Electronic Notes in Theoretical Computer Science 48 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume48.html` pp. 1 – 16

Operational Semantics for Reexecution-based Analysis of Logic Programs with Delay Declarations

Agostino Cortesi [1,2] Sabina Rossi [1,3]

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Mestre–Venezia, Italy

Baudouin Le Charlier [4]

Institut d'Informatique
Facultès Universitaires Notre-Dame de la Paix
Namur, Belgium

Abstract

We draw concrete and abstract operational semantics towards the analysis of logic programs with delay declarations. This is the basis to generalize the abstract interpreter GAIA to deal with logic programs employing dynamic scheduling. The concrete and abstract semantics are introduced explicitly and express both deadlock information and qualified answers. Sure deadlock and sure no deadlock information might be eventually inferred by the resulting analysis.

1 Introduction

Most of the logic programming languages in use (e.g., SICStus Prolog [10], Prolog-III, CHIP, SEPIA, etc.) do not force the user to follow the Prolog left-to-right scheduling rule; instead, in order to gain efficiency, they provide dynamic scheduling: atom calls are delayed until their arguments are sufficiently instantiated, and procedures are augmented with delay declarations. The analysis of logic programs with dynamic scheduling was first investigated

[1] Partially supported by the Italian MURST with the National Reserch Project "Certificazione automatica di programmi mediante interpretazione astratta".
[2] Email:`cortesi@dsi.unive.it`
[3] Email:`srossi@dsi.unive.it`
[4] Email:`ble@info.fundp.ac.be`

by Marriott *et al.* in [15,9]. A more general (denotational) semantics of this class of programs, extended to the general case of CLP, has been presented by Falaschi *et al.* in [8], while verification and termination issues have been investigated by Apt and Luitjes in [2] and by Marchiori and Teusink in [14], respectively.

In this paper, we discuss an alternative, strictly operational approach to the definition of concrete and abstract semantics for logic programs with delay declarations.

The main intuitions behind our proposal can be summarized as follows:

- to define in a uniform way concrete, collecting, and abstract semantics, in the spirit of [11]: this allows us to easily derive correctness proofs of the whole analyses;
- to define the analysis as an extension of the framework depicted in [11]: this allows us to reuse existing code for program analysis, with minimal additional effort;
- to explicitly derive deadlock information (definite deadlock, possible deadlock, deadlock freeness), producing as a result of the analysis an approximation of concrete qualified answers;
- to apply the reexecution technique developed in [12], that plays a crucial role here: at each step, an atom a whose delay declaration is satisfied by the activation substitution is executed; if its computation fully succeeds, some of the delayed atoms may be awakened by the resulting answer substitutions; if this is not the case (i.e., if during the computation of a a sequence of atoms, such that each of them do not satisfies the corresponding delay declaration, is reached), then atom a is added to the list of delayed atoms that will be reconsidered during the next reexecution step. This strategy allows us to avoid call state explosions.

The main difference between our approach and the ones already presented in literature is that we are mainly focused on analysis issues, in particular on deadlock analysis. This motivates the choice of a strictly operational approach, where deadlock information is explicitly maintained.

This paper illustrates the crucial steps toward the definition and implementation of an extension of the GAIA abstract interpreter [11] to deal with dynamic scheduling. It mainly focuses on the (concrete and abstract) semantics, and we believe that this work, even though it can be clearly considered to be just in the preliminaries of the picture above, represents in itself a valuable contribution.

The rest of the paper is organized as follows. Section 2 recalls basic notions about logic programs with delay declarations. Section 3 depicts the concrete semantics transition rules. Section 4 sketches the main features of the collecting and abstract semantics, and discusses our generic fixpoint algorithm. Section 5 concludes the paper.

2 Logic Programs with Delay Declarations

Logic programs with delay declarations consist of two parts: a logic program and a set of delay declarations, one for each of its predicate symbols.
A *delay declaration* associated for an n-ary predicate symbol p has the form

$$\texttt{DELAY}\ \ p(x_1,\ldots,x_n)\ \ \texttt{UNTIL}\ \ Cond(x_1,\ldots,x_n)$$

where $Cond(x_1,\ldots,x_n)$ is a formula in some assertion language. We are not concerned here with the syntax of this language since it is irrelevant for our purposes. The meaning of such a delay declaration is that an atom $p(t_1,\ldots,t_n)$ can be selected in a query only if the condition $Cond(t_1,\ldots,t_n)$ is satisfied. In this case we say that the atom $p(t_1,\ldots,t_n)$ *satisfies* its delay declaration.

A derivation ξ of a program augmented with delay declarations can be finite or infinite. Let ξ be finite. We say that ξ *succeeds* if it ends with the empty goal; ξ *fails* if it ends with a non-empty goal the selected atom of which satisfies its delay declaration but does not unify with the head of any clause in the program; ξ *deadlocks* if it ends with a non-empty goal no atom of which satisfies its delay declaration. A finite non-failing derivation ξ of a program with delay declarations computes a *qualified answer* which is a pair $\langle\theta, d\rangle$ where d is the last goal (that is a sequence of delayed atoms) and θ is the substitution obtained by concatenating the computed mgu's from the initial goal. Notice that if ξ is successful then d is the empty goal and θ restricted to the variables of the initial goal is its computed answer substitution. We denote by $qans_P(g)$ the set of qualified answers for a goal g and a program P.

We restrict our attention to delay declarations which are *closed under instantiation*, i.e., if an atom satisfies its delay declaration then also all its instances do. Notice that this is the choice of most of the logic programming systems dealing with delay declarations such as IC-Prolog, NU-Prolog, Prolog-II, Sicstus-Prolog, Prolog-III, CHIP, Prolog M, SEPIA, etc.

