Knowledge Representation Using High-Level Non-monotonic Reasoning

Mauricio Osorio[1], Juan Carlos Nieves[1], Fernando Zacarias[2], and Erika Saucedo[3]

[1] Universidad de las Americas,
Departamento de Ingenieria en Sistemas Computacionales,
Sta. Catarina Martir, Cholula
72820 Puebla, Mexico
josorio@mail.udlap.mx
[2] Benemerita Universidad Autonoma de Puebla,
Facultad de ciencias de la computación,
75570 Puebla, Mexico.
[3] Instituto Tecnológico y de Estudios Superiores de Monterrey, CEM,
Departamento de Ingenieria y Ciencias
Atizapan de Zaragoza, Edo. de Mexico.
esaucedo@campus.cem.itesm.mx

Abstract. We introduce the new paradigm of High-Level Non-Monotonic reasoning (HLNM). This paradigm is the consolidation of our recent results on disjunctions, sets, explicit and implicit negation, and partial-order clauses. We show how these concepts are integrated in a natural way into the standard logic programming framework. For this purpose, we present several well known examples from the literature that motivate the need of this new paradigm. Finally, we define a declarative semantics for HLNM reasoning.

1 Introduction

Knowledge based systems can be modeled adequately using non-monotonic logic, because it provides formal methods that enable intelligent systems to operate properly when they are faced with incomplete or changing information. There exist close interconnections between logic programming and non-monotonic reasoning. Actually, negation as failure, the interpretation of negation in logic programs as the failure to prove a ground atomic formula is a prothotypical example of non-monotonic behavior.

In this paper, we introduce the new paradigm High-Level Non-Monotonic reasoning, which supports the well-known formalism of definite logic programs by allowing for disjunctions in rule heads, general formulas in rule bodies, explicit and implicit negation, set notation $\{H|T\}$ like Prolog's list notation, and function atoms $f(t) = e$ or $f(t) \geq e$ for formulating conditions on functions like $intersect(\{X|\_\}, \{X|\_\}) \geq \{X\}$. Thus, High-Level Non-Monotonic reasoning integrates functional programming aspects, since functions may be defined by

O. Cairo, L.E. Sucar, and F.J. Cantu (Eds.): MICAI 2000, LNAI 1793, pp. 13–24, 2000.

logical rules. One main contribution of our paper is the definition of a semantics integrating all the above mentioned concepts.

The interest in allowing two kinds of negations (default negation and explicit negation) is considered elsewhere in the literature, see [4,3,7,18]. It has been shown in [3] how they can be used to represent defeasible reasoning. In [28] we discuss an application to the Botanic filed. Based on the work in [18] we also claim that HLNM reasoning is better suited for legal knowledge representation than normal logic programming, due to the presence, in HLNM, of both explicit and default negation.

With respect to default negation, it is well known that the stable semantics and the well founded semantics (WFS) are the two most acceptable definitions of this form of negation in Logic Programming. Our use of default negation is different than in previous approaches. We allow the combination of the idea of negation as failure with reasoning by cases to obtain a powerful use of negation. Preliminary research on this direction is presented in [16,10].

Disjunctive logic programming is one of the most expressive declarative logic programming language [8]. It is more natural to use since it permits direct translation of disjunctive statements from natural language and from informal specifications. The additional expressive power of disjunctive logic programs significantly simplifies the problem of translation of non-monotonic formalisms into logic programs, and, consequently, facilitates using logic programming as an inference engine for non-monotonic reasoning. A novel contribution of our paper is a translation T from HLNM programs P to disjunctive logic programs (with default negation) P'.

