Constraint Logic Programming with Dynamic Scheduling: A Semantics Based on Closure Operators *

Moreno Falaschi[§]
Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze, 206 33100 Udine, Italy
falaschi@dimi.uniud.it

Maurizio Gabbrielli
Dipartimento di Informatica, Università di Pisa, Corso Italia, 40 56125 Pisa, Italy
gabbri@di.unipi.it

Kim Marriott
Department of Computer Science, Monash University, Clayton 3168, Victoria, Australia
marriott@cs.monash.edu.au

Catuscia Palamidessi[¶]
DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy
catuscia@disi.unige.it

Abstract

The first logic programming languages, such as Prolog, used a fixed left-to-right atom scheduling rule. Recent logic programming languages, however, provide more flexible scheduling in which there is a default computation rule such as left-to-right but in which some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. Such languages include constraint logic programming languages, since most implementations of these languages delay constraints which are "too hard."

From the semantic point of view, the fact that an atom must be delayed under certain conditions, causes the standard semantics of (constraint) logic programming to be no longer adequate to capture the meaning of a program. In our paper we attack this problem and we develop a denotational semantics for constraint logic programming with dynamic scheduling. The key idea is that the denotation of an atom or goal is a set of closure operators, where different closure operators correspond to different sequences of rule choices.

Keywords: Logic programming, dynamic scheduling, denotational semantics.

1 Introduction

The first logic programming languages, such as DEC-10 Prolog, used a fixed scheduling rule in which all atoms in the goal were processed left-to-right. Unfortunately, this meant that programs written in a clean, declarative style might be inefficient, might only terminate when certain inputs were fully instantiated, and might produce wrong results if used with negation. For this reason, nearly all recent logic programming languages provide more flexible scheduling in which computation generally proceeds left-to-right but in which some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run efficiently. Most constraint logic programming languages [Jaffar and Lassez 1987] also employ dynamic scheduling. If a constraint is "too hard" for the solver, then it is delayed until it becomes simpler. For example, in CLP($\mathcal{R}$) non-linear arithmetic constraints are delayed until they become linear.

Despite the practical importance of logic languages with dynamic scheduling, there has been surprisingly little work devoted to their semantics. Their operational semantics have been discussed by Naish [1986]. Yellick and Zachary [1989] and Naish [1992] show that, under certain restrictions, the operational semantics is confluent in the sense that different atom schedulings give rise to the same possible outcomes. The only denotational

---

*To appear in *Information and Computation*.
[§]Partially supported by the HCM project CONSOLE.
[¶]Partially supported by the HCM project EXPRESS.

semantics that we are aware of for logic languages with delay is that of Marriott *et al* [1994], which is quite complex since it is very close to the operational semantics and explicitly models the delayed atoms. Denotational semantics for logic programs and constraint logic programs with left-to-right atom selection are not easily modified to deal with logic languages with dynamic scheduling. One reason is that they cannot capture the possibility of a goal "floundering" in the sense that no atom in the goal can be selected for reduction.

In this paper we develop a denotational semantics, much simpler than the one proposed in [Marriott et al. 1994], for constraint logic programming with dynamic scheduling. The key idea is to use closure operators, following the approach of Saraswat *et al* [1991] for deterministic concurrent constraint programming (ccp). In fact, a delay condition can be seen as an *ask* operation, and a constraint atom can be seen as a *tell* operation, hence deterministic clp programs can be considered a particular case of deterministic ccp programs and therefore interpreted as monotonic, idempotent and increasing functions from initial constraints to answer constraints (i.e. closure operators on the constraint domain). However, our language is nondeterministic, so we must take a more sophisticated approach. Fortunately the kind of choice we are dealing with is "local" (or "angelic", in the sense of Jagadeesan *et al* [1991]), and it is therefore more easy to treat than the "global choice" of nondeterministic ccp [Saraswat 1989; Saraswat et al. 1991]. Jagadeesan *et al* [1991] showed that, if one is interested only in observing the upward closure of the computed answers, then the denotational semantics of local (angelic) ccp can still be constructed, by using closure operators, in a very simple way. However, the price to pay for the simplicity of this approach, is that a program like

$$\begin{array}{lll} \mathtt{p(X)} & \leftarrow & \mathtt{X = a.} \\ \mathtt{p(X)} & \leftarrow & \mathtt{true.} \end{array}$$

is identified with the program

$$\mathtt{p(X)} \quad \leftarrow \quad \mathtt{true.}$$

Identifying the two programs above is particularly undesirable in the case of logic programming with dynamic scheduling. In fact, if we add a declaration for a predicate q, of the form

$$\mathtt{q(X,Y)} \quad \leftarrow \quad \mathtt{Y = X\ when\ ground(X).}$$

(which means that the evaluation of $q(X, Y)$ must be delayed until X is ground), then the goal

$$? - \mathtt{p(X), q(X,Y)}$$

will flounder in the second program, while it computes the answer $X = a$, $Y = a$ in the first program.

