(s) & (A \text{ atomic}) \\
s, M \models_I \psi_1 \wedge \psi_2 & \text{iff} & s, M \models_I \psi_1 \text{ and } s, M \models_I \psi_2 & \\
s, M \models_I \exists x \psi & \text{iff} & s, M \models_I \psi[x/t] & t \in Dom(s) \\
s, M \models_I \forall x \psi & \text{iff} & r, M \models_I \psi[x/t] & \forall\, r \geq s,\ t \in Dom(r) \\
s, M \models_I \psi_1 \supset \psi_2 & \text{iff} & r, M \models_I \psi_1 \;\Rightarrow\; r, M \models_I \psi_2 & \forall\, r \geq s
\end{array}
$$

Definition 4.2. [Models] We say that an intuitionistic structure M is a model for a formula ψ and we write $M \models_I \psi$ iff $s, M \models_I \psi$ holds true for all the substates s of M such that the set of function and constant symbols occurring in ψ belong to the signature of s.

To establish the truth of a formula at a given state s, we require that the formula be satisfied not only at s but also at all the possible $r \geq s$. This property of the intuitionistic notion of truth is left implicit for some cases in the above definition but it can easily shown to be implied by it if we assume, as we do, that the mapping ϕ is monotone. It is this very same property that makes the above definition capture the constructive flavour of the intuitionistic notion of proof. Note in particular the cases of implicative and universally quantified formulas. In both these cases the definition of truth differs from the classical one. To assert $D \supset G$ at a given state, we require that at any later state where we can assert (and prove) D we can also assert G. Consider for instance the formula $p \supset q$ and the structure $M = \langle S, \leq, \phi, Dom \rangle$ where $S = \{s_1, s_2\}$, $\phi(s_1) = \emptyset$ and $\phi(s_2) = \{p\}$. Now, $s_1 \models p \supset q$ classically but $s_1, M \not\models_I p \supset q$, for $s_2 \geq s_1$ and $s_2, M \models_I p$ but $s_2, M \not\models_I q$.

The argument for universally quantified formulas is similar. Here $\models_I$ models the intensional interpretation of universal quantifiers peculiar of intuitionistic logic. To assert $\forall x\ p(x)$ at state s we require not simply that $p(t)$ holds for any choice of term t in $Dom(s)$, but rather that $p(t)$ holds for any new element that can ever be introduced in the domain of discourse.

Based on this definition of satisfiability, we finally have a corresponding notion of entailment: for any two (sets of) formulas ψ_1 and ψ_2 we say that ψ_1 entails intuitionistically ψ_2, and we write $\psi_1 \models_I \psi_2$, iff $M \models_I \psi_2$ for all the intuitionistic models M of ψ_1.

Having set the appropriate formal framework we now turn to the proof that Miller's operational semantics is sound and complete with respect to intuitionistic entailment.

4.2.1. Embedded Implications and Dynamic Scope. We first restrict to the language of existentially quantified embedded implications discussed in section 3.2. This will allow us to assume a simpler definition of intuitionistic structures: for any program P, we will assume that in any given intuitionistic structure the domain of each world is the same and coincides with the Herbrand universe built over the constant and function symbols of P. This assumption, which helps simplify the result of completeness, will be shown to not involve any loss of generality given the restrictions we impose on the use of universal quantifiers. Proving the soundness and completeness result amounts to proving the following equivalence.

Theorem 4.2. For any program P and closed goal G, $P \vdash_{\supset} G$ iff $P \models_I G$.

Note how this is different from the corresponding equivalence between operational and fixed point semantics discussed in section 3.2.2. The $\Im$-interpretation computed using Miller's fixed point construction is – structurally – an intuitionistic interpretation for the program. As a matter of fact, it is also an intuitionistic model, as it will become apparent later. However, in that context we didn't even define what a model for a formula should be; we simply demanded that one "point" in that $\Im$-interpretation, the point associated with the program P, be a "weak model" for all of (and only) the *goals* which are provable from P. Hence, there was no account of entailment in that result.

The proof of theorem 4.2 could be derived indirectly relying on the equivalence between operational and intuitionistic provability established by theorem 4.1 and by noting that intuitionistic provability is sound and complete with respect to intuitionistic entailment. Here we will appeal to this latter result to sustain the proof that $P \vdash_{\supset} G \;\Rightarrow\; P \models_I G$ and we will instead prove the opposite implication by a direct and more constructive argument.

A similar result has been proved by Gabbay in [36]. Here we present an equivalent but refined proof inspired by a similar completeness proof contained in [7]. What we obtain is not only a proof of equivalence but also a constructive method for defining a canonical model for any program.

For any program P, let U_P denote the Herbrand universe of P. We define the *canonical* model M_P of P as the quadruple $\langle S, \subseteq, \phi, Dom\rangle$ with the following structure:

$$\begin{array}{lcl} S & = & \{s \mid s \text{ is a set of } D\text{-formulas}\} \\ \subseteq & = & \text{set inclusion} \\ \phi(s) & = & \{A \mid A \text{ is atomic and } P \cup s \vdash_{\supset} A\} \\ Dom(s) & = & U_P \;\; \text{(constant)} \end{array}$$

Note that the structure computed by the fixed point iteration used by Miller (cf. section 3.2.2) is isomorphic to M_P. Hence, as shown below, that structure is not only an intuitionistic interpretation, but indeed a model for the program.

The two following properties of M_P can be proved inductively on the structure of D- and G-formulas:

(a) $M_P \models_I G \;\Rightarrow\; P \cup s \vdash_{\supset} G$ for all $s \in M_P$.
(b) M_P is an intuitionistic model for P: for any $D \in [P]$, $M_P \models_I D$.

From (a) and (b) above, we have an immediate proof of the following completeness result.

Theorem 4.3. For any program P and closed goal G, $P \models_I G \;\Rightarrow\; P \vdash_{\supset} G$.

PROOF.

$$\begin{array}{rcl}
P \models_I G & \Longleftrightarrow & M \models_I P \;\Rightarrow\; M \models_I G \quad \text{for any intuitionistic structure } M \\
\text{by (b)} & \Longrightarrow & M_P \models_I G \quad \text{being } M_P \models_I P \\
\text{by (a)} & \Longrightarrow & P \cup s \vdash_{\supset} G \quad \forall s \in M_P \\
 & \Longrightarrow & P \vdash_{\supset} G \quad \text{choosing } s = \emptyset
\end{array}$$

As a corollary, we have that $P \models_I G \;\Rightarrow\; P \vdash_{\supset} G$ even if we take $\models_I$ to stand for entailment in intuitionistic structures with constant domain. Note, in fact, that the argument used in the previous proof applies just as well to this latter case being M_P defined over substates with constant domain.

This justifies our initial claim that the restriction to the class of intuitionistic structures (and similarly of Kripke interpretations) with constant domain does not involve any loss of generality.

As already anticipated a construction similar to the one we have just presented is used by McCarty *et al.* in [7] to prove a completeness result for a language which allows universal quantification over embedded implications. The extended use of universal quantifiers enforces them to have a more general definition of canonical model where they assume that the domain of each substate is constant but defined over a signature containing infinitely many constant symbols. In our case we don't need this generality thanks to the restriction we impose on the use of universal quantifiers. This point is discussed in more detail in the next section where we consider intuitionistic semantics of Hereditary Harrop formulas.

4.2.2. Universal Quantification. As mentioned above, the syntactic restriction on the use of universal quantifiers allowed us to state the soundness and completeness results by assuming the simplified framework of intuitionistic interpretations with constant domain. The canonical model used in theorem 4.3 made specific appeal to this assumption.