Let us motivate the need for partial-order clauses. In a recent meeting organized by Krzyststof Apt (president of the Association of Logic Programming), whose aim was to get a clear picture of the current activities in Logic Programming and its role in the coming years, Gelfond pointed out: "We need to find elegant ways for accomplishing simple tasks, e.g. counting the numbers of elements in the database satisfying some property P. At the moment we have neither clear semantics nor efficient implementation of *setof* of other aggregates used for this type of problems" [9]. We propose partial-order clauses as an answer to this issue. The evolution of partial-order clauses is shown in the papers [25,26,23,22,15,29]. Related work on this topic is found in the following papers: [2,20,13,12,31,1,19,30]. A central concept in most of the works that include sets is one which we call *collect-all*. The idea is that a program needs only to define explicitly what are the members of the set and, by assuming a *closed world*, the set becomes totally defined. Moreover, all the just mentioned works, have defined a precise semantics for restricted classes of programs and they agree on the common subclasses. Also, a common assumption is to consider only finite sets. According to our knowledge, our approach generalizes all the work done so far about this topic.

HLNM programs also allow general formulas as the body of a clause. In [17,21] it is found motivation on this issue.

A preliminary research on this topic is presented in [29]. The current paper presents the full language and a more refined semantics.

Our paper is structured as follows: In section 2, we introduce our new paradigm High-Level Non-Monotonic reasoning, giving its grammatical definition and motivation. In section 3, we define the declarative semantics of our language. Finally, in our section we present our conclusions.

2 High-Level Non-monotonic Programs

We will now give a brief informal introduction to HLNM programs to give an intuition of the style of programming in this language and also to provide some context for discussing the semantic issues it raises. HLL clauses have the following form:

$$
\begin{array}{ll}
rule & ::= head \leftarrow body. \mid head. \\
head & ::= literal_s \mid literal_s \ ; \ head \\
literal_s & ::= literal_e \mid POC \\
POC & ::= Fu(list\_terms) \geq Fu(argument_f) \mid Fu(list\_terms) \geq term \\
body & ::= literal_b \mid (body \ Ob \ body) \mid (Q \ Var \ body) \mid (\neg body) \\
literal_b & ::= atom_f \mid literal_e \\
literal_e & ::= -atom \mid atom \\
atom & ::= Pr(list\_terms) \\
atom_f & ::= Fu(list\_terms) = term \\
arguments_f & ::= argument_f \mid argument_f, \ arguments_f \\
argument_f & ::= f(arguments_f) \mid terms \\
list\_terms & ::= term \mid term, \ list\_terms \\
term & ::= Const \mid Var \mid CT(list\_terms)
\end{array}
$$

where Ob, Q, Pr, Fu, CT, and $Cons$ are nonterminal symbols defined as follows: $Ob \in \{\wedge, ;, \rightarrow\}$, $Q \in \{\exists, \forall\}$, Pr is a user-defined predicate symbol, Fu is any functional symbol, CT is any contructor symbol, and $Cons$ is any constant symbol (i.e. a constructor of arity 0). We use ";" to denote the disjunctive logical connective. We also use "," to denote the conjunctive logical connective $\wedge$. We use $-$ and $\neg$ to denote explicit and default negation respectively. A HLNM program is a set of program rules (also called clauses).

We adopt four constructors: *cons* (of arity 2), *nil* (of arity 0), *scons* (of arity 2), and *empty* (of arity 0). The two constructors *cons* and *nil* are used to represent lists as usual in logic programming. Moreover, we prefer the notation $[X|Y]$ instead of $cons(X, Y)$ and [] instead of *nil*. To represent finite sets, we use the two constructors *scons* and *empty*. Again, we prefer to use the more suggestible notation $\{x \backslash t\}$ and ϕ. The notation $\{x \backslash t\}$ refers to a set in which x is one element and t is the remainder of the set. We permit as syntactic sugar $\{expr\}$ to stand for $\{expr \backslash \phi\}$, and $\{e_1, e_2, \ldots, e_k\}$, where all e_i are distinct, to stand for $\{e_1 \backslash \{e_2 \backslash \ldots \{e_k \backslash \phi\}\}\}$. To illustrate the set constructor, matching $\{\mathtt{X} \backslash \mathtt{T}\}$ against a ground set $\{\mathtt{a}, \ \mathtt{b}, \ \mathtt{c}\}$ yields three different substitutions, $\{\mathtt{X} \leftarrow \mathtt{a}, \mathtt{T} \leftarrow \{\mathtt{b}, \mathtt{c}\}\}$, $\{\mathtt{X} \leftarrow \mathtt{b}, \mathtt{T} \leftarrow \{\mathtt{a}, \mathtt{c}\}\}$, and $\{\mathtt{X} \leftarrow \mathtt{c}, \mathtt{T} \leftarrow \{\mathtt{a}, \mathtt{b}\}\}$. One should contrast $\{\mathtt{X} \backslash \mathtt{T}\}$ from $\{\mathtt{X}\} \cup \mathtt{T}$.