In our paper we deal with the problem of developing a semantics based on closure operators which is still simple, yet able to capture the *exact* computed answers, i.e. to distinguish the two programs above. We construct such a semantics by using *sets* of closure operators. Intuitively, the fact that we can use sets instead of more complicated structures (like trees) corresponds to the intuition that local choices do not depend on the current environment. Hence the computation tree (as far as the input-output relation is concerned) can be represented by the set of computational branches, and therefore by a set of closure operators, since every branch represents a deterministic computation.

A natural question at this point is whether our semantics could be adequate also for full ccp. The answer is negative, because in the presence of global choice the branching structure of the tree becomes relevant. This is well-known from process algebra theory, and the standard counterexamples apply to our case too. In literature, there have been two equivalent (independent) proposals [de Boer and Palamidessi 1991; Saraswat et al. 1991] for the denotational semantics of nondeterministic ccp. These are based on considering all the potential resting points of a process, and by attaching, to each of them, the reactive sequences of the processes, i.e. its possible interaction with an hypothetical environment, up to that resting point. Full abstraction is achieved by *saturation*, i.e. by adding all sequences corresponding to "lazier" behaviors. The description of this semantics in [Saraswat et al. 1991] is actually based on closure operators, as the reactive sequences can be represented as a special kind of such functions, called *trace operators*. It is worth noting that the semantics in [de Boer and Palamidessi 1991; Saraswat et al. 1991] would remain fully abstract when restricted to local choice ccp, because the full abstraction proof is based on a distinguishing context which is a deterministic process. However, we would like to argue that it is more complicated than our semantics, as its representation involves more complex (higher-order) structures: sets of closure operators on subdomains (determined by the potential resting points). Let us consider, for instance, the process $tell(c)$. In our case the semantics will be a singleton set containing the closure operator corresponding to the input-output semantics of $tell(c)$, i.e. a function associating to each constraint d the join of d with c. In their case, it will be the union of all the sets $\{f \mid f \leq f_d\}_{d \geq c}$, where f_d is the closure operator corresponding to $tell(c)$ in the subdomain $\downarrow d$ (downward closure of d), and $\leq$ is the usual pointwise ordering induced on functions. Furthermore let us remark that in general, even when the property of full abstraction is preserved when restricting to a sublanguage, it is often the case that the semantics structures studied for the superlanguage become redundant and inadequate to understand the properties of the sublanguage, and that a simpler model exists. For instance, consider again the fully abstract semantics for (full) ccp in [de Boer and Palamidessi 1991; Saraswat et al. 1991]. This is also fully abstract for the deterministic subset of ccp, since as we mentioned above the context used in the full abstraction proof is deterministic. However, for deterministic ccp a much simpler (fully abstract) semantics exists, which is the semantics that associates to each process the closure operator representing the input-output semantics of that process [Saraswat et al. 1991]. Note that this latter model captures for instance the property of processes being deterministic (a process is a closure operator, hence a function), while this is not visible in the fully abstract semantics inherited by the fully abstract semantics of ccp.

The plan of the paper is as follows: In the next section we give some examples to illustrate the usefulness of dynamic scheduling and the fact that the standard semantics of clp is inadequate to capture the behavior of programs with dynamic scheduling. In Section 3 we define the operational semantics of constraint logic programming with dynamic scheduling. In Section 4 we show a semantics based on and-trees which will act as a bridge between the operational semantics and the denotational semantics. In Section 5 we develop the denotational semantics based on closure operators. In Section 5.1 we discuss full abstraction. Finally, in Section 6 we summarize our results.