This is no longer possible if we assume the use of universal quantifiers allowed by the definition of Hereditary Harrop formulas. Consider the proof predicate $\vdash_{\forall}$ defined in section 3.7. The completeness result of theorem 4.3 would now be stated as follows:

If $P \models_I G$ then $\Sigma; P \vdash_{\forall} G$ for any program P and goal G over the signature Σ

It is easy to see that the canonical model of theorem 4.3 is useless here. Let P be the program $\{p(a) \wedge p(b)\}$ and G be the goal $\forall x\ p(x)$. Then clearly, for any $\Sigma \supseteq \{a, b\}$, $\Sigma; P \not\vdash_{\forall} \forall x\ p(x)$. In fact, by an application of (GENERIC$_{\forall}$) we obtain:

$$\Sigma; P \vdash_{\forall} \forall x\ p(x) \quad \text{only if} \quad \Sigma + \{c\}; P \vdash_{\forall} p(c)$$

and there is no way we can reduce this sequent any further. Conversely, $M_P \models_I \forall x\ p(x)$ since for all its substates $Dom(s) = \{a, b\}$ and $\phi(s) \supseteq \{p(a), p(b)\}$. As a matter of fact, any intuitionistic model of P whose states have all the Herbrand universe of P as the associated domain will satisfy $\forall x\ p(x)$ and thus the completeness proof will not go through under this assumption. Indeed, this shouldn't sound surprising: $\forall x\ p(x)$ is certainly not (either classically or intuitionistically) entailed by the conjunctive formula $\{p(a) \wedge p(b)\}$ and thus the fact that $\forall x\ p(x)$ cannot be proved is just what we should expect.

The point is that we need to consider a wider class of intuitionistic structures. One possible solution is to allow infinitely many constants in the domain of each substate. Under this assumption we can still have a completeness proof following the same construction proposed by Bonner *et al.* in [7]. A more general approach consists of considering structures with non-constant domain as it is done by Miller in [72] and by McCarty in [64].

Miller's construction in [72] is based on a canonical model defined along the same guidelines as that used in theorem 4.3. The difference is that the substates of his model are defined as a pairs of the form $\langle \Sigma, s\rangle$. The *domain* function *Dom* applied to substate $\langle \Sigma, s\rangle$ returns the set of all the Σ-terms and the growth of the universe is captured by means of a refined definition of the ordering relation $\leq$ over the substates. Namely, $\langle \Sigma, s\rangle \leq \langle \Sigma', s'\rangle$ iff $\Sigma \subseteq \Sigma'$ and $s \subseteq s'$.

In [64], McCarty takes a different approach and defines a fixed point computation of the intuitionistic models he associates with his programs. The approach is in some respects similar to the fixed point construction proposed by Miller in [71] with a few interesting differences. Instead of considering the set of all the possible programs as substates, he starts with an initial substate s_0, which is essentially a set of facts (atomic formulas) and then works, for any program, with all the substates $s' \geq s_0$ which satisfy the formulas in that program. The result of his fixed point construction is again an intuitionistic model for the program and, within that model, he is also able to identify one particular substate, the *minimal one*, which satisfies all (and only) the goals which can be proved from the program.

4.3. S4-Modal Logic: Foundations for Embedded Implications

We conclude our analysis by discussing the modal reconstruction of embedded implications proposed by Giordano and Martelli in [41]. In the following, we assume that the reader is familiar with the notions of modal logic. Here we will only give an intuitive account of the basics in the attempt to keep the discussion self-contained. The interested reader will find in [25] and [51] a full description of the underlying theory.

Modalities and Modal Logics. The extension of a logical system to accommodate the modal operators is meant to capture notions of truth and falsity richer than the classical ones. Among true propositions we allow ourselves to "distinguish between those which merely *happen* to be true and those which are *bound* to be true [51]".

Similarly, between propositions which are false, we distinguish between those which are simply false and those which are *necessarily* false. The concepts of necessary truth and falsity have a very elegant and intuitive interpretation in terms of Kripke's possible-world semantics. If we think of a proposition as stating a property about a given world (the elements of the domain of that world) then we can interpret a necessarily true proposition as one that "could not fail to be true no matter how the present situation evolved", i.e. a proposition which is true in every possible world accessible from the present one. In contrast, a true proposition is one which is true in the present world but which could turn out to be false in (at least one of) the situations accessible from the present one.

Given this intuition, it is of course reasonable to expect that there exist formal ways of forming propositions which are necessarily true, and for distinguishing them from simply true propositions. Similarly, there should be formal methods for inferring necessarily true and true consequences from a given set of hypotheses. Modal logic provides an adequate ground for this formalization. Of course, even richer notions of truth can be considered and correspondingly different modal languages can be used to formalize them. The language we consider here is the S4-modal system with a single modal operator of *necessity*. More precisely, we will consider a subset of this logic whose language is the same as that we have considered so far with the addition of the modal operator $\Box$. For any D- or G-formula F, the intended interpretation of the corresponding modal formula $\Box F$ will be "it is necessary that F".

The definition of the S4-modal calculus is simply obtained by extending the axiomatization of classical first-order logic with the following three axioms:

$$
\begin{array}{ll}
(1) & \Box\alpha \supset \alpha \\
(2) & \Box(\alpha \supset \beta) \supset \Box\alpha \supset \Box\beta \\
(3) & \Box\alpha \supset \Box\Box\alpha
\end{array}
$$

and by adding a third inference rule, the *rule of necessitation* $\dfrac{\vdash_{S4}\ \alpha}{\vdash_{S4}\ \Box\alpha}$

A few remarks will help clarify the intuitive reading of the axioms and the rule of necessitation. The latter formalizes the (quite reasonable) intuition that "any proposition which has the form of a valid formula is not merely true but rather necessarily true [51]". As for the axioms, axiom (1) should be clear since it states that whatever is necessarily true is also true. Axiom (2) is simply a more convenient representation of the formula $(\Box\alpha \wedge \Box(\alpha \supset \beta)) \supset \Box\beta$ which formalizes another intuitive statement: "whatever (β in the above formula) logically follows from a necessary truth (α) is itself a necessary truth". Finally, axiom (3) formalizes the assumption, which is peculiar of S4-modal logic, that "whatever is necessary is also necessarily necessary". Interpreted in a possible-world semantics, this simply means that not only will a formula $\Box\alpha$ be true in all the possible situations accessible from the present one, but it will be necessarily true in all such situations.

S4-modal Satisfaction and Validity. These intuitive arguments can be formalized in the following definition of S4-modal satisfaction. As it is done in [41], in the remainder of this section we will restrict to study the semantics of the propositional subsets of the languages we have considered. Accordingly, we will consider a propositional modal language defined over the connectives $\wedge$, $\supset$ and the modal operator $\Box$.

Within this setting, an S4-Kripke interpretation can be defined as a triple $K = \langle W, \leq, \phi \rangle$ where W is the set of worlds, $\leq$ is a reflexive and transitive relation over W and the *valuation* function ϕ is defined over W and ranges over the power-set of the predicate symbols of the language. The truth of a formula α in an S4-Kripke interpretation K at world w is formalized by the following definition [41].

$$\begin{array}{llll} w, K \models_{S4} \alpha & \text{iff} & \alpha \in \phi(w) & (\alpha \text{ atomic}) \\ w, K \models_{S4} \alpha \wedge \beta & \text{iff} & w, K \models_{S4} \alpha \ \text{ and } \ w, K \models_{S4} \beta & \\ w, K \models_{S4} \alpha \supset \beta & \text{iff} & w, K \models_{S4} \alpha \ \Rightarrow \ w, K \models_{S4} \beta & \\ w, K \models_{S4} \Box\alpha & \text{iff} & w', K \models_{S4} \alpha & \text{for all } w' \geq w \end{array}$$

A formula α is said to be true in K iff $w, K \models_{S4} \alpha$ for all the worlds of K; α is S4-valid iff it is true in every S4-Kripke interpretation.