The 'logical' view of logic programming is that a program is a theory and computation is logical inference. In the classical approach to logical programming, the inference mechanism is faithful to *logical consequence* in classical first-order logic. This paper adopted the *canonical model* approach, where the inference mechanism is faithful to truth in the "intended" model of the program.

2.1 Negation in Logic Programming.

Default negation has proved to be quite useful in many domains and applications, however, this in not the unique form of negation which is needed for non-monotonic formalisms. Explicit negation has many characteristics which makes it a good candidate to represent non-monotonic reasoning. It can occur in the head of a clause and so it allows us to conclude negatively. Explicit Negation treats negative information and positive information in a simetric form, that is to say, it does not give any kind of preference. However, it allows the representation of exceptions and preference rules. We show how can we represent defeasible reasoning with default and explicit negation. In particular, we discuss the representation of exceptions and preference rules. These ideas are not ours, but taken from [3]. The notion of exception may be expressed in three different ways:

Exceptions to predicates. We express that the rule $angiosperm(X) \longleftarrow tree(X)$ applies whenever possible but can be defeated by exceptions using the rule:

`angiosperm(X)` $\longleftarrow$ `tree(X), ¬ ab(X)`

If there is a tree `a` which is known that is not an angiosperm we may express it by `¬angiosperm(a)`. In this case `¬angiosperm(a)` establishes an exception to the conclusion predicate of the defeasible rule.

Exceptions to rules. A different way to express that a given element is some exception is to say that a given rule must not be applicable to the element. If, for instance, element `a` is an exception to the rule *trees are angiosperms*, we express it as `ab(a)`. In general, we may want to express that a given `X` is abnormal under certain conditions. This is the case where we want to express *Pines are abnormal* to the rule about `angiosperms` given above. We write this as follows:

`ab(X)` $\longleftarrow$ `pine(X)`

Exceptions to exceptions. In general we may extend the notion of exceptioned rules to exception rules themselves, i.e. exception rules may be defeasibles. This will allow us to express an exception to an exception rule.

Preference rules. We may express now preference between two rules, stating that if one rule may be used, that constitutes an exception to the use of the other rule:

`angiosperm(X)`$\longleftarrow$ `tree(X), ¬ ab`$_1$`(X)`
`- angiosperm(X)`$\longleftarrow$ `pine(X), ¬ ab`$_2$`(X)`
`tree(X)` $\longleftarrow$ `pine(X)`

In some cases we want to apply the most specific information; above, there should be (since a pine is a specific kind of tree) an explicit preference of the rule about non-angiosperm pines over the rule about angiosperm trees.

$\texttt{ab}_1\texttt{(X)} \longleftarrow \texttt{pino(X)}, \neg\ \texttt{ab}_2\texttt{(X)}$

Botanic Field. We have studied a small fragment part of the Botanic field (the Mycota Kingdom), where we found that is natural to use exceptions and preference rules. Our information was taken from the *Britanic Encyclopedia OnLine* and our results are presented in [28].

2.2 Disjunctions

Disjunctive reasoning is more expressive and natural to use since it permits direct translation of disjunctive statements from natural language and from informal specifications. The additional expressive power of disjunctive logic programs significantly simplifies the problem of it translation of non-monotonic formalisms into logic programs, and consequently, facilitates using logic programming as an inference engine for non-monotonic reasoning.
The following is a well known example that can not be handled adequately by Circumscription.

