Reexecution-Based Analysis of Logic Programs with Delay Declarations

Agostino Cortesi[1], Baudouin Le Charlier[2], and Sabina Rossi[1]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia,
via Torino 155, 30172 Venezia, Italy
{cortesi,srossi}@dsi.unive.it

[2] Universite Catholique de Louvain, Departement d'ingegnierie Informatique
2, Place Sainte-Barbe, B-1348 Louvain-la-Neuve (Belgique)
blc@info.ucl.ac.be

Abstract. A general semantics-based framework for the analysis of logic programs with delay declarations is presented. The framework incorporates well known refinement techniques based on reexecution. The concrete and abstract semantics express both deadlock information and qualified answers.

1 Introduction

In order to get more efficiency, users of current logic programming environments, like Sictus-Prolog [13], Prolog-III, CHIP, SEPIA, etc., are not forced to use the classical Prolog left-to-right scheduling rule. Dynamic scheduling can be applied instead where 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 by Marriott *et al.* in [18,11]. 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 [12], while verification and termination issues have been investigated by Apt and Luitjes in [2] and by Marchiori and Teusink in [17], 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 [14]: this allows us to easily derive correctness proofs of the whole analyses;
- to define the analysis as an extension of the framework depicted in [14]: this allows us to reuse existing code for program analysis, with minimal additional effort;
- to explicitly derive deadlock information (possible deadlock and deadlock freeness) producing, as a result of the analysis, an approximation of concrete qualified answers;

D. Bjørner, M. Broy, and A. Zamulin (Eds.): PSI 2001, LNCS 2244, pp. 395–405, 2001.

- to apply the reexecution technique developed in [15]: if during the execution of an atom a a deadlock occurs, then a is allowed to be reexecuted at a subsequent step.

The main difference between our approach and the ones already presented in the literature is that we are mainly focussed on analysis issues, in particular on deadlock and no-deadlock analysis. This motivates the choice of a strictly operational approach, where deadlock information is explicitly maintained.

In this paper we present an extension of the specification of the GAIA abstract interpreter [14] to deal with dynamic scheduling. We design both a concrete and an abstract semantics, as well as a a generic algorithm that computes a fixpoint of the abstract semantics. This is done following the classical abstract interpretation methodology.

The main idea is partitioning literals of a goal g into three sets: literals which are delayed, literals which are not delayed and have not been executed yet, and literals which are allowed to be reexecuted as they are not delayed but have already been executed before and fallen into deadlock. This partitioning dramatically simplifies both concrete and abstract semantics with respect to the approach depicted in [8], where a preliminary version of this work was presented.

Our approach uses the reexecution technique which exploits the well known property of logic programming that a goal may be reexecuted arbitrarily often without affecting the semantics of the program. This property has been pointed out since 1987 by Bruynooghe [3,4] and subsequently used in abstract interpretation to improve the precision of the analysis [15]. In this framework, reexecution allows to improve the accuracy of deadlock analysis, and its application may be tuned according to computational constraints.

The rest of the paper is organized as follows. In the next section we recall some basic notions about logic programs with delay declarations. Section 3 depicts the concrete operational semantics which serves as a basis for the new abstract semantics introduced in Section 4. Correctness of our generic fixpoint algorithm is discussed. 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 of a program augmented with delay declarations *succeeds* if it ends with the empty goal; while it *deadlocks* if it ends with a non-empty goal

no atom of which satisfies its delay declaration. Both successful and deadlocked derivations compute *qualified answers*, i.e., pairs of the form $\langle \theta, d \rangle$ where d is the last goal (that is a possibly empty sequence of delayed atoms) and θ is the substitution obtained by concatenating the computed mgu's from the initial goal. Notice that, if $\langle \theta, d \rangle$ is a qualified answer for a successful derivation then d is the empty goal and θ restricted to the variables of the initial goal is the corresponding 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.

The following example illustrates the use of delay declarations in logic programming.