Example 2.1 Consider the programs APPEND and IN_ORDER defined below

```
%  append(Xs,Ys,Zs) ← Zs is the result of concatenating the lists Xs and Ys
   append([H|Xs],Ys,[H|Zs]) ← append(Xs,Ys,Zs).
   append([],Ys,Ys).

%  in_order(Tree,List) ← List is an ordered list of the nodes of Tree
   in_order(tree(Label,Left,Right),Xs) ← in_order(Left,Ls),
      in_order(Right,Rs), append(Ls,[Label|Rs],Xs).
   in_order(void,[]).
```

together with the query

$Q :=$ `read_tree(Tree), in_order(Tree,List), write_list(List).`

where the predicates `read_tree` and `write_list` are defined elsewhere in the program. If `read_tree` cannot read the whole tree at once it would be nice to be able to run `in_order` and `write_list` on the available input. This can only be done if one uses a dynamic selection rule (Prolog's rule would call `in_order` only when `read_tree` had finished, while other fixed rules would immediately diverge). In order to avoid nontermination one should adopt appropriate delay declarations, namely,

```
DELAY in_order(T,_) UNTIL nonvar(T).
DELAY append(Ls, _, _) UNTIL nonvar(Ls).
DELAY write_list(Ls,_) UNTIL nonvar(Ls).
```

These declarations avoid that `in_order`, `append` and `write_list` be selected "too early". Notice that with these declarations `IN_ORDER` enjoys a parallel execution by means of interleaving.

Under the assumption that delay declarations are closed under instantiation, the following result, which is a variant of Theorem 4 in Yelick and Zachary [18], holds.

Theorem 2.2 *Let P be a program augmented with delay declarations, g be a goal and g' be a permutation of g. Then $qans_P(g)$ and $qans_P(g')$ are equals modulo the ordering of delayed atoms.*

It follows that both successful and deadlocked derivations are "independent" from the choice of the selection rule. Moreover, Theorem 2.2 allows us to treat goals as multisets instead of sequences of atoms.

3 Concrete Semantics

In this section we describe a concrete semantics for pure Prolog augmented with delay declarations. The concrete semantics is the link between the standard semantics of the language and the abstract one.

We assume a preliminary knowledge of logic programming (see, [1,13]).

Programs are assumed to be normalized according to the abstract syntax given in Fig. 1. The variables occurring in a literal are distinct; distinct procedures have distinct names; all clauses of a procedure have exactly the same head; if a clause uses m different program variables, these variables are e $x_1, \ldots, x_m$.
If $g := a_1, \ldots, a_n$ we denote by $g \setminus a_i$ the goal $g' := a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

3.1 Basic Semantic Domains

The basic semantic domains of substitutions are defined as follows.

We assume the existence of two disjoint and infinite sets of variables, denoted by PV and SV. Elements of PV are called *program variables* and are denoted by x_1, x_2, …, x_i, …. The set PV is totally ordered; x_i is the i-th element of PV. Elements of SV are called *standard variables* and are denoted

P	$\in$ *Programs*	P	$::= pr_1, \ldots, pr_n \ (n > 0)$
pr	$\in$ *Procedures*	pr	$::= c_1, \ldots, c_n \ (n > 0)$
c	$\in$ *Clauses*	c	$::= h : -g.$
h	$\in$ *ClauseHeads*	h	$::= p(x_1, \ldots, x_n) \ (n \geq 0)$
g	$\in$ *AtomSequences*	g	$::= a_1, \ldots, a_n \ (n \geq 0)$
$\bar{g}$	$\in$ *DelayedAtomSequences*	$\bar{g}$	$::= a_1, \ldots, a_n \ (n \geq 0)$
l	$\in$ *Literals*	l	$::= a \mid b$
a	$\in$ *Atoms*	a	$::= p(x_{i_1}, \ldots, x_{i_n}) \ (n \geq 0)$
b	$\in$ *Built-ins*	b	$::= x_i = x_j \mid x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$
p	$\in$ *ProcedureNames*		
f	$\in$ *Functors*		
x_i	$\in$ *ProgramVariables (PV)*		

Fig. 1. Abstract Syntax of Normalized Programs

by letters y and z (possibly subscripted). Terms are built using standard variables only.

Standard substitutions are substitutions in the usual sense [1,13] which use standard variables only. The set of standard substitutions is denoted by SS. Renamings are standard substitutions that define a permutation of standard variables. The domain and the codomain of a standard substitution σ are denoted by $dom(\sigma)$ and $codom(\sigma)$, respectively. We denote by $mgu(t_1, t_2)$ the set of standard substitutions that are a most general unifier of terms t_1 and t_2.

A program substitution is a set $\{x_{i_1}/t_1, \ldots, x_{i_n}/t_n\}$, where $x_{i_1}, \ldots, x_{i_n}$ are distinct program variables and $t_1, \ldots, t_n$ are terms. Program substitutions are not substitutions in the usual sense; they are best understood as a form of program store which expresses the state of the computation at a given program point. It is meaningless to compose them as usual substitutions or to use them to express most general unifiers. The domain of a program substitution $\theta = \{x_{i_1}/t_1, \ldots, x_{i_n}/t_n\}$, denoted by $dom(\theta)$, is the set of program variables $\{x_{i_1}, \ldots, x_{i_n}\}$. The codomain of θ, denoted by $codom(\theta)$, is the set of standard variables occurring in $t_1, \ldots, t_n$. Program and standard substitutions cannot be composed. Instead, standard substitutions are *applied* to program substitutions. The application of a standard substitution σ to a program substitution $\theta = \{x_{i_1}/t_1, \ldots, x_{i_n}/t_n\}$ is the program substitution $\theta\sigma = \{x_{i_1}/t_1\sigma, \ldots, x_{i_n}/t_n\sigma\}$. The set of program substitutions is denoted by PS. The application $x_i\theta$ of a program substitution θ to a program variable x_i is defined only if $x_i \in dom(\theta)$; it denotes the term bound to x_i in θ. Let D be a finite subset of PV and θ be a program substitution such that $D \subseteq dom(\theta)$. The *restriction* of θ to D, denoted by $\theta_{/D}$, is the program substitution such that $dom(\theta_{/D}) = D$ and $x_i(\theta_{/D}) = x_i\theta$, for all $x_i \in D$. We denote by PS_D the set of program substitutions whose domain is D.