Example 1 (Poole's Broken arm, [7]).

```
left-use(X) ← ¬ ab(left,X).
ab(left,X) ← left-brok(X).
right-use(X) ← ¬ ab(right,X).
ab(right,X) ← right-brok(X).
left-brok(fred) ; right-brok(fred) ←.
make-cheque(X) ← left-use(X).
make-cheque(X) ← right-use(X).
disabled(X) ← left-brok(X), right-brok(X).
```

The well known semantics D-WFS and DSTABLE derive that Fred is not disabled (see [7]). We get the same answer in our approach. Moreover, DSTABLE (but not D-WFS) derives that Fred can make out a cheque. We get also this result in our approach.

2.3 Partial-Order Clauses

As we have said, Gelfond pointed out: "We need to find elegant ways for accomplishing simple tasks, e.g. counting the numbers of elements in the database satisfying some property P. At the moment we have neither clear semantics nor efficient implementation of *setof* of other aggregates used for this type of problems" [9]. We propose to use partial-order clauses to solve this problem. The first author of this paper has done an extensive research on this topic in [25,26,23,22,15,29] and we have strong evidence to make us think that our results in this paper will help to obtain a final answer.

Modeling Setof. The original motivation of partial-order clauses was to include a kind of setof operator. There are two mayor problems with the setof operator in PROLOG. First, it has no formal definition and second, it is very weak. See [23].

Example 2. Consider that we want to obtain the set of all students that are taking both the cs101 class and cs202 class. We write a clause in Prolog that do the given function as follow:

$\texttt{both(X)} \ :- \ \texttt{setof(W,(cs101(W),cs202(W)),X)}$

while in our paragigm this represented by:

$\texttt{both(X)} \ \geq \ \{\texttt{X}\} \leftarrow \texttt{cs101(W)},\texttt{cs202(W)}$

We note that our paradigm is not only a change of notation, but we define a more powerful notion of 'setof' than PROLOG. The reader is invited to see an example of this nature in [29].

General Domains. We generalize our approach of partial-orders to different domains and not just sets. The following example illustrates this idea.

Example 3 (Min-Max Program1 [13]).

$\texttt{p(X)} \leq \texttt{X1} \ \leftarrow \texttt{final(X,X1).}$
$\texttt{p(X)} \leq \texttt{X2} \ \leftarrow \texttt{edge2(X,Y), q(Y)=X2.}$
$\texttt{q(X)} \geq \texttt{X3} \ \leftarrow \texttt{final(X,X3).}$
$\texttt{q(X)} \geq \texttt{X4} \ \leftarrow \texttt{edge1(X,Y), p(Y)=X4.}$

This program models a two-player game defined on a bipartite graph represented by predicates `edge1` and `edge2`. The function `p` obtains the minimum value, while `q` obtains the maximum value. The program is considered non-monotonic due to the interaction of $\geq$ with $\leq$. Consider the following extensional database:

```
final(d,0).  edge1(a,b).  edge2(b,d).
final(e,1).  edge1(a,c).  edge2(b,e).
final(f,-1).              edge2(c,f).
final(g,0).               edge2(c,g).
```

In our approach we obtain the intended model, where `p(a)=0`. Our operational semantics with pruning (introduced in [27]) behaves very much as the well known alpha-beta procedure. Other interesting examples about partial-order are presented in [23].

It is important to observe, as will see in the following section, that we have a formal definition of the semantics of partial-order clauses.

3 Declarative Semantics

We first present our definition of the declarative semantics for propositional disjunctive programs. Then we explain how we generalize our results to the general framework of HLNM programs.