2 Examples

The following program adapted from Naish [1986], illustrates the power of dynamic scheduling. The program `permute` is a definition of the relation "to be a permutation of". It makes use of the procedure `delete(X,Y,Z)` which holds if Z is the list obtained by removing X from the list Y (uppercase letters denote variables and ":" denotes list concatenation).

$$
\begin{array}{lcl}
\mathtt{permute(X,Y)} & \leftarrow & \mathtt{X = nil,} \\
 & & \mathtt{Y = nil.} \\
\mathtt{permute(X,Y)} & \leftarrow & \mathtt{X = U : X1,} \\
 & & \mathtt{delete(U,Y,Z),} \\
 & & \mathtt{permute(X1,Z).} \\
\mathtt{delete(X,Y,Z)} & \leftarrow & \mathtt{Y = X : Z.} \\
\mathtt{delete(X,Y,Z)} & \leftarrow & \mathtt{Y = U : Y1,} \\
 & & \mathtt{Z = U : Z1,} \\
 & & \mathtt{delete(X,Y1,Z1).}
\end{array}
$$

Clearly the relation declaratively given by `permute` is symmetric. Unfortunately, the behavior of the program with traditional Prolog is not: Given the goal, $Q1$,

$$?-\mathtt{Y = a : b : nil, permute(X,Y)}$$

Prolog will correctly backtrack through the answers $X = a : b : nil$ and $X = b : a : nil$. However for the goal, $Q2$,

$$?-\mathtt{X = a : b : nil, permute(X,Y)}$$

Prolog will first return the answer $Y = a : b : nil$ and on subsequent backtracking will go into an infinite derivation without returning any more answers.

For languages with delay the program `permute` does behave symmetrically. For instance, if the above program is given to the NU-Prolog compiler, a pre-processor will generate the following *when declarations*:

$$
\begin{array}{l}
?-\mathtt{permute(X,Y)\ when\ X\ or\ Y.} \\
?-\mathtt{delete(X,Y:Z,U)\ when\ Z\ or\ U.}
\end{array}
$$

These may be read as: the call `permute(X,Y)` should delay until X or Y is not a variable, and that the call `delete(X,Y:Z,U)` should delay until Z or U is not a variable. Of course programmers can also annotate their programs with `when` declarations.

Given these declarations, the above goals will behave in a symmetric fashion, backtrack through the possible permutations and then fail. What happens is that, with $Q1$, execution proceeds as in standard Prolog because no atoms are delayed. With $Q2$, however, calls to `delete` are delayed and only woken after the recursive calls to `permute`.

It is also interesting to consider what will happen with the goal, $Q3$,

$$?-\mathtt{permute(X,Y).}$$

In this case the call will delay, and the answer *true* will be returned. This behavior is very different from traditional Prolog executed with any computation rule, and illustrates the difference that dynamic scheduling brings.

As another example consider the following program to find paths in a graph. The arcs in the graph are represented by facts.

$$\begin{array}{rcl} \texttt{path(X,Y)} & \leftarrow & \texttt{arc(X,Y)}. \\ \texttt{path(X,Y)} & \leftarrow & \texttt{arc(X,Z)},\ \texttt{path(Z,Y)}. \\ \texttt{arc(X,Y)} & \leftarrow & \texttt{X} = \texttt{a},\ \ \texttt{Y} = \texttt{b}. \\ \texttt{arc(X,Y)} & \leftarrow & \texttt{X} = \texttt{b},\ \ \texttt{Y} = \texttt{c}. \end{array}$$

The programmer can provide the when declaration:

$$?-\texttt{arc(X,Y) when ground(X) or ground(Y)}.$$

which indicates that, for efficiency, a call to `arc` should only be evaluated if one of the arguments is "ground", that is takes a fixed value.

Concerning the denotational semantics, the standard declarative approaches cannot be easily extended to deal with the `when` declarations. The standard model-theoretic semantics of van Emden and Kowalski [1976], indeed, is upward-closed with respect to the answers (for instance it identifies the two programs for p in the introduction), and therefore it presents the same problem as the semantics in [Jagadeesan et al. 1991]. But also a semantics which models exactly the computed answers, like the S-semantics [Falaschi et al. 1989], cannot capture the meaning of a `when` declaration. In fact in the S-semantics if an atom $q(X, Y)$ is in the model (its computation gives the empty answer) then also the atom $q(a, Y)$ should be in the model, while this is not the case with the program for q given in the introduction. In other words, the S-semantics is not able to capture the conditional information implicit in the `when` declaration.

3 Operational semantics

In this section we recall some basic notions and we define an operational semantics for constraint logic programs with dynamic scheduling. The operational semantics is based on that given in [Marriott et al. 1994; Debray et al. 1994].

A *constraint logic program*, or *program*, is a finite set of rules. A *rule* is of the form $H \leftarrow B$ where H, the *head*, is an atom and B, the *body*, is a finite, non-empty sequence of literals. We let *nil* denote the empty sequence. A *literal* is either an atom or a primitive constraint. An *atom* has the form $p(x_1, ..., x_n)$ where p is a predicate symbol and the x_i are distinct variables. A *primitive constraint* is essentially a predefined predicate, such as term equations or inequalities over the reals. Arguments to a primitive constraint are terms which may be constructed by using predefined functions such as real addition. The syntax given here is more restrictive than usual, as this will simplify the rest of the paper. However the restrictions are only syntactic, as we can always rewrite an atom $p(t_1, ..., t_n)$ with arbitrary terms as arguments into $x_1 = t_1, ..., x_n = t_n, p(x_1, ..., x_n)$.