4.3.1. Embedded Implications and Dynamic Scope. We first consider the propositional case of the language proposed by Miller. The interpretation of this language within S4-modal logic is based on the well-known mapping between intuitionistic logic and S4-modal logic [33]. Applied to the set of D- and G-formulas, this mapping can be defined as follows:

$$\begin{array}{rcl} \top^* & = & \top \\ A^* & = & \Box A \\ (\alpha \wedge \beta)^* & = & \alpha^* \wedge \beta^* \\ (\alpha \supset \beta)^* & = & \Box(\alpha^* \supset \beta^*) \end{array}$$

The corresponding language of D- and G-formulas is defined by the following productions:

$$\begin{array}{rcl} G^* & ::= & \top \mid \Box A \mid G^* \wedge G^* \mid \Box(D^* \supset G^*) \\ D^* & ::= & \Box A \mid D^* \wedge D^* \mid \Box(G^* \supset \Box A) \end{array}$$

In view of the definitions of S4 and intuitionistic satisfiability, it should be clear that this mapping provides a semantic preserving transformation. Given any S4-Kripke interpretation K and any propositional formula α, it is immediate to verify that $w, K \models_I \alpha$ iff $w, K \models_{S4} \alpha^*$. The following result is, in fact, an instance of the equivalence proved by Fitting in [33].

Theorem 4.4. Let P be a set of propositional D-formulas and G be a propositional goal formula. Then: $P \models_I G$ iff $P^* \models_{S4} G^*$

In [41], Giordano and Martelli present an interesting reconstruction of Miller's proof-predicate $\vdash_{\supset}$. Their idea is to look at the modal formula $\Box(D^* \supset G^*)$ as specifying a transition to a new world where $D^* \supset G^*$ is true and where we can attempt a proof for G^* by adding D^* to the set of available formulas. Now assume that $\Box(D^* \supset G^*)$ occurs as a goal in one of the D-formulas of P^*. All such formulas are of the form $\Box(G^* \supset \Box A)$ and are thus necessarily true in all the worlds accessible from the current one. But then, in any new world reached in the attempt to prove $D^* \supset G^*$, both the formulas in D^* and P^* will be at our disposal.

More importantly, the formulas in D^* will be available to construct a proof for any of the goals occurring in the D-formulas of P^* (and vice-versa). This is how the dynamic scope rule encompassed by (AUGMENT) is captured in the modal framework.

The same argument suggests how the static scope mechanism embedded in the proof predicate $\models_{\overline{\text{stk}}}$ should be modeled.

4.3.2. Embedded Implications and Lexical Scope. The idea is that, when moving to a new world to construct a proof for $\Box(D \supset G)$, we should have at our disposal not the D-formulas of P^* but rather only the atomic formulas which can be derived from them. But this implies that the implication symbol we use for clauses and embedded implications should be given different characterizations in the modal language. This observation led Giordano and Martelli to formulate their modal reconstruction of the proof predicate $\models_{\overline{\text{stk}}}$ in terms of a new mapping that assumes the co-existence of two interpretations of the implication connective: classical and intuitionistic implication.

For the purpose of illustrating this point, we will find it convenient to adopt the syntax of [41] to explicitly distinguish between classical implication (denoted with $\Rightarrow$) and intuitionistic implication ($\supset$). The resulting language is defined by the following productions.

$$\begin{array}{lcl} G & ::= & \top \mid A \mid G \wedge G \mid \exists x G \mid D \supset G \\ D & ::= & A \mid D \wedge D \mid \forall x D \mid G \Rightarrow A \end{array}$$

The mapping from this language to S4-modal logic behaves as the the previous one on atomic, conjunctive and disjunctive formulas and it applies two distinct transformations for implicative formulas. Namely:

$$\begin{array}{rcl} (\alpha \supset \beta)^* & = & \Box(\alpha^* \supset \beta^*) \\ (\alpha \Rightarrow \beta)^* & = & \alpha^* \supset \beta^* \end{array}$$

When applied to the D and G-formulas introduced above, this transformation produces the modal language defined by the following productions:

$$\begin{array}{lcl} G^* & ::= & \top \mid \Box A \mid G^* \wedge G^* \mid G^* \vee G^* \mid \Box(D^* \supset G^*) \\ D^* & ::= & \Box A \mid D^* \wedge D^* \mid G^* \supset \Box A \end{array}$$

Note that $G \Rightarrow A$ is not interpreted as the necessarily true implication $\Box(G^* \supset \Box A)$ but simply as $G^* \supset \Box A$. Hence, when attempting a proof for the G-formula $\Box(D^* \supset G^*)$ in a program P^*, the set of formulas available for backchaining on the goals occurring in D^* will not be the clauses of P^*, but rather their *atomic consequences* which are in fact necessarily true statements of the form $\Box A$. This is precisely the meaning of the abstract definition of the (AUGMENT) rule we introduced in section 3.3 to account for the interpretation of $\supset$ as a lexical scope rule.

The following example, which we borrow form [41], provides a nice and quite intuitive picture.

Example 4.1. Consider again program $P = \{a \Rightarrow b\}$ and goal $G = a \supset b$ of example 3.2. We have shown there that $P \not\models_{\overline{\text{stk}}} G$. Now we show that the same behaviour is obtained by interpreting P and G in S4-modal logic. Consider in fact the transformed formulas $P^* = \{\Box a \supset \Box b\}$ and $G^* = \Box(\Box a \supset \Box b)$. It is easy to see that $P^* \not\models_{S4} G^*$. Consider the following counter-model. Take $K = \langle \{w_1, w_2\}, \leq, \phi \rangle$ with $w_1 \leq w_2$, $\phi(w_1) = \emptyset$ and $\phi(w_2) = \{a\}$. Then $w_1, K \models_{S4} \Box a \supset \Box b$, but $w_1, K \not\models_{S4} \Box(\Box a \supset \Box b)$

The soundness and completeness result for the proof-predicate $\vdash_{\mathrm{stk}}$ with respect to entailment in S4-modal logic is stated and proved in [41].

Theorem 4.5. [41] *For any propositional program P and propositional goal G, $P \vdash_{\mathrm{stk}} G$ iff $P^* \models_{S4} G^*$.*

The proof is derived by defining several intermediate semantics which are shown to be equivalent to the operational definition of $\vdash_{\mathrm{stk}}$. The first step is represented by the fixed point semantics described in section 3.3; this is proved equivalent to an ad-hoc Kripke semantics where the two connectives $\Rightarrow$ and $\supset$ are interpreted respectively as classical and intuitionistic implication. Finally, in [41] this Kripke semantics is shown to coincide, through the mapping (*), with the S4-Kripke semantics we have outlined in this section.

4.3.3. Embedded Implications and Closed Scope We conclude by noting that the modal framework used so far can also be employed to give a formal semantics for the closed scope mechanism described in section 3.4. The idea has been again proposed in [41], and consists of interpreting a goal $D \supset G$ as specifying a transition to a world where none of the currently available formulas denote true propositions. The following definition of the mapping meets this requirement.

$$\begin{array}{lcl} G^* & ::= & \top \mid A \mid G^* \wedge G^* \mid G^* \vee G^* \mid \Box(D^* \supset G^*) \\ D^* & ::= & A \mid D^* \wedge D^* \mid G^* \supset A \end{array}$$

Now $D \supset G$ specifies a change of context from P to a new world where no formulas other than those introduced by D are valid.