3.1 Declarative Semantics of Propositional Disjunctive Programs

A *signature* is any finite set. An *atom* is any element of a signature. A *positive literal* is an atom. A *negative literal* is a formula of the form $\neg a$, where a is an atom. A *literal* is a positive literal or a negative literal. Two literals are of the same sign if both are positive or both are negative. Propositional disjunctive clauses are of the form:

$A_1; \ldots; A_n \leftarrow L_1, \ldots L_m$

where $n \geq 1, m \geq 0$, every A_i is an atom and every L_i is a literal. When $m = 0$, we say that the clause is a fact. When $n = 1$ the clause is called normal. A normal program consists of just normal clauses. Traditional logic programming is based on this kind of programs. When $m = 0$ and $n = 1$ the clause is called a normal fact. We also would like to denote a disjunctive clause by $C{:=}\ \mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$, where $\mathcal{A}$ denotes the set of atoms in the head of the clause (namely $\{A_1, \ldots A_n\}$;), $\mathcal{B}^+$ denotes the set of positive atoms in the body, and $\mathcal{B}^-$ denotes the set of negative atoms in the body. A pure disjunction a disjunction of literals of the same sign.

The stable semantics and the well founded semantics are the two most well known definitions of semantics for Logic Programs. Our proposed semantics is different than in previous aproaches and more powerful even for normal programs. We allow the combination of the idea of negation as failure with reasoning by cases to obtain a powerful use of negation.

We now provide our formal definition of our declarative semantics.
We start with some definitions. Given a program P, we define $HEAD(P) := \bigcup_{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P} \mathcal{A}$. It will be useful to map a disjunctive program to a normal program. Given a clause $C{:=}\ \mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$, we write *dis-nor(C)* to denote the set of normal clauses:

$\{a \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \cup (\mathcal{A} \setminus \{a\}) | a \in \mathcal{A})\}$.

We extend this definition to programs as follows. If P is a program, let *dis-nor(P)* denotes the normal program: $\bigcup_{C \in P} dis - nor(C)$.

We first discuss the notion of a supported model for a disjunctive program. It generalizes the notion of a supported model for normal programs (which in turns is equivalent to a model of the Clark's completion of a program).

Definition 1 (Supported model).
Let P be a program. A supported model M of P is model of $dis - nor(P)$ such that for every atom a that occurs in P and is true in M, there exists a clause C in $dis - nor(P)$ such that a is the head of C and the body of C is true in M.

We now discuss the notion of a D-WFS partial model. The key concept of this approach is the idea of a *transformation* rule. We adopt the transformation rules introduced in [5,7], which are: RED$^+$, RED$^-$, GPPE, TAUT, SUB. We define $\mathcal{CS}_1$ to be the rewriting system which contains the just mentioned rules. It is known that this system is confluent and terminating, see [6].

Definition 2 (D-WFS partial model). *Let P be a program and P' its normal form, i.e. P' is the program obtained after reducing P by $\mathcal{CS}_1$. The D-WFS*

partial model of P is defined as: $\{a|$ *a is a normal fact of* $P'\ \} \cup \{\neg a|$ *a is an atom in the language of P that does not occurs in* $P'\ \}$.

We can now present the main definition of this section:

Definition 3. *Let P be a program. We define a DWFS-DCOMP model as a two-valued minimal supported model that extends D-WFS(P). Any such extension is a model that agrees with the true/false assigments given by D-WFS(P). The scenario DWFS-DCOMP semantics of P is the set of DWFS-DCOMP models of P. The sceptical DWFS-DCOMP semantics for P is the set of pure disjunctions that are true in every DWFS-DCOMP model of P. If no such model exists then we say that the program is inconsistent.*

In the above definition we stress the importance of considering minimal models ([21]), since they are critical in logic programming. In this paper our concern is only in the sceptical semantics.

3.2 Moving to Predicates and Adding the Set Constructor

We will work with Herbrand interpretations, where the Herbrand Universe of a program P consists only of ground terms, and is referred to as U_P. The Herbrand Base B_P of a program P consists of ground atoms as usual.

The generalization of the function $dis - nor$ to predicate logic is straightforward. Given this , we need a definition of supported model for predicate normal programs. This has been done in [21]. D-WFS for predicate programs is presented in full detail in [11]. We can therefore define DWFS-DCOMP models with no problem.