We consider also the possible states that an output state may enjoy with respect to deadlock. An atom in the body of a clause may belong to three (successive) states: non-activable (when its activation substitution does not satisfies its delay declaration), or reexecutable (when its activation substitution satisfies its delay declaration, but in its derivation tree a deadlock is encountered that forces the atom to be reconsidered afterwards), or executable. Of course, its computation may get stuck at each stage, giving raise to a deadlock detection. We represent the deadlock information explicitly, by means of a deadlock state. A *deadlock state* is an element in the set $\{\delta, \nu\}$, where δ denotes definite deadlock, and ν denotes no deadlock. This set will be extended in the collecting (and abstract) semantics by introducing an additional state μ (standing for *may* deadlock).

3.2 Concrete Behaviors

The notion of concrete behavior provides a mathematical model for the input/output behavior of programs. To simplify the presentation, we do not parameterize the semantics with respect to programs. Instead, we assume given a fixed underlying program P augmented with delay declarations.

We define a *concrete behavior* as a relation from input states to output states as defined below. The *input states* have the form

- $\langle \theta, p \rangle$, where p is the name of a procedure and θ is a program substitution also called activation substitution. Moreover, $\theta \in PS_{\{x_1,\ldots,x_n\}}$, where $x_1, \ldots, x_n$ are the variables occurring in the head of every clause of p.

The *output states* have the form

- $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_n\}}$ and κ is a deadlock state, i.e., $\kappa \in \{\delta, \nu\}$. In case of no deadlock (i.e., $\kappa = \nu$) θ' restricted to the variables $\{x_1, \ldots, x_n\}$ is a computed answer substitution (the one corresponding to a successful derivation), while in case of definite deadlock (i.e., $\kappa = \delta$) θ' is the substitution part of a qualified answer to p and coincides with a partial answer substitution for it.

We use the relation symbol $\longmapsto$ to represent concrete behaviors, i.e., we write $\langle \theta, p \rangle \longmapsto \langle \theta', \kappa \rangle$: this notation emphasizes the similarities between this concrete semantics and the structural operational semantics for logic programs defined in [12]. Concrete behaviors are intended to model finite non-failing derivations of atomic queries.

3.3 Concrete Semantics Rules

The concrete semantics of an underlying program P with delay declarations is the least fixpoint of a continuous transformation on the set of concrete behaviors. This transformation is defined in terms of ten semantic rules that naturally extend concrete behaviors in order to deal with clauses and atoms.

In particular, a concrete behavior is extended through intermediate states of the form $\langle \theta, c \rangle$ and $\langle \theta, g, \bar{g} \rangle$, where c is a clause and $g, \bar{g}$ are subsets of a body of a clause of P, in such a way that

- each pair $\langle \theta, c \rangle$, where c is a clause, $\theta \in PS_{\{x_1,\dots,x_n\}}$ and $x_1, \dots, x_n$ are the variables occurring in the head of c, is related to an output state $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\dots,x_n\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state;
- each triplet $\langle \theta, g, \bar{g} \rangle$, where g is a set of atoms not considered yet or whose activation state does not satisfy the delay declaration, $\bar{g}$ is a set of delayed atoms that have been recognized to be reexecutable, $\theta \in PS_{\{x_1,\dots,x_m\}}$ and $x_1, \dots, x_m$ are the variables occurring in $(g, \bar{g})$, is related to an output state $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\dots,x_m\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state;

We briefly recall here the concrete operations which are used in the definition of the concrete semantics. The reader may refer to [11] for a complete description of all operations but the last two, `DELAY` and `EQUIV`, that are brand new.

- `EXTC` is used at clause entry: it extends a substitution on the set of variables occurring in the body of the clause.
- `RESTRC` is used at clause exit: it restricts a substitution on the set of variables occurring in the head of the clause.
- `RETRG` is used when a literal l occurring in the body of a clause is analyzed. Let $\{x_{i_1}, \dots, x_{i_n}\}$ be the set of variables occurring in l. This operation expresses a substitution in terms of the formal parameters $x_1, \dots, x_n$.
- `EXTG` it is used to combine the analysis of a built-in or a procedure call (expressed in terms of parameters $x_1, \dots, x_n$) with the activating substitution.
- `UNIF-FUNC` and `UNIF-VAR` are the operations that actually perform the unification of equations of the form $x_i = x_j$ or $x_{i_1} = f(x_{i_2}, \dots, x_{i_n})$, respectively.
- `DELAY` is the operation that verifies whether the delay declaration associated to an atom is satisfied or not. Let a be an atom in the body of a clause c, D be the set of all the variables occurring in c and $\theta \in PS_D$. `DELAY`(a, θ) is *true* if $a\theta$ does not satisfy its delay declaration, *false* otherwise.
- `EQUIV` is used to verify whether two program substitutions with the same domain are equal modulo renaming of range variables.