Example 1. Consider the program `PERMUTE` discussed by Naish in [19].

```
%  perm(Xs,Ys)  ← Ys is a permutation of the list 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 = [X1|Y1s], Zs = [X1|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

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

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

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

Prolog will first return the answer $\texttt{Xs} = [\texttt{a}, \texttt{b}]$ 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. ∎

Under the assumption that delay declarations are closed under instantiation, the following result, which is a variant of Theorem 4 in Yelick and Zachary [21], holds.

$$
\begin{array}{lll}
P \in \mathit{Programs} & P & ::= pr_1, \ldots, pr_n \ (n > 0) \\
pr \in \mathit{Procedures} & pr & ::= c_1, \ldots, c_n \ (n > 0) \\
c \in \mathit{Clauses} & c & ::= h : -g. \\
h \in \mathit{ClauseHeads} & h & ::= p(x_1, \ldots, x_n) \ (n \geq 0) \\
g \in \mathit{LiteralSequences} & g & ::= l_1, \ldots, l_n \ (n \geq 0) \\
l \in \mathit{Literals} & l & ::= a \mid b \\
a \in \mathit{Atoms} & a & ::= p(x_{i_1}, \ldots, x_{i_n}) \ (n \geq 0) \\
b \in \mathit{Built\text{-}ins} & b & ::= x_i = x_j \mid x_{i_1} = f(x_{i_2}, \ldots, x_{i_n}) \\
p \in \mathit{ProcedureNames} & & \\
f \in \mathit{Functors} & & \\
x_i \in \mathit{ProgramVariables} & &
\end{array}
$$

Fig. 1. Abstract syntax of normalized programs

Theorem 1. *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 1 allows us to treat goals as multisets instead of sequences of atoms.

3 The Concrete Operational Semantics

In this section we describe a concrete operational 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, [1,16]).

3.1 Programs and Substitutions

Programs are assumed to be normalized according to the 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 $x_1, \ldots, x_m$. If $g := a_1, \ldots, a_n$ we denote by $g \setminus a_i$ the goal $g' := a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

We assume the existence of two disjoint and infinite sets of variables: *program variables*, which are ordered and denoted by x_1, x_2, ..., x_i, ..., and *standard variables* which are denoted by letters y and z (possibly subscripted). Programs are built using program variables only.

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 (built with standard variables only). 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. 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 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 set of program variables. We denote by PS_D the set of program substitutions whose domain is D.

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., it is an element from the set $\{\delta, \nu\}$, where δ stands for *definite deadlock*, while ν stands for *no deadlock*. In case of no deadlock, 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 deadlock, 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 [15]. Concrete behaviors are intended to model successful and deadlocked derivations of atomic queries.

3.3 Concrete Semantic 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 semantic rules that naturally extend concrete behaviors in order to deal with clauses and goals. In particular, a concrete behavior is extended through intermediate states of the form $\langle \theta, c \rangle$ and $\langle \theta, g\_d, g\_e, g\_r \rangle$, where c is a clause and g_d, g_e, g_r is a partition of a goal g such that: g_d contains all literals in g which are delayed, g_e contains all literals in g which are not delayed and have not been executed yet, g_r contains all literals in g which are allowed to be reexecuted, i.e., all literals that are not delayed and have already been executed but fallen into a deadlock.

– Each pair $\langle \theta, c \rangle$, where c is a clause, $\theta \in PS_{\{x_1,\ldots,x_n\}}$ and $x_1, \ldots, x_n$ are the variables occurring in the head of c, is related to an output state $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_n\}}$ and $\kappa \in \{\delta, \nu\}$ is a deadlock state;

– Each tuple $\langle \theta, g\_d, g\_e, g\_r \rangle$, where $\theta \in PS_{\{x_1,\ldots,x_m\}}$ and $x_1, \ldots, x_m$ are the variables occurring in (g_d, g_e, g_r), is related to an output state $\langle \theta', \kappa \rangle$, where $\theta' \in PS_{\{x_1,\ldots,x_m\}}$ 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 semantic rules depicted in Fig. 2. The reader may refer to [14] for a complete description of all operations but the last one, `SPLIT`, that is 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 $\{x_{i_1}, \ldots, x_{i_n}\}$ be the set of variables occurring in l. This operation expresses a substitution on variables $x_{i_1}, \ldots, x_{i_n}$ in terms of the formal parameters $x_1, \ldots, x_n$.