With respect to the set constructor we proceed as follows. We continue working with Herbrand interpretations. But due to the equational theories for constructors, the predicate = defines an equivalence relation over the Herbrand Universe. But, we can always *contract* a model to a so-called normal model where = defines only an identity relation [24] as follows: Take the domain D' of I to be the set of equivalence classes determined by = in the domain U_P of I. Then use Herbrand $\equiv$-interpretations, where $\equiv$ denotes that the domain is a quotient structure. We then should refer to elements in D' by $[t]$, i.e. the equivalence class of the element t, but in order to make the text more readable, we will refer to the $[t]$ elements just as t, keeping in mind that formally we are working with the equivalence classes of t. These details are explained in [14].

3.3 Adding Partial-Order Clauses

The first step is to obtain the flattened form [14] of every partial-order clause. Consider the following example:

`f(X)` $\geq$ `g(h(X)) ; f(Z)` $\geq$ `{m(Z)}` $\leftarrow$ `l(X).`

then its flattened form is:

`(f(X)` $\geq$ `X1` $\leftarrow$ `h(X)=X2, g(X2)=X1 ; f(Z)` $\geq$ `{ Z1 }` $\leftarrow$ `m(Z)= Z1)` $\leftarrow$ `l(X).`

The second step is to transform this formula to a disjunctive clause. With this same example, we get:

$\texttt{f(X)} \geq \texttt{X1; f(Z)} \geq \{\ \texttt{Z1}\ \} \leftarrow \texttt{h(X)=X2, g(X2)=X1, m(Z)=Z1, l(X).}$

As the third step we translate the clause to its relational form. Again, using this example, we get:

$\texttt{f}_{\geq}\texttt{(X,X1); f}_{\geq}\texttt{(Z,\{Z1\})} \leftarrow \texttt{h}_{=}\texttt{(X,X2), g}_{=}\texttt{(X2,X1), m}_{=}\texttt{(Z,Z1), l(X).}$

These steps can easily be defined for the general case. We suggest the reader to see [26] to check the details.

The fourth step is to add axioms that related the predicate symbols $\texttt{f}_{=}$ with $\texttt{f}_{\geq}$ for each functional symbol f. Let us consider again our example. The axioms for f in this case are as follows:

(1) $\texttt{f}_{=}\texttt{(Z, S)} \leftarrow \texttt{f}_{\geq}\texttt{ (Z, S)},\ \neg\, \texttt{f}_{>}\texttt{(Z, S)}$

(2) $\texttt{f}_{>}\texttt{(Z, S)} \leftarrow \texttt{f}_{\geq}\texttt{(Z,S1)},\ \texttt{S1 > S}$

(3) $\texttt{f}_{\geq}\texttt{(Z, S)} \leftarrow \texttt{f}_{\geq}\texttt{(Z,S1)},\ \texttt{S1 > S}$

(4) $\texttt{f}_{\geq}\texttt{ (Z, }\bot\texttt{)}$

(5) $\texttt{f}_{\geq}\texttt{(Z,C)} \leftarrow \texttt{f}_{\geq}\texttt{(Z,C}_1\texttt{)},\ \texttt{f}_{\geq}\texttt{(Z,C}_2\texttt{)},\ \ \texttt{lub(C}_1\texttt{,C}_2\texttt{,C).}$

We understand that `S1 > S` means that $\texttt{S1} \geq \texttt{S}$ and $\texttt{S1} \neq \texttt{S}$. $\bot$ is a constant symbol used to represent the bottom element of the lattice and $\texttt{lub}(C_1, C_2, C)$ interprets that C is the least upper bound of C_1 and C_2. The first two clauses are the same (modulo notation) as in definition 4.2 in [32]. Clause (5) is not useful for total-order domains. It is easy to see that symmetric definitions can be provided for $f_{\leq}$ symbols.

3.4 Allowing a General Formula as the Body Clause

Here, we adopt the approach suggested by Lloyd in his well known book [21]. The idea is to apply an algorithm that transform a clause (with a general formula as the body) into one or more clauses with simple bodies (i.e. conjunction of literals). Space limitations disallow us to explain the algorithm, instead, we work out an example and let the interested reader to check the details in the above mentioned reference.