A *constraint* is a conjunction of primitive constraints. Constraints are treated modulo logical equivalence, which we will denote by $\equiv$, and are assumed to be closed under existential quantification and conjunction. Constraints can be ordered by logical implication, that is $\theta \leq \theta'$ iff $\theta' \Rightarrow \theta$. We will assume that the set of constraints, under this ordering, is a complete lattice. The greatest constraint is denoted by *false* (unsatisfiable constraint) and it represents inconsistency. The least constraint is denoted by *true*. It is the always satisfiable constraint. We let $\exists_W \theta$ denote the constraint $\exists V_1 \exists V_2 \cdots \exists V_n \theta$ where variable set $W = \{V_1, \ldots, V_n\}$, and we let $\exists_{\bar{W}} \theta$ denote the restriction of the constraint θ to the variables in W. That is, $\exists_{\bar{W}} \theta$ is $\exists_{(vars\ \theta) \setminus W} \theta$, where the function *vars* takes a syntactic object and returns the set of (free) variables occurring in it.

Var is the set of variables, *Atom* the set of atoms, *Prim* the set of primitive constraints, *Con* the set of constraints, *Lit* the set of literals, *Rule* the set of rules, and *Prog* the set of programs.

A *renaming* is a bijective mapping from *Var* to *Var*. We let *Ren* be the set of renamings, and naturally extend renamings to mappings between atoms, rules, and constraints. Syntactic objects s and s' are said to be *variants* if there is a $\rho \in Ren$ such that $\rho\ s = s'$. The *definition of an atom A in program P*, $defn_P\ A$, is the set of variants of rules in P such that each variant has A as a head and, apart from the variables in A, has distinct new variables.

The operational semantics of a program is given in terms of the "derivations" from goals. Derivations are sequences of reductions between "states", where a *state* is a tuple $\langle G, \theta, D \rangle$ which contains the current literal sequence or "goal" G, the current constraint θ, and the current sequence of delayed atoms D. At each reduction step, a literal in the goal is *selected* according to some fixed *computation rule*, which is often left-to-right. If the literal is a primitive constraint, and it is consistent with the current constraints, then it is added to these and delayed atoms woken by this addition are added to the current goal. If the literal is an atom, then there are two cases. If the literal is not sufficiently instantiated to be processed, then it is placed in the delayed atom sequence. Otherwise, it is replaced by the body of one of the rules in its definition.

More formally, our definition of the operational semantics makes use of three functions which depend on the implementation or language being modeled. These are, *delay* A θ, which holds iff a call to the atom A delays with the constraint θ; *woken* D θ, which is the subsequence of literals in the sequence of delayed literals D that are woken by the constraint θ; and *merge* D G which returns the sequence of literals obtained by merging the sequence of the woken atoms D with the current goal G. Note that the order of the calls returned by *woken* and the position in which they are placed in the current goal by *merge* is system dependent.

A *derivation* from a goal G in a program P is a sequence of states $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ where S_0 is $\langle G, true, nil\rangle$ (nil is the empty goal) and each S_{i-1} is reduced to S_i, using clauses in P, as follows: Assume S_{i-1} is the state $\langle G, \theta, D\rangle$, select a literal L from G and let G' be the remaining literals in G. Then:

1. If $L \in Prim$ and $\theta \wedge L$ is satisfiable, then it is reduced to $\langle (merge\ D'\ G'), \theta \wedge L, D \setminus D'\rangle$ where $D' = woken\ D\ (\theta \wedge L)$.

2. If $L \in Atom$ and $delay\ L\ \theta$ holds, then it is reduced to $\langle G', \theta, L : D\rangle$.

3. If $L \in Atom$ and $delay\ L\ \theta$ does not hold, then it is reduced to $\langle B : G', \theta, D\rangle$ for some $(L \leftarrow B) \in defn_P\ L$ such that, for all $j \in [1, i-1]$,
$$(vars\ B) \cap (vars\ S_j) \subseteq (vars\ L).$$

Here we have used the symbol ":" to denote concatenation of literals, since the "," is used already to separate the different components of a state. In the following, we will consider these two symbols interchangeable and we will switch from one to the other when convenient.