- `EXTG` is used to combine the analysis of a built-in or a procedure call (expressed in terms of the formal parameters $x_1, \ldots, x_n$) 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 $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$, respectively.
- `SPLIT` is a new operation: given a substitution θ and a goal g, it partitions g into the set of atoms g_d which do not satisfy the corresponding delay declarations, and then are not executable, and the set of atoms g_e which satisfy the corresponding delay declarations, and then are executable.

The definition of the concrete semantic rules proceeds by induction on the syntactic structure of program P. 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 input substitution after splitting the body into two parts: executable literals and delayed literals. Rule $\mathbf{R}_3$ defines the result of executing the empty goal, generating a successful output substitution. Rule $\mathbf{R}_4$ defines a deadlock situation that yields a definite deadlock information δ. Rules $\mathbf{R}_5$ to $\mathbf{R}_8$ specify the execution of a literal. First, the literal is executed producing an output substitution θ_3; then reexecutable atoms are (re)executed through the auxiliary relation $\langle \theta_3, g\_r \rangle \longmapsto_r \langle \theta_4, \bar{g}\_r \rangle$: its effect is to refine θ_3 into θ_4 and to remove from g_r the atoms that are completely solved in θ_4 returning the new list of reexecutable atoms $\bar{g}\_r$; finally, the sequence of delayed atoms with the new substitution θ_4 is partitioned in two sets: the atoms that are still delayed and those that have been awakened. Rules $\mathbf{R}_5$ and $\mathbf{R}_6$ specify the execution of built-ins and use the unification operations. Rules $\mathbf{R}_7$ and $\mathbf{R}_8$ define the execution of an atom a

The reexecutable rules defining the auxiliary relation $\longmapsto_r$ can be easily obtained according to the methodology in [15].

The concrete semantics of a program P with delay declarations is defined as a fixpoint of this transition system. We can prove that this operational semantics is safe with respect to the standard resolution of programs with delay declarations.

$$\textbf{R1}\ \frac{c \text{ is a clause defining } p \quad \langle \theta, c\rangle \longmapsto \langle \theta', \kappa\rangle}{\langle \theta, p\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R2}\ \frac{\begin{array}{c} c := h :- g \\ \theta_1 = \texttt{EXTC}(c, \theta) \\ \langle g\_d, g\_e\rangle = \texttt{SPLIT}(\theta_1, g) \\ \langle \theta_1, g\_d, g\_e, <>\rangle \longmapsto \langle \theta_2, \kappa\rangle \\ \theta' = \texttt{RESTRC}(c, \theta_2) \end{array}}{\langle \theta, c\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R3}\ \frac{}{\langle \theta, <>, <><>, \rangle \longmapsto \langle \theta, \nu\rangle}$$

$$\textbf{R4}\ \frac{\text{either } g\_d \neq <> \text{ or } g\_r \neq <>}{\langle \theta, g\_d, <>, g\_r\rangle \longmapsto \langle \theta, \delta\rangle}$$

$$\textbf{R5}\ \frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus b \\ b := x_i = x_j \\ \theta_1 = \texttt{RESTRG}(b, \theta) \\ \theta_2 = \texttt{UNIF\_VAR}(\theta_1) \\ \theta_3 = \texttt{EXTG}(b, \theta, \theta_2) \\ \langle \theta_3, g\_r\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \texttt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R6}\ \frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus b \\ b := x_i = f(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \texttt{RESTRG}(b, \theta) \\ \theta_2 = \texttt{UNIF\_FUNC}(b, \theta_1) \\ \theta_3 = \texttt{EXTG}(b, \theta, \theta_2) \\ \langle \theta_3, g\_r\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \texttt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R7}\ \frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \texttt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \nu\rangle \\ \theta_3 = \texttt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g\_r\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \texttt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle}$$