$\texttt{rp}(\{\texttt{X}\backslash\_\}) \geq \{\texttt{X}\} \leftarrow \forall \texttt{Y}(\texttt{d}(\texttt{X},\texttt{Y}) \rightarrow \texttt{ontime}(\texttt{X},\texttt{Y}))$

If we use the rules of [21] the translation becomes:

$\texttt{rp}(\{\texttt{X}\backslash\_\}) \geq \{\texttt{X}\} \leftarrow \neg\texttt{c}(\texttt{X})$

$\texttt{c}(\texttt{X}) \leftarrow \texttt{d}(\texttt{X},\texttt{Y}), \neg\texttt{ontime}(\texttt{X},\texttt{Y}).$

where c is new predicate symbol.

3.5 Explicit Negation

We show how to reduce programs with explicit negation to programs without it. The idea is originally considered in [4] but we have also explored it in [28]. Let T be the program that includes explicit negation. Let P be the program T plus the following set of clauses:

$-p(X_1, \ldots, X_n) \leftarrow \neg p(X_1, \ldots X_n)$

for all predicate symbols p from the language of T. For any predicate symbol p ocurring in P, let p' be a new predicate symbol of the same arity. The atom $p'(t_1, \ldots p_n)$ will be called the *positive form* of the negative $literal_e$ $-p(t_1, \ldots p_n)$. Every positive $literal_e$ is, by definition its own form. Let P' the program that we obtain form P by replacing every $literal_e$ by its positive form.

There is a simple way of evaluating queries in T. To obtain an answer for p run queries p and p' on P'. If the answer to p is yes then we say that T derives p. If the answer to p' is yes then we say that T does not derives p.

4 Conclusions

We presented our notion of HLNM program. It includes standard logic programming plus disjunctions, partial-order clauses, two kinds of negation and general body formulas. For this purpose, we had to define a new declarative semantics (an hybrid of D-WFS and supported models) that is suitable to express the above mentioned notions. To our knowledge, this is the first proposal that extends Logic Programming that far. We presented several and different interesting problems considered in the literature. HLNN programming defines the intended meaning of each of them. Our paradigm combines several ideas from several authors and our own earlier work and it seems promising.

The bad news is that the operational semantics of the full language is not computable. However we can identify large and interesting fragments of the language that are computable. Our main research on this direction is presented in [23,22].

References

1. E. Pontelli A. Dovier, E. G. Omodeo and G. Rossi. {log}: A logic programming language with finite sets. In *Proc. 8th International Conference of Logic Programming*, pages 111–124, Paris, June 1991. 1991.
2. S. Abiteboul and S. Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.
3. Jose Julio Alferes and Luiz Moniz Pereira, editors. *Reasoning with Logic Programming*, LNAI 1111, Berlin, 1996. Springer.
4. Chitta Baral and Michael Gelfond. Logic Programming and Knowlege Representation. *Journal of Logic Programming*, 19-20:73–148, 1994.
5. Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. (Extended abstract appeared in: Characterizations of the Stable Semantics by Partial Evaluation *LPNMR, Proceedings of the Third International Conference, Kentucky*, pages 85–98, 1995. LNCS 928, Springer.).
6. Stefan Brass and Jürgen Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning*, 20(1):143–165, 1998. (Extended abstract appeared in: Characterizing D-WFS: Confluence and Iterated GCWA. *Logics in Artificial Intelligence, JELIA '96*, pages 268–283, 1996. Springer, LNCS 1126.).