The concrete semantic rules are depicted in Fig. 2. The definition proceeds by induction on the syntactic structure of P. The semantics of a program P with delay declarations can be defined as a fixpoint of this transition system.

- Rule $\mathbf{R}_1$ defines the result of executing a procedure call. This is obtained by executing any clause defining it.
- Rule $\mathbf{R}_2$ defines the result of executing a clause. This is obtained by executing its body under the same activation substitution. Notice that, at the beginning, all literals in the body of the clause have to be considered and there are no delayed atoms (i.e., g is the body of c and $\bar{g}$ is the empty goal denoted by $< >$).
- Rules $\mathbf{R}_3$ and $\mathbf{R}_4$ specify the execution of built-ins: the usual unification operations are applied. Notice that built-ins can only occur in the g part of the goal, i.e., the subset of literals in the current goal that have not been considered yet.
- Rules $\mathbf{R}_5$ and $\mathbf{R}_6$ define the execution of an atom a in the case that a has not yet been considered and the activation substitution θ satisfies the corresponding delay declaration. The first rule applies when the execution of a is deadlock free. The second rule applies when the execution of a with the current activation substitution falls into deadlock: in this case, a is moved in the delayed atoms list, waiting for a reexecution step.
- Rules $\mathbf{R}_7$ and $\mathbf{R}_8$ are the reexecution rules. Atoms that have been delayed and that might be executable without falling into deadlock now are reconsidered. Notice that, by the assumption that delay declarations are closed under instantiation, we do not need to check whether an atom in the delayed list verifies its delay declaration, as the activation substitution is possibly more instantiated now. Operation `EQUIV` in rule $\mathbf{R}_8$ guarantees that only atoms whose reexecution produces a more instantiated result are allowed to be reexecuted.
- Rule $\mathbf{R}_9$ defines a deadlock situation: the activation substitution θ satisfies none of the delay declarations of atoms in g, and all atoms in $\bar{g}$ deadlock.
- Rule $\mathbf{R}_{10}$ defines the result of executing the empty goal, generating a successful output substitution.

4 Collecting and Abstract Semantics

In this section we briefly describe how to define abstract semantics for logic programs with delay declarations that are based on the concrete semantics defined in the previous section.

As usual in the *Abstract Interpretation* literature [6,7], we proceed in three steps. First, we depict a collecting semantics, by lifting the concrete semantics to deal with sets of substitutions. Then, any abstract semantics will be defined as an abstraction of the collecting semantics: it is sufficient to provide an abstract domain that enjoys a Galois connection with the concrete domain $\wp(\mathit{Subst})$, and a suite of abstract operations that safely approximate the concrete ones. Finally, we draw an algorithm to compute a (post-)fixpoint of an

$$\textbf{R1}\quad \frac{\begin{array}{c} c \text{ is a clause defining } p \\ \langle \theta, c\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, p\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R2}\quad \frac{\begin{array}{c} c := h : -g \\ \theta_1 = \mathtt{EXTC}(c, \theta) \\ \langle \theta_1, g, < > \rangle \longmapsto \langle \theta_2, \kappa\rangle \\ \theta' = \mathtt{RESTRC}(c, \theta_2) \end{array}}{\langle \theta, c\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R3}\quad \frac{\begin{array}{c} b \text{ is a built-in of } g \\ g' := g \setminus b \\ b := x_i = x_j \\ \theta_1 = \mathtt{RESTRG}(b, \theta) \\ \theta_2 = \mathtt{UNIF\_VAR}(\theta_1) \\ \theta_3 = \mathtt{EXTG}(b, \theta, \theta_2) \\ \langle \theta_3, g', \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R4}\quad \frac{\begin{array}{c} b \text{ is a built-in of } g \\ g' := g \setminus b \\ b := x_i = f(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(b, \theta) \\ \theta_2 = \mathtt{UNIF\_FUNC}(b, \theta_1) \\ \theta_3 = \mathtt{EXTG}(b, \theta, \theta_2) \\ \langle \theta_3, g', \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R5}\quad \frac{\begin{array}{c} a \text{ is a literal of } g \\ g' := g \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \neg\mathtt{DELAY}(a, \theta) \\ \theta_1 = \mathtt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \nu\rangle \\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g', \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R6}\quad \frac{\begin{array}{c} a \text{ is a literal of } g \\ g' := g \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \neg\mathtt{DELAY}(a, \theta) \\ \theta_1 = \mathtt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \delta\rangle \\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g', \bar{g} \cup \{a\}\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R7}\quad \frac{\begin{array}{c} a \text{ is a literal of } \bar{g} \\ \bar{g}' := \bar{g} \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \nu\rangle \\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g, \bar{g}'\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle} \qquad \textbf{R8}\quad \frac{\begin{array}{c} a \text{ is a literal of } \bar{g} \\ \bar{g}' := \bar{g} \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \delta\rangle \\ \neg\mathtt{EQUIV}(\theta_1, \theta_2) \\ \theta_3 = \mathtt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R9}\quad \frac{\begin{array}{c} g := a_1, \ldots, a_n \text{ and } \forall a_i : \mathtt{DELAY}(a_i, \theta); \\ \bar{g} := \bar{a}_1, \ldots, \bar{a}_m \text{ and } \forall \bar{a}_j : \\ \left\{\begin{array}{l} \bar{a}_j := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \mathtt{RESTRG}(\bar{a}_j, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \delta\rangle \\ \mathtt{EQUIV}(\theta_1, \theta_2) \end{array}\right. \end{array}}{\langle \theta, g, \bar{g}\rangle \longmapsto \langle \theta, \delta\rangle} \qquad \textbf{R10}\quad \frac{}{\langle \theta, < >, < >\rangle \longmapsto \langle \theta, \nu\rangle}$$

Fig. 2. Concrete Semantic Rules

abstract semantics defined this way. This third step gives the developers the key ideas on how to implement a practical analyser, and can be actually seen as an extension of GAIA [11].