In a recent paper [6], Badaloni *et al.* present an extension of the modal framework we have outlined in this section and propose a modal language in which richer forms of module composition and powerful scope rules can be elegantly integrated. The language is defined as a clausal fragment of a multimodal logic for which they present a Kripke semantics and a sequent calculus.

5. IMPLEMENTATION

In the last part of this survey we focus on the implementation of the algebraic operators and of the scoping mechanisms outlined in the previous sections.

At the implementation level, the issues that arise in the two cases are closely related. The different forms of algebraic composition are realized in terms of corresponding binding policies for the local and non-local references to a predicate. The same principles apply to the modular languages based on the use of embedded implications. In this latter case, however, the design is complicated by the dynamic evolution of the structure of a program. Thus, the implementation will have to support the dynamic update of the binding for a predicate as well as provide adequate data structures for the run-time representation of a program. In this respect, the architectural design for the modular languages described in section 3 generalizes the case of the composition operators of section 2.

A natural way to think of the run-time representation of a program is in terms of the list of its component clauses. Whenever an embedded implication $D \supset G$ is encountered, the clauses in D are added to the list and as soon as G is deterministically solved or finitely fails, they are discarded. This run-time representation can serve different purposes depending on the specific approach adopted for the implementation. We have in fact two possibilities: we can either add an extra layer on top of Prolog and have this layer handle the list of clauses explicitly, or hide the representation and let the underlying engine manipulate it.

Meta-Interpretative and Transformational Implementations. The first idea has been exploited in the literature in two different ways. In [17] Brogi and Turini show that their algebraic operators can be implemented using a meta-interpreter which takes the list of modules as an extra argument. A similar technique is used by Gabbay and Reyle [37] for developing a prototypical implementation of N-Prolog. Although elegant and well-suited for quick prototyping, this solution is by far too inefficient to be used for real applications.

A more refined idea, to which we alluded earlier in the paper (cf. section 3.1) consists of defining a transformation mapping from modular programs into ordinary logic programs. For instance, in [53] the authors implement a logic programming module system with a preprocessor that maps modular logic programs into flat Prolog code. This approach may turn out to be quite effective depending on the scoping constructs we are to implement. For example, the operator described by Moscowitz and Shapiro in [82] to transform lexical logical programs into corresponding logic programs produces quite efficient programs as it only relies on a renaming scheme for predicate names and variables. However, for more dynamic composition mechanisms, as those adopted by Miller or in Contextual Logic Programming, the transformation requires that new arguments be added to the predicates of the transformed program to represent the list of the clauses currently in use. This introduces a considerable overhead due to the unification of the extra-arguments. The interested reader is referred to the work of Denti *et al.* [29] for a fuller discussion on this point.

Compilative Implementations. Most of the proposals found in the literature (see for example [5, 58, 30, 52, 23]) rely on a run-time program representation defined in terms of internal data structures manipulated by the underlying engine. They propose different extensions of the standard Prolog abstract machine, the Warren's Abstract Machine introduced by D.H.D. Warren in [92], with new instructions and data structures needed to support the modular languages under consideration. Throughout this section, we will concentrate on this approach as it provides a good framework for an efficient treatment of all the compositional and scoping constructs we have considered in this survey. We will review the existing proposals and discuss some optimizations. We assume familiarity the workings of the WAM; the interested reader will find in [1] and in [92] a comprehensive and detailed description.

The fundamental issues that must be dealt with in an implementation of the dynamic aspects of modules are:

- The treatment of embedded implications: this conceptually amounts to "asserting" and "retracting" program clauses. Since embedded implications can be nested arbitrarily, several nested applications of these operations may have to be performed at run time. However, as stated by Nadathur *et al.* [84], "the assertion and retraction of program clauses follows a stacking discipline, and may as such be implemented using a run-time stack". Of course, backtracking will need special treatment as it may require the reinstatement of a program "asserted" at earlier stages of the computation.

- The treatment of existentially quantified embedded implications: the evaluation of one such goal may enforce the assertion of program clauses containing variables that are dynamically instantiated.

- The treatment of universally quantified goals. As noted in section 3.7, this may require the introduction of "fresh" constants to be substituted for the universally quantified variables occurring in the goal. Furthermore, universal and existential quantifiers might appear in any order in a goal (in particular the existential quantification may surround the universal one). In this case, we must also guarantee that the existentially quantified variables be not instantiated to the constants introduced in place of the universally quantified variables that occur within the scope of the existential quantifier. This requires an appropriate treatment of unification and the introduction of tagged variables [83].

For the sake of clarity, these issues will be dealt with separately. There is in fact little overlap among the techniques needed to handle each of them and thus their integration can be accomplished smoothly.

5.1. Embedded Implications

We first briefly review the systems described in section 3 to point out some important remarks about the terminology used there. We have used the term *lexical* to refer to the scope rules adopted in Contextual Logic Programming as well as in the language proposed by Giordano *et al.* in [42]. As a matter of fact, the term is consistent with the use of embedded implications presented in [42] but not with the use of extension goals suggested by Monteiro and Porto in [78]. The problem is that in Contextual Logic Programming, modules have *names* and the same name can occur in several extension goals. Thus, the same module may have different surrounding scopes and the binding for the references to non-local predicates in that module depends on the different possible scopes. In other words, there is no way that we can determine *lexically* which definition to associate with a reference to a non-local predicate in a module. This is possible if we assume, as Giordano *et al.* in [43], that the clauses (and *not* their name) be nested by means of an embedded implication. In fact, under this assumption, an embedded implication can be compiled away by means of a renaming schema similar to that used by Moscowitz and Shapiro in [82].

In the following discussion, we will assume that modules are designated by names and that the same name may have multiple occurrences in the same program. We will, accordingly, use the (more appropriate) term *quasi-static* to refer to the *lexical* scope rules introduced in section 3 for Contextual Logic Programming.

There are (at least) two properties which should be satisfied by a realistic implementation of a modular system.

- Separate compilation of each module. This a fundamental requirement for programming in the large; separate segments of compiled code should be produced for each module and dynamically linked in the run-time representation of the program.

- Code sharing in module representation. If we can use a module in different contexts or have multiple occurrences of the same module in the same context we would like to maintain one single copy of the module code to be used in any context. This naturally leads to a dynamic representation of modules corresponding to the *closures* used in functional languages.

Modules as Closures. Modules will be represented in the WAM as closures. A closure for a module M consists of a set of bindings for the module's parameters and local variables, plus a set of bindings for the predicate calls occurring in M.

The stack discipline underlying the workings of embedded implications can be implemented using a run-time stack to hold the closures of the modules that occur in the embedded implications. When an implication goal $M \supset G$ is encountered, the closure of M gets recorded on the context stack and the access to the code is made relative to it. Upon completing the evaluation of the embedded implication, the closure of M is discarded and becomes inaccessible to subsequent goals.

In the architecture proposed by Lamma *et al.* in [57, 58], closures are implemented as new data structures called *instance environments*. A corresponding structure, called *implication point*, is used by Jayaraman, Nadathur *et al.* in [52, 84, 56] for the same purpose. The difference between these two approaches is that in the former closures are allocated in a new stack (the *instance environment stack*), whereas in the latter they are held in the local stack of the WAM.

To outline the basic features of the implementation of embedded implications, in the following we will assume that the run-time support holds the closures in a separate stack. We will refer to it as the *context stack* and let `pc` denote its top element. The new instructions `allocate_ctx` and `deallocate_ctx` introduced in [57, 58] will be used for manipulating the context stack (two corresponding instructions, `push_impl_point` and `pop_impl_point` are used in [84, 56]).