7. Gerhard Brewka and Jürgen Dix. Knowledge representation with logic programs. Technical report, Tutorial Notes of the 12th European Conference on Artificial Intelligence (ECAI '96), 1996. Also appeared as Technical Report 15/96, Dept. of CS of the University of Koblenz-Landau. Will appear as Chapter 6 in *Handbook of Philosophical Logic*, 2nd edition (1998), Volume 6, Methodologies.
8. Baral C. and Son T.C. Formalizing sensing actions: a transition function based approach. In *Cognitive Robotics Workshop of AAAI Fall Symposium*, pages 13–20, 1998.
9. Jürgen Dix. The Logic Programming Paradigm. *AI Communications*, Vol. 11, No. 3:39–43, 1998. Short version in Newsletter of ALP, Vol. 11(3), 1998, pages 10–14.
10. Jürgen Dix, Mauricio Osorio, and Claudia Zepeda. A General Theory of Confluent Rewriting Systems for Logic Programming and its Applications. *Annals of Pure and Applied Logic*, submitted, 2000.
11. Jürgen Dix and Frieder Stolzenburg. A Framework to incorporate Nonmonotonic Reasoning into Constraint Logic Programming. *Journal of Logic Programming*, 37(1,2,3):47—76, 1998. Special Issue on *Constraint Logic Programming*, Guest Editors: Kim Marriott and Peter Stuckey.
12. S. Greco G. Ganguly and C. Zaniolo. Minimum and maximum predicates in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 154–16. 1991.
13. A. Van Gelder. The well-founded semantics of aggregation. In *Proc. ACM 11th Principles of Database Systems*, pages 127–138. San Diego, 1992.
14. D. Jana and Bharat Jayaraman. Set constructors, finite sets, and logical semantics. *Journal of Logic Programming*, 38(1):55–77, 1999.
15. Bharat Jayaraman and K. Moon. Subset logic programs and their implementation. *Journal of Logic Programming*, To appear:?–?, 1999.
16. Jürgen Dix Jose Arrazola and Mauricio Osorio. Confluent rewriting systems in non-monotonic reasoning. *Computacion y Sistemas*, Volume II, No. 2-3:104–123, 1999.
17. R. Kowalski. *Logic for problem solving.* North Holland Publishing Company, 1979.
18. R. Kowalski and F. Toni. Abstract Argumentation. *Artificial Intelligence and Law Journal*, pages 275–296, September 1996.
19. G. M. Kuper. Logic programming with sets. *JCSS*, 41(1):44–64, 1990.
20. M. Liu. Relationlog: A Typed Extension to Datalog with Sets and Tuples. In John Lloyd and Evan Tick, editors, *Proceedings of the 1995 Int. Symposium on Logic Programming*, pages 83–97. MIT, June 1995.
21. John W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1987. 2nd edition.
22. Bharat Jayaraman Mauricio Osorio and J. C. Nieves. Declarative pruning in a functional query language. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 588–602. MIT Press, 1999.
23. Bharat Jayaraman Mauricio Osorio and David Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.
24. E. Mendelson. *Introduction Mathematical logic (3th ed.).* Wasdworth and Brooks/Cole Advanced Books and Software, United Stated, 1987.
25. Mauricio Osorio. Semantics of partial-order programs. In J. Dix and L.F. del Cerro andU. Furbach, editors, *Logics in Artificial Intelligence (JELIA '98)*, LNCS 1489, pages 47–61. Springer, 1998.
26. Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New generation computing*, 17(3):255–284, 1999.

27. Mauricio Osorio and J. C. Nieves. Extended partial-order logic programming. In Gianfranco Rossi and Bharat Jayaraman, editors, *Proceedings of the Workshop on Declarative Programming with Sets*, pages 19–26, Paris, France, 1999.
28. Mauricio Osorio and Erika Saucedo. Aplicación de wfsx en el campo de la botánica. In *Proceedings of II Encuentro Nacional de Computación*, pages ?–? Mexico, 1999.
29. Mauricio Osorio and Fernando Zacarias. High-level logic programming. LNCS 1762, pages ?–?, Berlin, 2000. Springer Verlag.
30. K. Ross. Modular stratification and magic sets for datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.
31. K.A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *Proc. 11th ACM Symp. on Principles of Database Systems*, pages 114–126. San Diego, 1992.
32. Allen Van Gelder. The Well-Founded Semantics of aggregation. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, USA*, pages 127–138. ACM Press, 1992.