In the rest of the section we describe the main features of these steps.

4.1 Collecting Semantics

In the case of logic programs with delay declarations, the collecting semantics cannot be obtained trivially from the concrete one. Fig. 3 contains its rules, whose main differences can be summarized by:

- an additional deadlock state μ (*may* deadlock) is introduced, therefore in all rules $\kappa \in \{\delta, \nu, \mu\}$;
- an additional transition rule $\mapsto'$ is introduced to compute qualified answers even when the μ deadlock state is reached, i.e., when precision about deadlock is definitely lost. If this is the case, the computation may continue exactly as in the GAIA framework, by completely disregarding the deadlock information, that will be stationary.

Let us try to briefly motivate these features. Rules $\mathbf{R_5}, \mathbf{R_6}$ and $\mathbf{R_9}$ of the concrete semantics require precision when dealing with delay declarations. It is clear that these rules have to be properly extended in order to deal with sets of substitutions whose elements may have opposite behavior with respect to the same declaration. This situation is tackled by rules $\mathcal{R}_{\mathbf{10}}$ and $\mathcal{R}_{\mathbf{11}}$, the only ones that, in fact, may produce and propagate, respectively, the new deadlock state μ. Rule $\mathcal{R}_{\mathbf{10}}$ applies when there are no literals in g which are surely selectable, i.e., there not exist some $a \in g$ such that for all $\theta \in \Theta$: $\neg\mathtt{DELAY}(a, \theta)$, but for some literal a in g (the literals that have not to be reconsidered), there exists θ_1, θ_2 such that $\mathtt{DELAY}(a, \theta_1) \wedge \neg\mathtt{DELAY}(a, \theta_2)$. Rule $\mathcal{R}_{\mathbf{11}}$ behave the same way, but it applies when imprecision arises during the inner computation of a literal. Finally, observe that rules $\mathcal{R}_{10}$ and $\mathcal{R}_{11}$ force the reexecution of the analysis of some literals, in order to improve the precision of the whole analysis. This is achieved through the use of the auxiliary transition rule $\longmapsto'$.

4.2 From the Collecting to the Abstract Semantics

Once the collecting semantics is fixed, deriving abstract semantics is almost an easy job.

- Any domain abstracting substitutions can be used to describe abstract activation states. Similarly to the concrete case, we distinguish among input states and output states. Clearly, the accuracy of deadlock analysis will depend on the matching between delay declarations and the information represented by the abstract domains.
- It is easy to understand, by looking at the collecting semantics defined above, that very few additional operations should be implemented on an abstract substitution domain like the ones in [11,4,5], while a great amount of existing specification and coding can be reused for free.

At each step, in the abstract semantics, only an atom in the goals that surely satisfies the corresponding delay declarations is selected. If the empty

$$\mathcal{R}\mathbf{1} \quad \frac{\begin{array}{c} c \text{ is a clause defining } p \\ \langle \Theta, c \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, p \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{2} \quad \frac{\begin{array}{c} c := h : -g \\ \Theta_1 = \mathtt{EXTC}(c, \Theta) \\ \langle \Theta_1, g, < > \rangle \longmapsto \langle \Theta_2, \kappa \rangle \\ \Theta' = \mathtt{RESTRC}(c, \Theta_2) \end{array}}{\langle \Theta, c \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{3} \quad \frac{\begin{array}{c} b := x_i = x_j \\ g' := g \setminus b \\ \Theta_1 = \mathtt{RESTRG}(b, \Theta) \\ \Theta_2 = \mathtt{UNIF\_VAR}(\Theta_1) \\ \Theta_3 = \mathtt{EXTG}(b, \Theta, \Theta_2) \\ \langle \Theta_3, g', \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{4} \quad \frac{\begin{array}{c} b := x_i = f(x_{i_1}, \ldots, x_{i_n}) \\ g' := g \setminus b \\ \Theta_1 = \mathtt{RESTRG}(b, \Theta) \\ \Theta_2 = \mathtt{UNIF\_FUNC}(b, \Theta_1) \\ \Theta_3 = \mathtt{EXTG}(b, \Theta, \Theta_2) \\ \langle \Theta_3, g', \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{5} \quad \frac{\begin{array}{c} a := p(x_{i_1}, \ldots, x_{i_n}) \\ g' := g \setminus a \\ \forall \theta \in \Theta : \ \neg\mathtt{DELAY}(a, \theta) \\ \Theta_1 = \mathtt{RESTRG}(a, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \nu \rangle \\ \Theta_3 = \mathtt{EXTG}(a, \Theta, \Theta_2) \\ \langle \Theta_3, g', \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{6} \quad \frac{\begin{array}{c} a := p(x_{i_1}, \ldots, x_{i_n}) \\ g' := g \setminus a \\ \forall \theta \in \Theta : \ \neg\mathtt{DELAY}(a, \theta) \\ \Theta_1 = \mathtt{RESTRG}(a, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \kappa' \rangle \\ \kappa' \neq \nu \\ \Theta_3 = \mathtt{EXTG}(a, \Theta, \Theta_2) \\ \langle \Theta_3, g', \bar{g} \cup \{a\} \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{7} \quad \frac{\begin{array}{c} a := p(x_{i_1}, \ldots, x_{i_n}) \\ \bar{g}' := \bar{g} \setminus a \\ \Theta_1 = \mathtt{RESTRG}(a, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \nu \rangle \\ \Theta_3 = \mathtt{EXTG}(a, \Theta, \Theta_2) \\ \langle \Theta_3, g, \bar{g}' \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{8} \quad \frac{\begin{array}{c} a := p(x_{i_1}, \ldots, x_{i_n}) \\ \Theta_1 = \mathtt{RESTRG}(a, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \kappa' \rangle \\ \kappa' \neq \nu \\ \neg\mathtt{EQUIV}(\Theta_1, \Theta_2) \\ \Theta_3 = \mathtt{EXTG}(a, \Theta, \Theta_2) \\ \langle \Theta_3, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \kappa \rangle}$$