A derivation from G is *complete* if the last state has the form $\langle nil, \theta, D\rangle$. In this case, the constraint $\exists_{\overline{(vars\ G)}}\theta$ is an *answer to* G. Given a program P and a goal G, we denote by $ans_P\ G$ the set of answers to G in P. (Since there can be more than one rule in a predicate's definition, there may be several answers generated from a given initial constraint.) In the case when no literals delay and the constraints are term equations, this semantics is the same as the usual operational semantics of pure Prolog. Note that the answers to a goal must always be satisfiable: from the definition it is impossible for *false* to be a valid answer.

As an example, consider the program `path` from Section 2 and the goal $Y = c : path(X, Y)$. Using a left-to-right computation rule, these have the complete derivation shown in Figure 1, which gives the answer $X = a \wedge Y = c$.

A complete derivation with last state $\langle nil, \theta, D\rangle$ is said to have *floundered* if D is not the empty goal. An answer to goal G for program P is *non-floundered* if it arises

$$\langle Y = c : path(X, Y),\ \ true,\ \ nil\rangle$$
$$\downarrow$$
$$\langle path(X, Y),\ \ Y = c,\ \ nil\rangle$$
$$\downarrow$$
$$\langle arc(X, Z) : path(Z, Y),\ \ Y = c,\ \ nil\rangle$$
$$\downarrow$$
$$\langle path(Z, Y),\ \ Y = c,\ \ arc(X, Z)\rangle$$
$$\downarrow$$
$$\langle arc(Z, Y),\ \ Y = c,\ \ arc(X, Z)\rangle$$
$$\downarrow$$
$$\langle Z = b : Y = c,\ \ Y = c,\ \ arc(X, Z)\rangle$$
$$\downarrow$$
$$\langle arc(X, Z) : Y = c,\ \ Z = b \wedge Y = c,\ \ nil\rangle$$
$$\downarrow$$
$$\langle X = a : Z = b : Y = c,\ \ Z = b \wedge Y = c,\ \ nil\rangle$$
$$\downarrow$$
$$\langle Z = b : Y = c,\ \ X = a \wedge Z = b \wedge Y = c,\ \ nil\rangle$$
$$\downarrow$$
$$\langle Y = c,\ \ X = a \wedge Z = b \wedge Y = c,\ \ nil\rangle$$
$$\downarrow$$
$$\langle nil,\ \ X = a \wedge Z = b \wedge Y = c,\ \ nil\rangle$$

Figure 1: Example of a derivation

from a complete derivation which has not floundered. We denote the set of non-floundered answers to G for program P by $nfans_P\ G$.

For example, the goal $Y = c : path(X, Y)$ and the program `path` from Section 2 have the non-floundered answers $X = a \wedge Y = c$ and $X = b \wedge Y = c$. The goal $path(X, Y)$, however, has the floundered answer $true$.

Following [Marriott et al. 1994] we assume that the functions *delay* and *woken* satisfy the following four conditions. The first ensures that there is a correspondence between the conditions for delaying a literal and waking it:

(1) $L \in (woken\ D\ \theta) \Leftrightarrow L \in D \wedge \neg\,(delay\ L\ \theta)$.

The remaining conditions ensure that *delay* behaves reasonably. It should not take variable names into account:

(2) Let $\rho \in Ren$. Then $delay\ L\ \theta \Leftrightarrow delay\ (\rho L)\ (\rho\ \theta)$.

It should only be concerned with the effect of θ on the variables in L:

(3) $delay\ L\ \theta \Leftrightarrow delay\ L\ \exists_{\overline{(vars\ L)}}\theta$.

Finally, if a literal is not delayed, adding more constraints should never cause it to delay:

(4) If $\theta' \leq \theta$ and $delay\ L\ \theta$, then $delay\ L\ \theta'$.

These conditions are crucial to the development of our semantics.

For simplicity we have ignored constraints which delay. These may be modeled in our setting by wrapping them with atoms which can delay. For example delay of non-linear multiplication constraints $X = Y * Z$ in CLP($\mathcal{R}$) is captured by the rule

$$\mathtt{mult(X, Y, Z)} \leftarrow \mathtt{X = Y * Z}.$$

where $delay\ mult(X, Y, Z)\ \theta$ holds whenever θ does not constrain Y or Z to be ground.

4 And-trees

In this section we give a semantics for languages with dynamic scheduling which is based on "and-trees". These capture the structure of derivations in a compositional way, and will provide the bridge between the operational semantics based on derivations, and the denotational semantics. In fact, we will see that an and-tree can be interpreted as a closure operator.

The following definition is given modulo reordering of literals in sequences to simplify the notation.