As an example, consider the goal $(M_1 \supset G_1) \wedge G_2$. The compiled code for this goal is opened by an `allocate_ctx` instruction for M_1, it is followed by the code for G_1 (which is thus evaluated in a context stack containing the closure of M_1), then by a `deallocate_ctx` which deallocates the closure of M_1 and restores the previous context before proceeding to the code for G_2.

The interaction of backtracking with this compilation schema has to be considered carefully. If, in the previous example, G_2 fails, then the closure of M_1 needs to be restored in the context stack before considering alternative choices for G_1. The implementation must provide methods for realizing this behaviour in an efficient manner. In the case of the implication $M \supset G$, this is obtained in the existing implementations by performing the physical deallocation of the closure of M only when G is deterministically solved, and by performing the necessary bookkeeping to backtrack correctly. In particular, an extra-field is added to each choice point in order to record the value of the `pc` register at the time the choice point is created. This solution is adopted both in [57, 58] and in [56].

Predicate Bindings. In the WAM the address used by the `call` and the `execute` instructions is determined statically. In the implementation of modular languages, instead, this address must be determined dynamically. Determining the bindings between predicate calls and definitions represents the major source of run-time overhead for these systems as the cost of binding resolution amounts to a look-up access in the context stack to determine the correct address for the call.

In [21] and [22], we showed how the overhead due to the look-up can be substantially reduced by means of a source-to-source transformation based on partial evaluation. The interested reader in referred to [21] and [22] for more details on this issue. Here, we concentrate on the compilative approaches proposed in the literature. We will distinguish two modes for embedded implications. The non-monotonic extension of the program, peculiar of Contextual Logic Programming and its variation proposed by Mello *et al.* in [67] will be referred to as the *overriding* mode. The term *extension* will be instead used to qualify the scope mechanisms used by Miller in [71] and by Giordano *et al.* in [42].

5.1.1. Overriding Mode for Embedded Implications

Dynamic Scope Rules. In the architecture proposed in [57, 58], for each module the compiler produces a table (the module's p-table) which associates the names of the predicates defined in that module with the address of their local definition. When a closure is allocated on the context stack for that module, it is made to point to the module's p-table.

At run time, the binding for a predicate call is computed by inspecting the p-tables referenced by the closures that occur in the context stack, starting from the closure pointed by `pc`. The address of the call is retrieved from the first p-table that contains an entry for the predicate being called. Although the access to the tables can be optimized using a hashing schema, the search along the context stack is linear in the length of the stack and it is performed at each call. Thus the overhead can be high.

An improvement to this schema is proposed in [52, 56]. In this case the access to the code is performed via a hash function which represents the set of all the bindings between predicate calls and definitions in use at the current computation stage. Given a predicate name, the hash function returns the appropriate entry point in the code area if a definition exists and an indication of failure otherwise.

This way, the time spent in the search of a predicate definition is considerably reduced. However, the hash function must be updated each time an implication goal is encountered and previous hash functions have to be restored when an implication goal is successfully solved or upon backtracking. Thus the major overhead of this solution is in the creation of a new access function each time an implication goal is encountered.

A similar approach is adopted in the implementation of *CSM* (*Contexts as SICStus Modules* [29]) which adds contexts to SICStus Prolog. As in [52, 56], hash functions indexed on predicate names are used to represent contexts. One hash function at time is in use, and previously computed functions are recorded in order to be restored as soon as the corresponding context is either restored (upon backtracking or success) or re-built.

More Static Scope Rules. The *quasi-static* scope rule of Contextual Logic Programming leaves room for more efficient and specialized implementations. In this case, even though the binding for a call depends on the structure of the context stack, the associated definition can be determined as soon as the context stack gets updated with a new closure. This is possible since the reference to a predicate in a context stack is not affected by any further extension of that stack. This is different from case of the dynamic scope rule since in that case the reference to a predicate can be resolved on at the time of the call.

Thus, the idea presented in [57, 58] is to compute the bindings for non-local calls occurring in a module only once, at the time the context stack gets extended with the closure of that module or, more precisely, at the time the call is firstly performed. The bindings are recorded as extra entries in the module's closure so that they can be accessed for future calls, indirectly, without repeating the search. The access to these entries in the closure can be performed by offset as their position in the closure can be fixed ahead by the compiler.

Inheritance When modules are statically configured into *isa*-hierarchies as suggested in section 3.6, the cost of binding resolution can be made constant independently of the scope rules that are adopted. If we assume a static scope rule, the binding between each call and the associated definition can be computed at compile time. The case of dynamic scope can be handled almost as efficiently as suggested by Bugliesi and Nardiello in [23]. In fact, in this case the evolution of the context stack is subject only to the evaluation of a message-sent. At each stage of the computation, the context stack contains the closures of the modules in the *isa*-hierarchy associated with the receiver of the last message-sent. Since the *isa*-hierarchies are associated statically with each module, the context stack can be represented with a single register, `self`, which points to the (closure of) the module which is the receiver of the last message-sent. The trick to handle a dynamic reference to a non-local predicate is the following. The modules' p-tables produced during compilation are set up so as to ensure the alignment of the p-table entries for the predicates in all the modules belonging to the same branch of the *isa*-hierarchy. At run time, the p-tables can then be accessed by offset, to realize an indirect call mechanism in much the same way as virtual tables are used in C++ to evaluate a virtual function.

5.1.2. Extension Mode for Embedded Implications. The extension mode for embedded implications has some effects on the implementation. In fact, when evaluating a call we may need to select clauses from more than one of the modules currently in use. Consequently, the clauses for the same predicate occurring in different modules should be conceptually chained.

For a dynamic scope rule, the chaining of clauses belonging to different modules in the same context cannot be performed at compile time. For the same reason, we cannot establish whether a predicate is deterministic or not. Thus, both in [57, 58] and [56, 84], the compiled code for a definition is always preceded by a `try_me_else` instruction even when the definition is deterministic. Similarly, the last instruction of the definition should guarantee that other clauses that may become available will be tried. Hence, the code for the last clause of the definition p is preceded by the `retry_me_else` instruction and finally followed by a new instruction `trust_extends` P_i.

The reference P_i is left unsolved at compile-time and is solved only as soon as a new closure gets pushed on the context stack. The corresponding address for the (possible) new definition will be inserted in the i-th position of that closure. The `trust_extends` instruction is similar to the `trust` instruction of the WAM; the only difference is that the instruction counter `P` of the WAM and the new register `pc` are now set to the values stored in the i-th position of the current closure in order to perform inter-module backtracking. It must be noted, however, that this implementation of the extension mode makes the indexing scheme of the WAM much less effective.

The approach based on the p-tables discussed in section 5.1.1 can be substantially improved if we assume an extension mode for embedded implications with dynamic scope rules. In fact, in this case the run-time representation always correspond to the union of the clauses contained in the modules currently in use. If $M_1, \ldots, M_n$ are the modules of a given program P, we can represent the evolving list of the modules in use with an n-bit vector. If at a given stage, M_i is in use, the i's position of the bit vector is set to 1. With each predicate name defined in the program, we can then associate a table recording (as a bit vector) the modules where that name is defined, together with the corresponding addresses. With this representation, binding a predicate call for p requires simply a bit-wise AND between the bit vector representing the context stack and the bit vector associated with p. All the addresses occurring at positions set to 1 after the bit-wise AND are accessed for evaluating p. Thus, the cost of computing a binding is constant. Furthermore this solution performs the extension of the context stack in a very efficient way. When the context stack is extended with module, the corresponding position in the bit vector is set to 1. Obviously, the current bit vector needs saving in order to make it possible to restore after the execution of the implication goal has been completed or upon backtracking.