$$\textbf{R8}\ \frac{\begin{array}{c} \bar{g}\_e := g\_e \setminus a \\ a := p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 = \texttt{RESTRG}(a, \theta) \\ \langle \theta_1, p\rangle \longmapsto \langle \theta_2, \delta\rangle \\ \theta_3 = \texttt{EXTG}(a, \theta, \theta_2) \\ \langle \theta_3, g\_r.a\rangle \longmapsto_r \langle \theta_4, \bar{g}\_r\rangle \\ \langle \bar{g}\_d, \bar{g}'\_e\rangle = \texttt{SPLIT}(\theta_4, g\_d) \\ \langle \theta_4, \bar{g}\_d, \bar{g}\_e \cup \bar{g}'\_e, \bar{g}_r\rangle \longmapsto \langle \theta', \kappa\rangle \end{array}}{\langle \theta, g\_d, g\_e, g\_r\rangle \longmapsto \langle \theta', \kappa\rangle}$$

Fig. 2. Concrete semantic rules

4 Collecting and Abstract Semantics

As usual in the *Abstract Interpretation* approach [9,10], in order to define an abstract semantics 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 $\wp(Subst)$, and a suite of abstract operations that safely approximate the concrete ones. Finally, we draw an algorithm to compute a (post-)fixpoint of an abstract semantics defined this way.

The collecting semantics can be trivially obtained from the concrete one by

- replacing substitutions with sets of substitutions;
- using μ, standing for *possible deadlock*, instead of δ;
- redefining all operations in order to deal with sets of substitutions (as done in [14]).

In particular, the collecting version of operation `SPLIT`, given a set of substitutions Θ, will partition a goal g into the set of atoms g_d which do not satisfy the corresponding delay declarations for some $\theta \in \Theta$, and the set of atoms g_e which do satisfy the corresponding delay declarations for some $\theta \in \Theta$. Notice that this approach is sound, i.e., if an atom is executed at the concrete level then it will be also at the abstract level. However, since some atoms can be put both in g_d and in g_e some level of imprecision could arise.

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, e.g., $\langle \beta, p \rangle$ where β is an approximation of a set of activation substitutions, and output states, e.g., $\langle \beta', \kappa \rangle$ where β' is an approximation of a set of output substitutions and $\kappa \in \{\mu, \nu\}$ is an abstract deadlock state. 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 concrete semantics presented above, that very few additional operations should be implemented on an abstract substitution domain like the ones in [6,7,14], while a great amount of existing specification and coding can be reused for free.

Fig. 3 reports the final step in the Abstract Interpretation picture described above: an abstract transformation that abstracts the concrete semantics rules. The abstract semantics is defined as a post-fixpoint of transformation *TAB* on sets of abstract tuples, *sat*, as defined in the picture. An algorithm computing the abstract semantics can be defined by simple modification of the reexecution fixpoint algorithm presented in [15]. The reexecution function T_r is in the spirit of [15]. It uses the abstract operations `REFINE` and `RENAME`, where

- `REFINE` is used to refine the result β of executing an atom by combining it with the results obtained by reexecution of atoms in the reexecutable atom lists starting from β itself;
- `RENAME` is used after reexecution of an atom a: it expresses the result of reexecution in terms of the variables $x_{i_1}, \ldots, x_{i_n}$ occurring in a.

$TAB(sat) = \{(\beta, p, \langle\beta', \kappa\rangle) : (\beta, p)$ 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 $\langle\beta', \kappa\rangle = T_b(\texttt{EXTC}(c, \beta), g\_d, g\_e, <>, sat)$,
$\langle g\_d, g\_e\rangle = \texttt{SPLIT}(\beta, b)$ where b is the body of c.

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

$T_b(\beta, g\_d, <>, g\_r, sat) = \langle\beta, \mu\rangle$
where either g_d or g_r is not empty.

$T_b(\beta, g\_d, l.g\_e, g\_r, sat) = T_b(\beta_4, \bar{g}\_d, g\_e.\bar{g}\_e, \bar{g}\_r, sat)$
where
$\langle\bar{g}\_d, \bar{g}\_e\rangle = \texttt{SPLIT}(\beta_4, g\_d)$
$\langle\beta_4, \bar{g}\_r\rangle = T_r(\beta_3, g\_r, sat)$ if $\kappa = \nu$,
$T_r(\beta_3, g\_r.l, sat)$ if $\kappa = \mu$,
$\beta_3 = \texttt{EXTG}(l, \beta, \beta_2)$,
$\langle\beta_2, \kappa\rangle = sat(\beta_1, p)$ if l is $p(\cdots)$
$\langle\texttt{UNIF\_VAR}(\beta_1), \nu\rangle$ if l is $x_i = x_j$,
$\langle\texttt{UNIF\_FUNC}(l, \beta_1), \nu\rangle$ if l is $x_i = f(\cdots)$,
$\beta_1 = \texttt{RESTRG}(l, \beta)$.