$$\mathcal{R}\mathbf{9} \quad \frac{\begin{array}{c} \forall a \in g \text{ and } \forall \theta \in \Theta \text{: } \mathtt{DELAY}(a, \theta) \\ \forall \bar{a} \in \bar{g}\text{: } \left\{ \begin{array}{l} \bar{a} := p(x_{i_1}, \ldots, x_{i_n}) \\ \Theta_1 = \mathtt{RESTRG}(\bar{a}, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \delta \rangle \\ \mathtt{EQUIV}(\Theta_1, \Theta_2) \end{array} \right. \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta, \delta \rangle}$$

$$\mathcal{R}\mathbf{10} \quad \frac{\begin{array}{c} \forall a \in g,\ \exists \theta \in \Theta\text{: } \mathtt{DELAY}(a, \theta) \\ \exists a \in g \text{ and } \theta \in \Theta\text{: } \neg\mathtt{DELAY}(a, \theta) \\ \forall \bar{a} \in \bar{g}\text{: } \left\{ \begin{array}{l} \bar{a} := p(x_{i_1}, \ldots, x_{i_n}) \\ \Theta_1 = \mathtt{RESTRG}(\bar{a}, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \kappa \rangle \\ \kappa \neq \nu \\ \mathtt{EQUIV}(\Theta_1, \Theta_2) \end{array} \right. \\ \langle \Theta, (g, \bar{g}) \rangle \longmapsto' \Theta' \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \mu \rangle}$$

$$\mathcal{R}\mathbf{11} \quad \frac{\begin{array}{c} \forall a \in g,\ \exists \theta \in \Theta\text{: } \mathtt{DELAY}(a, \theta) \\ \forall \bar{a} \in \bar{g}\text{: } \left\{ \begin{array}{l} \bar{a} := p(x_{i_1}, \ldots, x_{i_n}) \\ \Theta_1 = \mathtt{RESTRG}(\bar{a}, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \kappa \rangle \\ \kappa \neq \nu \\ \mathtt{EQUIV}(\Theta_1, \Theta_2) \end{array} \right. \\ \exists \bar{a} \in \bar{g}\text{: } \langle \Theta_1, p \rangle \longmapsto \langle \Theta_2, \mu \rangle \\ \langle \Theta, (g, \bar{g}) \rangle \longmapsto' \Theta' \end{array}}{\langle \Theta, g, \bar{g} \rangle \longmapsto \langle \Theta', \mu \rangle}$$

$$\mathcal{R}\mathbf{12} \quad \frac{}{\langle \Theta, < >, < > \rangle \longmapsto \langle \Theta, \nu \rangle}$$

$$\mathcal{R}13 \quad \frac{\begin{array}{c} c \text{ is a clause defining } p \\ \langle \Theta, c \rangle \longmapsto' \Theta' \end{array}}{\langle \Theta, p \rangle \longmapsto' \Theta'} \qquad \mathcal{R}14 \quad \frac{\begin{array}{c} c := h : -g \\ \Theta_1 = \mathtt{EXTC}(c, \Theta) \\ \langle \Theta_1, g \rangle \longmapsto' \Theta_2 \\ \Theta' = \mathtt{RESTRC}(c, \Theta_2) \end{array}}{\langle \Theta, c \rangle \longmapsto' \Theta'}$$

$$\mathcal{R}15 \quad \frac{}{\langle \Theta, < > \rangle \longmapsto' \Theta} \qquad \mathcal{R}16 \quad \frac{\begin{array}{c} a \text{ is a literal of } g \\ g' := g \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \Theta_1 = \mathtt{RESTRG}(a, \Theta) \\ \langle \Theta_1, p \rangle \longmapsto' \Theta_2 \\ \Theta_3 = \mathtt{EXTG}(a, \Theta, \Theta_2) \\ \langle \Theta_3, g' \rangle \longmapsto' \Theta' \end{array}}{\langle \Theta, g \rangle \longmapsto' \Theta'}$$

Fig. 3. Collecting Semantic Rules

goal is reached, then the interpretation stops and returns an abstraction of the corresponding concrete qualified answers together with *no deadlock* information. Otherwise, a goal of the form $(g, \overline{g})$ is reached, where for all atoms a occurring in g, the activation substitutions (possibly) does not satisfy its delay declaration, and for all atoms a occurring in $\overline{g}$, a reexecution process may not refine the corresponding activation substitution. If the abstract domain is accurate enough to infer that a definite deadlock occurs, then the execution ends and returns an abstraction of concrete qualified answers together with *definite deadlock* information. If this is not the case, then the abstract computation continues by disregarding the deadlock information. In particular, atoms whose activation substitution *may* satisfy the corresponding delay declaration are selected. In this way we improve the computed abstraction of the corresponding concrete qualified answers while we return the information that the concrete computation *may deadlock*.