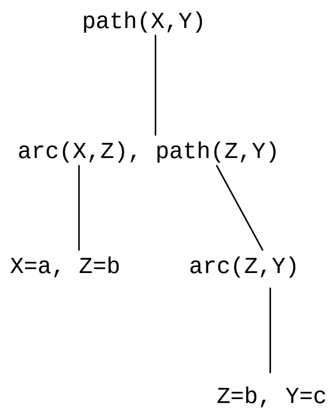

Figure 2: Example of an and-tree

DEFINITION 4.1 Let P be a program. An *and-tree* in P is defined inductively as follows:

- A node labeled with a sequence of literals is an and-tree;
- if T is an and-tree with a leaf node N labeled by $L_1 : \ldots : L_n$, then so is the tree obtained by adding nodes labeled by $B_{i_1}, \ldots, B_{i_m}$ as children of N, provided that the following conditions hold:

  (i) $m \leq n$ and $\{i_1, \ldots, i_m\}$ is a subset of the indexes $[1, n]$,

  (ii) for each $j \in [1, m]$ we have that $(L_{i_j} \leftarrow B_{i_j}) \in defn_P\ L_{i_j}$ and $(vars\ B_{i_j}) \cap (vars\ T) \subseteq (vars\ L_{i_j})$.

  In this case we say that the node labeled by B_{i_j} is attached to L_{i_j}.

In the following, to simplify the notation, we will often identify a node with its label, i.e. we assume the existence of a marking which associates an atom to the node in which it occurs. In this way we can distinguish among equal atoms labeling different nodes.

We say that T is an and-tree for the goal G in the program P if T is an and-tree in P and its root is labeled by G. Figure 2 shows an and-tree for the program `path` given in Section 2.

An and-tree can be traversed to give a derivation.

DEFINITION 4.2 Let T be an and-tree, and let θ be a constraint. An *and-tree derivation* based on T for θ of length m is a sequence of tuples of the form

$$\langle path(X,Y),\quad Y=c\rangle$$
$$\downarrow_{path(X,Y)}$$
$$\langle arc(X,Z) : path(Z,Y),\quad Y=c\rangle$$
$$\downarrow_{path(Z,Y)}$$
$$\langle arc(Z,Y) : arc(X,Z),\quad Y=c\rangle$$
$$\downarrow_{arc(Z,Y)}$$
$$\langle Z=b : Y=c : arc(X,Z),\quad Y=c\rangle$$
$$\downarrow_{Z=b}$$
$$\langle Y=c : arc(X,Z),\quad Z=b\wedge Y=c\rangle$$
$$\downarrow_{arc(X,Z)}$$
$$\langle X=a : Z=b : Y=c,\quad Z=b\wedge Y=c\rangle$$
$$\downarrow_{X=a}$$
$$\langle Z=b : Y=c,\quad X=a\wedge Z=b\wedge Y=c\rangle$$
$$\downarrow_{Z=b}$$
$$\langle Y=c,\quad X=a\wedge Z=b\wedge Y=c\rangle$$
$$\downarrow_{Y=c}$$
$$\langle nil,\quad X=a\wedge Z=b\wedge Y=c\rangle$$

Figure 3: Example of an and-tree derivation

$$\langle G_0,\theta_0\rangle \rightarrow_{L_0} \langle G_1,\theta_1\rangle \rightarrow_{L_1} \cdots \rightarrow_{L_{m-1}} \langle G_m,\theta_m\rangle$$

where:

1. Each G_i is a sequence of literals which appear in T, each θ_i is a constraint, and each L_i is a literal in G_i.
2. G_0 is (the label of) the root of T and θ_0 is $\exists_{(\overline{vars}\ G_0)}\theta$.
3. For each $i \in [0, m-1]$ there exists a literal L_i in the goal G_i such that one of the following cases holds:
   (a) Either L_i is an atom, *delay* $L_i\ \theta_i$ is not true, and L_i appears in a node N which has a child (labeled by) B_i and attached to L_i. In this case $\theta_{i+1} \equiv \theta_i$ and $G_{i+1} = B_i : G_i \setminus L_i$, where $G_i \setminus L_i$ denotes the sequence of literals obtained by removing L_i from G_i.
   (b) Otherwise, L_i is a primitive constraint. In this case $\theta_{i+1} \equiv \theta_i \wedge L_i$ and $G_{i+1} = G_i \setminus L_i$.

Note that, according to the previous definition, derivations are finite. A *maximal and-tree derivation* is a derivation in which from the last tuple $\langle G_m,\theta_m\rangle$ it is not possible to make any further step (either because G_m is empty or because the conditions 3(a) and 3(b) are not satisfied). A maximal and-tree derivation is *complete* if for all $A \in G_m$, *delay* $A\ \theta_m$ holds.