$T_r(\beta, (a_1, \ldots, a_n), sat) = \sqcap_{i=1}^{\infty}\langle\beta_i, g_i\rangle$
where $\langle\beta_0, g_0\rangle = \langle\beta, (a_1, \ldots, a_n)\rangle$
$\beta_{i+1} = \texttt{REFINE}(\beta_i, T_r(\beta_i, a_1, sat), \ldots, T_r(\beta_i, a_n, sat))$ $(i \geq 1)$
$g_{i+1} = \{a_i \mid i \in \{1, \ldots, n\}$ and $\langle\bullet, \mu\rangle = T_r(\beta_i, a_i, sat)\}$

$T_r(\beta, a, sat) = \langle\texttt{RENAME}(a, \beta_2), \kappa\rangle$
where $\langle\beta_2, k\rangle = sat(\beta_1, p)$ if a is $p(\cdots)$
$\beta_1 = \texttt{RESTRG}(a, \beta)$.

Fig. 3. The abstract transformation

As already observed before, most of the operations that are used in the algorithm are simply inherited from the GAIA framework [14]. The only exception is SPLIT, which depends on a given set of delay declarations.

The correctness of the algorithm can be proven the same way as in [14] and [15]. What about termination ? The execution of T_b terminates since the number of literals in g_d and g_e decreases of exactly one at each recursive call. The fact that the execution of T_r terminates depends on some hypothesis on the abstract domain such as 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).

Example 2. Consider again the program `PERMUTE` illustrated above. Using one of our domains for abstract substitutions, like `Pattern` (see [5,20]), 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 concrete execution, starting in a query of `perm` with the first argument being ground and the second one being variable, is deadlock free.

5 Conclusions

The framework presented in this paper is 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 final 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, as most of current logic programming environments integrate this feature.

In the next future, we plan to adapt the existing implementations of GAIA systems in order to practically evaluate the accuracy and efficiency of the this framework.

Acknowledgments. This work has been partially supported by the Italian MURST Projects "Interpretazione Astratta, Type Systems e Analisi Control-Flow", and "Certificazione automatica di programmi mediante interpretazione astratta".

References

1. K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. *Lecture Notes in Computer Science*, 936:66–80, 1995.
3. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.
4. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, California, August 1987. Computer Society Press of the IEEE.
5. A. Cortesi, G. Filé, and W. Winsborough. Optimal groundness analysis using propositional logic. *Journal of Logic Programming*, 27(2):137–167, May 1996.
6. A. Cortesi, 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)*, Portland, Oregon, January 1994.
7. A. Cortesi, 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(1–3):27–71, 2000.
8. A. Cortesi, S. Rossi, and B. Le Charlier. Operational semantics for reexecution-based analysis of logic programs with delay declarations. *Electronic Notes in Theoretical Computer Science*, 48(1), 2001. http://www.elsevier.nl/locate/entcs.

9. P. Cousot 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)*, pages 238–252, Los Angeles, California, January 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of Sixth ACM Symposium on Programming Languages (POPL'79)*, pages 269–282, Los Angeles, California, January 1979.
11. M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*, pages 417–431. MIT Press, 1995.
12. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: A semantics based on closure operators. *Information and Computation*, 137(1):41–67, 1997.
13. Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1998. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.
14. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.
15. B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.
16. J.W. Lloyd. *Foundations of Logic Programming*. Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Verlag, second, extended edition, 1987.
17. E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 447–464, Cambridge, December 4–7 1995. MIT Press.
18. K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 240–253. ACM Press, 1994.
19. L. Naish. *Negation and control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.
20. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the domain *Prop*. *Journal of Logic Programming*, 23(3):237–278, June 1995.
21. K. Yelick and J. Zachary. Moded type systems for logic programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 116–124, 1989.