5.1.3. Controlling Redundancy The treatment of implication goals described so far may result in the same closure being added several times to the context stack. This has a potential drawback since it may cause useless look-ups to be performed and the same solution to be produced several times.

One interesting question is whether the number of copies of any module in a program context can be restricted to just one. This can be done only for the dynamic scope rules and when we assume an extension mode for the evaluation of an embedded implication. As a counter-example, consider the modules:

$$m1 \;=\; \{b\text{:-}c\} \qquad\qquad m2 \;=\; \left\{ \begin{array}{c} c \\ a\text{:-}b \end{array} \right\}$$

The implication goal $m2 \supset m1 \supset m2 \supset a$ succeeds since a definition for a exists in the context $[m2, m1, m2]$ (in module $m2$, in particular), one for b is found in $[m2, m1]$, and finally one for c in $[m2]$. However, $[m2, m1, m2]$ is not equivalent to $[m2, m1]$ if we assume a quasi-static scope rule. It is in fact easy to check that the evaluation of $m1 \supset m2 \supset a$ fails under this assumption. It follows that for a quasi-static scope rule, neither the order of modules in the program context can be changed, nor the number of copies of module $m2$ restricted to just one. Conversely, if we consider a dynamic scope rule, then the program context is interpreted as the union of the clauses of component modules. Hence, multiple occurrences of the same module in the same context can be safely avoided. Notice, to this regard, that the same set of successful derivations is obtained both in $[m2, m1, m2]$ and $[m2, m1]$ with m_1 and m_2 defined as before.

In [56] the problem of redundancy in the case of dynamic scoping is solved by slightly changing the treatment of implication goals. To solve a goal of the form $D \supset G$ (where D is a set of clauses), D is added to the program context only if not already available. When an implication goal is of the form $\exists x_1 \ldots \exists x_n\ D \supset G$, the addition of $D[x_1/c_1 \ldots x_n/c_n]$ is performed only if the program context does not already contain $D[x'_1/c_1 \ldots x'_n/c_n]$.

An elegant solution to the problem of redundancy can be obtained using the bit vector representation of the context stack discussed in section 5.1.2. The position in the bit vector corresponding to a module, will be set to 1 only once, independently of the number of repeated occurrences of the same module in the context stack. However, this implementation works well only for dynamic scope rules and turns out to be inadequate for more static scope mechanisms.

5.2. Parametric Modules

As pointed out in section 3, an implication goal surrounded by existential quantifiers may give rise to a module *parametrized* by variable bindings. In particular, consider solving the implication goal $\exists x(\{p(x), r(x, a)\} \supset g(x))$. Assuming that x is replaced by the logic variable X, one would have to solve the goal $m(X) \supset g(X)$ where $m(X)$ is the parametric module consisting of the two clauses $p(X)$ and $r(X, a)$. In this case, we use a general term rather than just a constant to refer to the module, encoding the module name (m) as the main functor and the parameters (X) as arguments.

Assuming that the embedded implication is dealt with as required, with reference to the example above we would have to solve $g(X)$ with respect to a program that contains the two clauses $p(X)$ and $r(X, a)$. As stated in [84], the main difference with respect to the standard case is that now the variable X occurring in these clauses cannot be instantiated in arbitrary fashion, but only in a way consistent with the instantiation of the same variable in the goal. Moreover, this variable is shared between these two clauses: hence, starting from a program containing the clause $g(a) \leftarrow p(b)$ the goal $m(X) \supset g(X)$ fails since, to succeed, it would require instantiating X simultaneously with a and b.

To implement this behaviour correctly, the implementation must distinguish between existential and universal variables that appear in a module's designator, and provide mechanisms for dealing with this new kind of existential variables (called *parametrized variables* in the following).

This problem is solved both by Nadathur *et al.* in [84] and Lamma *et al.* [58] by recording the bindings for parametrized variables in the modules' closures. In particular, in [58] the structure holding the instance environments is expanded with a number of cells used to allocate the parameters of a module. Accordingly, `allocate_ctx` is given one additional argument specifying the number of parameters (and therefore of the additional cells to be allocated in the instance environment) of the module involved in the implication goal. While the offsets for these cells in the instance environment are determined at compile-time, their actual values will be determined dynamically. A new set of argument-registers (called `A_G` registers) is added to the WAM register set. These registers are unified with the arguments of a parametrized module before the extension of the program context takes place. Accordingly, the `get`, `put` and `unify` instructions are modified for handling these new registers together with the cells of the current instance environment.

A similar effect is obtained in [84] where an initializing step is performed for the clauses containing parametrized variables through the `initialize Vn,i` instruction. This instruction is similar to the `get_variable` instruction of the WAM with the difference that the second argument is obtained by using the i-th variable of the associated closure.

In both the implementations, to perform backtracking correctly, the references to the parametrized variables that have been bound during unification – and that must be unbound on backtracking – are recorded in the trail area.

The use of parametric modules raises new problems related to the redundancy checks described in section 5.1.3. In particular, the technique proposed in [56] does not apply directly in this case. For example, assume that the current context contains module $m(X)$ and assume that are to extend that context with $m(Y)$. The question is whether we should unify X and Y when checking to see if $m(Y)$ is already present in the context. It is easy to think of cases where binding them is not the intended behaviour. On the other hand, if we don't unify them, then it appears that $(m(Y) \supset G), X = Y$ and $X = Y, (m(Y) \supset G)$ may differ significantly, at least operationally[11].

5.3. Universally Quantified Goals

As pointed out in section 3.7 (similarly, at the beginning of section 5) permitting universal quantifiers in the body of a clause may produce alternated sequences of universal and existential quantifiers.

The problems related to a correct treatment of unification in these situations have been studied by several authors in the literature. In [63] McCarty presents a tableau proof procedure for his Clausal Intuitionistic Logic and proves it sound and complete with respect to intuitionistic entailment. In [31] Elliot and Pfenning present the implementation of Hereditary Harrop formulas in Standard ML. In [73], Miller discusses the combination of unification and quantifiers in a general higher-order setting.

Here we refer to the solution adopted by Nadathur *et al.* in [84] for λProlog [85]. In [83], Nadathur presents a correctness proof for this approach.

The idea is to use numeric tags for the constants and the logical variables involved in the computation. The tag is used to index the universes of symbols created during the execution. All the constant symbols appearing in the program clauses and the original goal are tagged with value 1. Each time a universal quantifier is encountered during the evaluation of the goal, (whence a new constant is introduced), the universe index is incremented by 1, and the newly added constant is tagged with this index. When an existential quantifier is encountered, a logic variable X is introduced tagged with the current value of the universe index.

During the unification process, a logic variable X can be bound only to terms with constants tagged by a tag smaller or equal to that associated to X. In particular, when binding a variable X with tag i to a term T, prior to permitting this instantiation, a consistency check is performed in order to ensure that i is greater or equal to the tag of any constant in T. The check, in practice, amounts to an *occur-check* test; if it succeeds so does the unification of X with T.

---

[11] We gratefully acknowledge Dale Miller [74] for bringing this point to our attention.

A different scenario arises when a variable of T is tagged with a universe index greater than i. This happens, for example, when the goal $\exists x \forall y \exists z\ p(x, f(z))$ gets evaluated in the program $\forall a\ p(a, a)$. Using the tagging schema outlined above, evaluating this goal reduces to evaluating $p(X^1, f(Z^2))$ (where the indexes annotate logic variables) in the program $p(A^1, A^1)$. A situation like this should not prevent unification to succeed provided that a proper tag propagation is performed on the term $f(Z)$ to which X is bound. This is needed to ensure that subsequent goals referencing Z be treated correctly. Consider for instance the program above, a new goal $\exists x \forall y \exists z\ p(x, f(z)) \wedge p(y, z)$ and its tagged version $p(X^1, f(Z^2)) \wedge p(c^2, Z^2)$ (c is new constant). As before, X^1 is bound to $f(Z^2)$; the problem is that now the subsequent attempt to bind Z^2 to the constant c^2 will succeed in spite of the fact that this binding violates the restrictions on the permitted instantiation for X^1 (being X^1 bound to a term which contains c^2).