Figure 3 is an example of a complete and-tree derivation based on the and-tree given in Figure 2 for $Y = c$.

Clearly there may be more than one maximal and-tree derivation for a given and-tree and constraint because of the different ways we can select $L_i \in G_i$. However, these will all give rise to the same last state, as shown by the following proposition.

PROPOSITION 4.3 *Let T be an and-tree and θ a constraint. Every maximal and-tree derivation based on T for θ has the same last state.*

In the case when constraints are term equations, this proposition is a corollary of the confluence result for the operational semantics given by Yellick and Zachary [1989] and Naish [1992]. For completeness, however, we give a direct and simple proof for arbitrary constraint domains. The proposition is an immediate consequence of the next lemma.

In the following we denote by $\rightarrow$ the relation obtained from $\rightarrow_L$ by abstracting from the subscripts and we denote by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. Moreover, given an and-tree derivation, ξ, we denote its length by $|\xi|$.

LEMMA 4.4 *Let T be an and-tree, θ a constraint and let*

$$\xi :\ \langle G_0,\theta_0\rangle \rightarrow^* \langle G_h,\theta_h\rangle$$

and

$$\xi' :\ \langle G_0,\theta_0\rangle \rightarrow^* \langle G'_k,\theta'_k\rangle$$

be two derivations based on T for θ_0. Then there exists a tuple $\langle G,\theta\rangle$ and two derivations

$$\xi_1 : \langle G_h,\theta_h\rangle \rightarrow^* \langle G,\theta\rangle$$

and

$$\xi'_1 : \langle G'_k,\theta'_k\rangle \rightarrow^* \langle G,\theta\rangle$$

such that $|\xi_1| \le |\xi'|$ and $|\xi'_1| \le |\xi|$.

PROOF. Let us define $w = max(|\xi|,|\xi'|)$. The case $w = 0$ is trivial. For $w > 0$ the proof is by induction on w.

For the base case $w = 1$ we assume that $|\xi| = 1$ and $|\xi'| = 1$, since the other cases are immediate. Then we have two different derivation steps

$$\langle G_0, \theta_0 \rangle \rightarrow_L \langle G_1, \theta_1 \rangle$$

and

$$\langle G_0, \theta_0 \rangle \rightarrow_{L'} \langle G_1', \theta_1' \rangle.$$

By definition of derivation it follows that $\theta_1 \Rightarrow \theta_0$ and $\theta_1' \Rightarrow \theta_0$. Condition (4) in previous assumptions on the *delay* function implies that, for any literal L, if *delay* $L\ \theta_0 =$ *false* then *delay* $L\ \theta_1 =$ *false* and *delay* $L\ \theta_1' =$ *false*. Therefore, since conjunction of constraints is associative and commutative, from a straightforward analysis of the four cases arising from conditions (a) and (b) in Definition 4.2 it follows that there exists $\langle G, \theta \rangle$ such that

$$\langle G_1, \theta_1 \rangle \rightarrow_{L'} \langle G, \theta \rangle$$

and

$$\langle G_1', \theta_1' \rangle \rightarrow_L \langle G, \theta \rangle,$$

which concludes the proof of the base case.

We consider now the inductive step, $w > 1$ (see Figure 4 for an outline of the proof). We assume that $|\xi| > 0$ and $|\xi'| > 0$, since the other cases are immediate. Then ξ and ξ' are derivations of the form

$$\xi : \langle G_0, \theta_0 \rangle \rightarrow^* \langle G_{h-1}, \theta_{h-1} \rangle \rightarrow \langle G_h, \theta_h \rangle$$

and

$$\xi' : \langle G_0, \theta_0 \rangle \rightarrow^* \langle G_{k-1}', \theta_{k-1}' \rangle \rightarrow \langle G_k', \theta_k' \rangle$$

(where possibly either $h - 1 = 0$ or $k - 1 = 0$). By inductive hypothesis there exist two derivations

$$\gamma : \langle G_{h-1}, \theta_{h-1} \rangle \rightarrow^* \langle M, \sigma \rangle$$

and

$$\gamma' : \langle G_{k-1}', \theta_{k-1}' \rangle \rightarrow^* \langle M, \sigma \rangle$$

such that $|\gamma| \leq |\xi'| - 1$ and $|\gamma'| \leq |\xi| - 1$. Moreover, since there exist the two derivations

$$\langle G_{h-1}, \theta_{h-1} \rangle \rightarrow \langle G_h, \theta_h \rangle$$