4.3 The Fixpoint Algorithm

Fig. 4 reports the final step in the Abstract Interpretation picture described above: an algorithm that computes a post-fixpoint of a given abstract semantics that abstracts the collecting one.

As already observed before, most of the operations that are used in the algorithm are simply inherited from the GAIA framework [11]. The only exception is DELAY, the satisfiability test of a delay declaration by an abstract substitution, whose specification as three-value function is as follows: $\mathtt{DELAY}(a, \beta)$ is *true* if for all θ described by β, $\mathtt{DELAY}(a, \theta)$ is *true*; $\mathtt{DELAY}(a, \beta)$ is *false* if for all θ described by β, $\mathtt{DELAY}(a, \theta)$ is *false*; otherwise, $\mathtt{DELAY}(a, \beta)$ is *maybe*.

The operator $T(\beta, g, sat)$ in Fig. 4 is defined exactly as T_b in [11] and corresponds to the auxiliary rules of the collecting semantics defining $\longmapsto'$.

```
$TAB(sat) = \{(\beta, p, \langle\beta', \kappa\rangle) : (\beta, p)$ is an *input state* and $\langle\beta', \kappa\rangle = T_p(\beta, p, sat)\}$.

$T_p(\beta, p, sat) = \texttt{UNION}(\langle\beta_1, \kappa_1\rangle \ldots, \langle\beta_n, \kappa_n\rangle)$
    where  $\langle\beta_i, \kappa_i\rangle = T_c(\beta, c_i, sat)$,
           $c_1, \ldots, c_n$ are the clauses defining $p$.

$T_c(\beta, c, sat) = \langle\texttt{RESTRC}(c, \beta'), \kappa\rangle$
    where  $\langle\beta', \kappa\rangle = T_b(\texttt{EXTC}(c, \beta), b, < >, sat)$,
           $b$ is the body of $c$.

$T_b(\beta, < >, < >, sat) = \langle\beta, \nu\rangle$.

$T_b(\beta, l.g, \bar{g}, sat) = T_b(\beta_3, g, \bar{g}, sat)$          if $l$ is a built-in
    where  $\beta_3 = \texttt{EXTG}(l, \beta, \beta_2)$,
           $\beta_2 = \texttt{UNIF\_VAR}(\beta_1)$            if $l$ is $x_i = x_j$,
                $\texttt{UNIF\_FUNC}(l, \beta_1)$          if $l$ is $x_i = f(\cdots)$,
           $\beta_1 = \texttt{RESTRG}(l, \beta)$.

$T_b(\beta, l.g, \bar{g}, sat) = $  $T_b(\beta_3, g, \bar{g}, sat)$         if $l$ is $p(\cdots)$ and $\texttt{DELAY}(l, \beta) = \mathit{false}$ and $\kappa = \nu$
                      $T_b(\beta_3, g, l.\bar{g}, sat)$        if $l$ is $p(\cdots)$ and $\texttt{DELAY}(l, \beta) = \mathit{false}$ and $\kappa \neq \nu$
    where  $\beta_3 = \texttt{EXTG}(l, \beta, \beta_2)$,
           $\langle\beta_2, \kappa\rangle = sat(\beta_1, p)$,
           $\beta_1 = \texttt{RESTRG}(l, \beta)$.

$T_b(\beta, g, l.\bar{g}, sat) = $  $T_b(\beta_3, g, \bar{g}, sat)$         if $l$ is $p(\cdots)$ and $\texttt{DELAY}(l, \beta) \in \{\mathit{true, maybe}\}$ and $\kappa = \nu$
                      $T_b(\beta_3, g, l.\bar{g}, sat)$        if $l$ is $p(\cdots)$ and $\texttt{DELAY}(l, \beta) \in \{\mathit{true, maybe}\}$ and $\kappa \neq \nu, \beta_1 \neq \beta_2$
    where  $\beta_3 = \texttt{EXTG}(l, \beta, \beta_2)$,
           $\langle\beta_2, \kappa\rangle = sat(\beta_1, p)$
           $\beta_1 = \texttt{RESTRG}(l, \beta)$.

$T_b(\beta, g, \bar{g}, sat) = $    $\langle\beta, \delta\rangle$                       if $\texttt{SUSPEND}(\beta, g, \bar{g})$
                      $\langle T(\beta, (g.\bar{g}), sat), \mu\rangle$    otherwise.

$\texttt{SUSPEND}(\beta, g, \bar{g}) = $  $\mathit{true}$                 if $\forall a \in g$: $\texttt{DELAY}(a, \beta)$
                                      and $\forall \bar{a} \in \bar{g}$: $\beta_1 = \beta_2$
                                      where  $l$ is $p(\cdots)$,
                                             $\langle\beta_2, \delta\rangle = sat(\beta_1, p)$,
                                             $\beta_1 = \texttt{RESTRG}(a, \beta)$,
                        $\mathit{false}$                if none of the above cases applies.
```

Fig. 4. The abstract transformation

The correctness of the algorithm in Fig. 4 can be proven the same way as in [11] and [12]. What about termination ? We may observe that in the collecting semantics the reexecution rule $\mathcal{R}_\mathbf{8}$ may introduce infinite loops. However, this problem does not arise in the abstract semantics as

1. the abstract domain is required to be a complete lattice (when this is not the case, and it is just a cpo, an additional widening operation is usually provided by the domain),
2. the derivation $\longmapsto$ produces a decreasing chain in its first argument.