To avoid this problem, as soon a variable X with universe index i is bound to a term T, the tags of all the variables occurring in T with universe index greater than i are set to i. With reference to the example above, after the binding of X^1 to $f(Z^2)$, the tag for Z becomes 1 thus preventing the unification of this variable with the constant c^2 and leading to a failure for the goal considered.

At the WAM level, the implementation requires the addition of a new field in each cell for representing the tags, and an extension of the WAM instructions for unification (e.g., `get_value`, `unify_value`, etc.) for performing both the consistency check on tags and the tag propagation.

6. CONCLUSIONS

We conclude our discussion with few additional remarks and considerations.

We hope that this survey has conveyed an adequate view of all the valuable work that has been done in the area over the past years. The extension of logic programming with module constructs was long understood as one of the keys to make this programming paradigm appealing to a wider community and adequate for developing practical applications. The research in the field has contributed, we believe, to make these expectations realistic.

The modular extensions we have outlined in this paper uphold the validity of this belief. They provide evidence of how different programming features and methodologies can be imported in logic programming to model abstraction mechanisms comparable in power to those available in procedural and functional languages.

In this respect, the foundations for future work appear to be solid and well-established as the logic of the modular systems we have described is relatively well-understood. Of course there are still several open questions as we have pointed out and a tighter integration of the two approaches we have outlined would be desirable. A further problem that certainly deserves future investigation is the impact of the different extensions on logic languages richer than those we have considered here. The treatment of negation in these languages represents one interesting instance of this problem.

At the current state of the art, however, there is already enough room to start moving from theory to practice. The time has come, or so it seems, to concentrate also on the design of logic languages that incorporate these features in elegant and, more importantly, practical ways. This first step will be certainly needed before this technology can be put to test in the development of practical and real-sized applications.

Acknowledgements

This article has greatly benefited from joint work and many fruitful discussions with Annalisa Bossi, Antonio Brogi, Paolo Mancarella, Dale Miller, Luis Monteiro, Antonio Natali, Antonio Porto, Gianfranco Rossi, Cristina Ruggieri and Giovanni Sambin.

Special thanks are due to Antonio Brogi, Laura Giordano, Dale Miller, Alberto Martelli, Andrea Omicini, Franco Turini and the anonymous referees for their useful comments on the first version of this paper.

This work has been partially supported by “Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo” of C.N.R. under grants n. 91.00898.PF69 and n. 93.01627.PF69.

REFERENCES

1. H. Aït-Kaci. *Warren's Abstract Machine*. The MIT Press, 1991.
2. H. Aït-Kaci and R. Nasr. Login: a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:182–215, 1986.
3. J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.
4. K. R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.
5. H. Bacha. MetaProlog design and implementation. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1371–1387. The MIT Press, 1988.
6. M. Baldoni, Giordano, and A. Martelli. A Multimodal Logic to Define Modules in Logic Programming. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium ILPS'93*, pages 473–487. The MIT Press, 1993.
7. A. J. Bonner, L. T. McCarty, and K. Vadaparty. Expressing database queries with intuitionistic logic. In L. Lusk and R.A. Overbeek, editors, *Proc. 1989 North American Conf. on Logic Programming*, pages 831–850. The MIT Press, 1989.
8. A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential Logic Programs. In *Proc. 20th Annual ACM Symp. on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.
9. K.A. Bowen and R.A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.
10. A. Brogi. *Program Construction in Computational Logic*. PhD thesis, Department of Computer Science, University of Pisa, Tech. Report TD-2/93, March 1993.
11. A. Brogi, E. Lamma, and P. Mello. Inheritance and Hypothetical Reasoning in Logic Programming. In *Proceedings of 9th European Conference on Artificial Intelligence*, pages 105–110. Pitman, 1990.
12. A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1):1–21, 1992.

13. A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In A.Voronkov, editor, *Logic Programming, LNCS*, pages 102–113. Springer-Verlag, 1992.

14. A. Brogi, E. Lamma, and P. Mello. Composing Open Logic Programs. *Journal of Logic and Computation*, 4(4):417–439, August 1993.

15. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition Operators for Logic Theories. In J.W. Lloyd, editor, *Computational Logic, Symposium Proceedings*, pages 117–134. Springer-Verlag, 1990.

16. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Meta for Modularising Logic Programming. In A. Pettorossi, editor, *Proceedings of the Third Workshop on Meta-programming in Logic META92*, pages 24–37, 1992.

17. A. Brogi and F. Turini. Metalogic for Knowledge Representation. In J.A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the 2nd Int. Conf.*, pages 100–106. Morgan Kaufmann, 1990.

18. M. Bugliesi. Inheritance Systems in Logic Programming: Semantics and Implementation. Ms Thesis, Dept. of Computer Science, Purdue University, West-Lafayette IN, USA.

19. M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.

20. M. Bugliesi. On the Semantics of Inheritance in Logic Programming: Compositionality and Full Abstraction. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming, LNAI 660*, pages 205–215. Springer-Verlag, 1993.

21. M. Bugliesi, E. Lamma, and P. Mello. Partial evaluation for hierarchies of logic theories. In S. Debray and M. Hermenegildo, editors, *Proc. 1990 North American Conf. on Logic Programming*, pages 359–376. The MIT Press, 1990.

22. M. Bugliesi, E. Lamma, and P. Mello. Partial Deduction for Structured Logic Programming. *Journal of Logic Programming*, 16:89–122, 1993.

23. M. Bugliesi and G. Nardiello. Inheritance Systems in Logic Programming: the Language and its Implementation. Technical report, Post Conf. Workshop on Practical Implementations and Systems Experience in L.P. ICLP'93, Budapest, 1992.

24. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

25. B. F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.

26. W. Chen and D. S. Warren. C-Logic for Complex Objects. In *ACM SIGMOD Conference on Management of Data*, 1989.

27. W. Chen and D.H. Warren. Objects as intensions. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 404–419. The MIT Press, 1988.

28. J.S. Conery. Logical objects. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 420–434. The MIT Press, 1988.

29. E. Denti, E. Lamma, P. Mello, A. Natali, and A. Omicini. Techniques for Implementing Contexts in Logic Programming. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming, LNAI 660*, pages 339–359. Springer-Verlag, 1993.

30. A. M. Dias. An implementation of a contextual logic programming system. Technical Report UNL-DI 26/90, Departamentop de Informatica, Universidade Nova de Lisboa, Master Thesis, September 1990.

31. C. Elliott and F. Pfenning. A Semi-functional Implementation of a Higher-order lLgic Programming Language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. The MIT Press, 1991.

32. M. Fitting. Enumeration Operators and Modular Logic Programming. *Journal of Logic Programming*, 4:11–21, 1987.

33. M. C. Fitting. Intuitionistic Logic, Model Theory and Forcing. *Studies in Logic and the Foundations of Mathematics, North-Holland*, 1969.

34. K. Fukunaga and S. Hirose. An experience with a Prolog-based object-oriented language. In *Proceedings of OOPSLA-86*. ACM Press, Portland (Oregon), 1986.

35. K. Furukawa, A. Takeuchi, S. Kunifujii, H. Yasukawa, M. Ohki, and K. Ueda. Mandala: A logic based knowledge programming system. In *Proc. FGCS'84 Int. Conference*, pages 613–622, Tokyo (J), 1984.

36. D. M. Gabbay. N-Prolog: an Extension of prolog with Hypothetical Implication. II Logical Foundations and Negation as Failure. *Journal of Logic Programming*, 4:251–283, 1985.

37. D.M. Gabbay and N. Reyle. N-Prolog: an Extension of Prolog with Hypothetical Implications. I. *Journal of Logic Programming*, 4:319–355, 1984.

38. M. Gabbrielli, G. Levi, and M. C. Meo. Observationally Equivalences for Logic Programs. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 131–145. The MIT Press, 1992.

39. H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM Press, 1989.

40. H. Gallaire. Merging objects and logic programming: Relational semantics. In *AAAI-86 Conference Proceedings*, pages 754–758, 1986.

41. L. Giordano and A. Martelli. A modal reconstruction of blocks and modules in logic programming. In *Proc. 8th Int. Conference on Logic Programming*, pages 239–253. The MIT Press, 1991.

42. L. Giordano, A. Martelli, and G.F. Rossi. Local definitions with static scope rules in logic languages. In *Proc. FGCS'88 Int. Conference*, pages 389–396, 1988.

43. L. Giordano, A. Martelli, and G.F. Rossi. Extending Horn Clause Logic with Modules Constructs. *Theoretical Computer Science*, 95:43–74, 1992.

44. R. Harrop. Concerning Formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)(B(x))$ in Intuitionistic Formal Systems. *Journal of Symbolic Logic*, 25(1):27–32, 1960.

45. P. Hill. A parametrised module system for constructing typed logic programs. In *Proceedings of 13th International Joint Conference on Artificial Intelligence*, pages 874–880. Morgan Kauffman, August 1993.

46. P.M. Hill and J. W. Lloyd. *The Gödel Programming Language*. The MIT Press. Forthcoming.

47. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*. To appear.

48. J. Hodas and D. Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conference on Logic Programming*, pages 511–526. The MIT Press, 1990.

49. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In G. Kahn, editor, *Proceedings of Sixth Annual Symposium on Logic in Computer Science*, pages 32–42, 1991.

50. E. Horowitz. *Programming Languages*. Springer-Verlag, second edition, 1984.

51. G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen & Co. Ltd, 1968.

52. B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *Proc. 8th Int. Conference on Logic Programming*, pages 871–886. The MIT Press, 1991.

53. I. Karali, E. Pelecanos, and C. Halatsis. A versatile module system for Prolog mapped to flat Prolog. In *Proceedings ACM Symposium on Applied Computing SAC93*, 1993.

54. M. Kifer and G. Lausen. F-Logic: a Higher-Order Language for Reasoning about Objects, Inheritance and Schema. In *ACM SIGMOD Conference on Management of Data*, pages 134–146, 1989.

55. S. A. Kripke. Semantical Analysis of Intuitionistic Logic. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland Pub. Co., 1965.

56. K. Kwon, G. Nadathur, and D.S. Wilson. Implementing a Notion of Modules in the Logic Programming Language λProlog. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming, LNAI 660*, pages 359–393. Springer-Verlag, 1993.

57. E. Lamma, P. Mello, and A. Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 303–317. The MIT Press, 1989.

58. E. Lamma, P. Mello, and A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming*, 14:187–222, 1992.

59. J.L. Lassez and M.J. Maher. Closures and fairness in the semantics of logic programming. *Theoretical Computer Science*, 29:167–184, 1984.

60. Logicware Inc., Toronto, Canada. *M-Prolog Language Reference*, 1985.

61. P. Mancarella and D. Pedreschi. An algebra of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1006–1023. The MIT Press, 1988.

62. F.G. McCabe. *Logic and Objects*. Prentice Hall International, London, 1992.

63. L. T. McCarty. Clausal Intuitionistic Logic II: Tableau Proof Procedures. *Journal of Logic Programming*, 5(2):93–132, 198.

64. L. T. McCarty. Clausal Intuitionistic Logic I: Fixed point Semantics. *Journal of Logic Programming*, 5(1):1–31, 1988.

65. P. Mello. Inheritance as Combination of Horn Clause Theories. In D. Lenzerini, D. Nardi, and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation*, pages 275–289. J. Wiley and Sons, 1989.

66. P. Mello and A. Natali. Objects as Communicating Prolog Units. In J. Bezivin, J.M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of ECOOP 87*, number 276 in LNCS, pages 181–191. Springer-Verlag, 1987.

67. P. Mello, A. Natali, and C. Ruggieri. Logic programming in a software engineering perspective. In L. Lusk and R.A. Overbeek, editors, *Proc. 1989 North American Conf. on Logic Programming*, pages 451–458. The MIT Press, 1989.

68. A. Meyer. Semantical Paradigms: Notes for an Invited Lecture. In *Proceedings 3rd Annual Symposium on Logic in Computer Science*, pages 236–242. IEEE Computer Society, 1988.

69. D. Miller. A theory of modules in logic programming. In *Proceedings of Symposium on Logic Programming*, pages 106–114, 1986.

70. D. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 268–283. The MIT Press, 1989.

71. D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

72. D. Miller. Abstract Syntax and Logic Programming. In *Proc. 1st and 2nd Russian Conf. on Logic Programming*, pages 322–337. Lecture Notes in AI, number 52, Springer-Verlag, 1992.

73. D. Miller. Unification under a mixed prefix. *Journal of Symbolic Logic*, pages 321–358, 1992.

74. D. Miller. e-mail communication. July 1993.

75. D. Miller. Invited Lecture. In *JICSLP92 Workshop on Modules in Logic Programming Languages*, Washington D.C. 1992.

76. D. Miller, G. Nadathur, F. Pfenning, and A. Shedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

77. M.Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, 1988.

78. L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 284–302. The MIT Press, 1989.

79. L. Monteiro and A. Porto. A transformational view of inheritance in Logic Programming. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conference on Logic Programming*, pages 481–494. The MIT Press, 1990.

80. L. Monteiro and A. Porto. Syntactic and Semantic Inheritance in Logic Programming. In J. Darlington and R. Dietrich, editors, *Workshop on Declarative Programming*. Workshops in Computing, Springer-Verlag, 1991.

81. L. Monteiro and A. Porto. A Language for Contextual Logic Programming. In J.W. de Bakker K.R. Apt and J.J.M.M. Rutten, editors, *Logic Programming Languages, Constraints, Functions and Objects*, pages 115–147. The MIT Press, 1993.

82. Y. Moscowitz and E. Shapiro. Lexical logic programs. In K. Furukawa, editor, *Proc. 8th Int. Conference on Logic Programming*, pages 349–363. The MIT Press, 1991.

83. G. Nadathur. A Proof Procedure for the Logic of Hereditary Harrop Formulas. *Journal of Automated Reasoning*, pages 115–145, August 1993.

84. G. Nadathur, B. Jayaraman, and K. Kwon. Scoping Constructs in Logic Programming: Implementation Problems and Their Solution. Technical Report CS-1993-17, Department of Computer Science, Duke University, July 1993.

85. G. Nadathur and D. Miller. An Overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 810–827. The MIT Press, 1988.

86. R. O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160. IEEE Computer Society Press, 1985.

87. Quintus Computer Systems, Inc., Mountain View, California. *Quintus Prolog User's Guide*, 1986.

88. U. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. In *Proc. Lisp and Functional Programming*, pages 289–297. ACM, 1988.

89. D.T. Sannella and L.A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147–177, 1992.

90. Swedish Institute of Computer Science, Kista, S. *SICStus Prolog User's Guide*, 1990.

91. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

92. D.H.D. Warren. An abstract Prolog instruction set. Technical Report TR 309, SRI International, 1983.