Example 4.1 Consider the program PERMUTE discussed by Naish in [16].

```
%  perm(Xs,Ys) ← Ys is a permutation of the lists Xs
   perm(Xs,Ys) ← Xs = [ ],
      Ys = [ ].
   perm(Xs,Ys) ← Xs = [X|X1s],
      delete(X,Ys,Zs),
       perm(X1s,Zs).

%   delete(X,Ys,Zs)  ← Zs  is the list obtained by removing  X  from the list  Ys
    delete(X,Ys,Zs)  ← Ys = [X|Zs].
    delete(X,Ys,Zs)  ← Ys = [X|Y1s],
       Zs = [X|Z1s],
       delete(X,Y1s,Z1s).
```

Clearly, the relation declaratively given by perm is symmetric. Unfortunately, the behavior of the program with Prolog (using the leftmost selection rule) is not. In fact, given the query

$$Q_1 := \leftarrow \texttt{perm}(\texttt{Xs}, [\mathsf{a}, \mathsf{b}]).$$

Prolog will correctly backtrack through the answers $\texttt{Xs} = [\mathsf{a}, \mathsf{b}]$ and $\texttt{Xs} = [\mathsf{b}, \mathsf{a}]$. However, for the query

$$Q_2 := \leftarrow \texttt{perm}([\mathsf{a}, \mathsf{b}], \texttt{Xs}).$$

Prolog will first return the answer $\texttt{Xs} = [\mathsf{a}, \mathsf{b}]$ and on subsequent backtracking will fall into an infinite derivation without returning answers anymore.

For languages with delay declarations the program PERMUTE behaves symmetrically. In particular, if we consider the delay declarations:

```
    DELAY perm(Xs,_) UNTIL nonvar(Xs).
    DELAY delete(_,_,Zs) UNTIL nonvar(Zs).
```

the query Q_2 above does not fall into a deadlock.

Using one of our domains for abstract susbtitutions, like Prop (see [3,17]), and starting from an activation state of the form perm(ground,var) our analysis returns the abstract qualified answer $\langle \texttt{perm}(\texttt{ground}, \texttt{ground}), \nu \rangle$. which provides the information that any corresponding concrete execution is deadlock free.

5 Conclusions

The semantics that has been discussed in these pages belongs to the foundation part of a project aimed at integrating most of the work (both theoretical and practical) on abstract interpretation of logic programs developed by the authors in the last years. The goal is to get a practical tool that tackles a variety of problems raised by the recent research and development directions in declarative programming. Dynamic scheduling is an interesting example in that respect. In the next future, we plan to adapt the existing implementations of GAIA systems in order to practically evaluate the accuracy and efficiency of these seminal ideas.

References

[1] Apt, K. R., "From Logic Programming to Prolog," Prentice Hall, 1997.

[2] Apt, K. R. and I. Luitjes, "Verification of Logic Programs with Delay Declarations," Lecture Notes in Computer Science **936**, Springer-Verlag, New York, 1995 pp. 66–80.

[3] Cortesi, A., G. Filé and W. Winsborough, *Optimal groundness analysis using propositional logic*, Journal of Logic Programming **27** (1996), pp. 137–167.

[4] Cortesi, A., B. Le Charlier and P. Van Hentenryck, *Combination of abstract domains for logic programming*, in: *Proceedings of the 21th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)* (1994), pp. 7–21.

[5] Cortesi, A., B. Le Charlier and P. Van Hentenryck, *Combination of abstract domains for logic programming: open product and generic pattern construction*, Science of Computer Programming **28** (2000), pp. 27–71.

[6] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)* (1977), pp. 238–252.

[7] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *Conference Record of Sixth ACM Symposium on Programming Languages (POPL'79)* (1979), pp. 269–282.

[8] Falaschi, M., M. Gabbrielli, K. Marriott and C. Palamidessi, *Constraint logic programming with dynamic scheduling: A semantics based on closure operators*, Information and Computation **137** (1997), pp. 41–67.

[9] Garcia de la Banda, M., K. Marriott and P. Stuckey, *Efficient analysis of logic programs with dynamic scheduling*, in: J. Lloyd, editor, *Proceedings of the 12st International Logic Programming Symposium* (1995), pp. 417–431.

[10] Intelligent Systems Laboratory, Swedish Institute of Computer Science, Kista, Sweden, "SICStus Prolog User's Manual," (1998).
URL `http://www.sics.se/isl/sicstus/sicstus_toc.html`

[11] Le Charlier, B. and P. Van Hentenryck, *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, ACM Transactions on Programming Languages and Systems (TOPLAS) **16** (1994), pp. 35–101.

[12] Le Charlier, B. and P. Van Hentenryck, *Reexecution in abstract interpretation of Prolog*, Acta Informatica **32** (1995), pp. 209–253.

[13] Lloyd, J., "Foundations of Logic Programming," Symbolic Computation – Artificial Intelligence, Springer-Verlag, 1987, second, extended edition.

[14] Marchiori, E. and F. Teusink, *Proving termination of logic programs with delay declarations*, Journal of Logic Programming **39** (1999), pp. 95–124.

[15] Marriott, K., M. Garcia de la Banda and M. Hermenegildo, *Analyzing logic programs with dynamic scheduling*, in: *Proceedings of the 21st Annual ACM Symp. on Principles of Programming Languages* (1994), pp. 240–253.

[16] Naish, L., "Negation and control in Prolog," Lecture Notes in Computer Science **238**, Springer-Verlag, New York, 1986.

[17] Van Hentenryck, P., A. Cortesi and B. Le Charlier, *Evaluation of the domain Prop*, Journal of Logic Programming **23** (1995), pp. 237–278.

[18] Yelick, K. and J. Zachary, *Moded type systems for logic programming*, in: *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89)* (1989), pp. 116–124.