and γ, again by inductive hypothesis there exist also the derivations

$$\xi_2 : \langle G_h, \theta_h \rangle \rightarrow^* \langle P, \tau \rangle$$

and

$$\gamma_2 : \langle M, \sigma \rangle \rightarrow^* \langle P, \tau \rangle$$

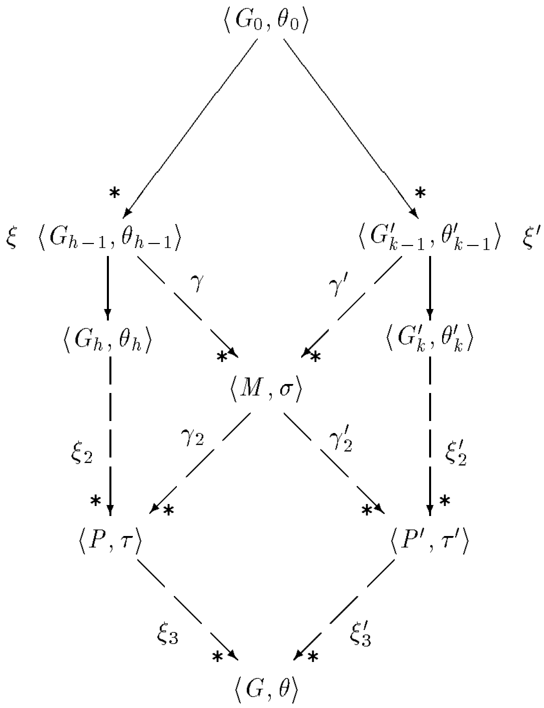

Figure 4: Outline of the proof of Lemma 4.4 for the inductive case.

such that

$$|\xi_2| \leq |\gamma| \leq |\xi'| - 1 \quad \text{and} \quad |\gamma_2| \leq 1. \tag{1}$$

Analogously there exist

$$\xi_2' : \langle G_k', \theta_k' \rangle \rightarrow^* \langle P', \tau' \rangle$$

and

$$\gamma_2' : \langle M, \sigma \rangle \rightarrow^* \langle P', \tau' \rangle$$

such that

$$|\xi_2'| \leq |\gamma'| \leq |\xi| - 1 \quad \text{and} \quad |\gamma_2'| \leq 1. \tag{2}$$

Finally, from the existence of the derivations γ_2' and γ_2 and the inductive hypothesis it follows that there exist $\langle G, \theta \rangle$ and two derivations

$$\xi_3 : \langle P, \tau \rangle \rightarrow^* \langle G, \theta \rangle$$

and

$$\xi_3' : \langle P', \tau' \rangle \rightarrow^* \langle G, \theta \rangle$$

such that

$$|\xi_3| \le |\gamma'_2| \le 1 \quad \text{and} \quad |\xi'_3| \le |\gamma_2| \le 1. \tag{3}$$

Now by taking $\xi_1 \equiv \xi_2 : \xi_3$ and $\xi'_1 \equiv \xi'_2 : \xi'_3$ from previous definitions it follows that

$$\xi_1 : \langle G_h, \theta_h \rangle \rightarrow^* \langle G, \theta \rangle$$

and

$$\xi'_1 : \langle G'_k, \theta'_k \rangle \rightarrow^* \langle G, \theta \rangle.$$

Moreover from the inequalities (1) and (3) it follows that $|\xi_1| \le |\xi'|$, while from (2) and (3) it follows $|\xi'_1| \le |\xi|$, which completes the proof. □

The previous result shows that, given an and-tree T, and a constraint θ, any maximal and-tree derivation based on T for θ will have the same final constraint, and that if one of them is complete, then all will be complete. In the latter case we say that the final constraint is the answer based on T for θ. In case the maximal derivations are not complete we say that the answer is *false*.

DEFINITION 4.5 Let T be an and-tree with root labeled by L and θ a constraint. The *answer* based on T for θ is the constraint $\theta \wedge \exists_{(\bar{vars}\ L)}\theta'$ if there is a complete and-tree derivation based on T for θ with last tuple $\langle G, \theta' \rangle$; otherwise it is *false*.

For instance, the answer based on the and-tree in Figure 2 for *true* is *true* and for $Y = c$ is $X = a \wedge Y = c$.

An and-tree can be seen as a function mapping an initial constraint to its answer. More precisely:

DEFINITION 4.6 Let T be an and-tree. The operator *ans* T on constraints maps a constraint θ to the answer based on T for θ.

A comparison of the derivation in Figure 1 and the and tree-derivation in Figure 3 shows the close relationship between and-tree derivations and derivations. This relationship is formalized by the following result.

PROPOSITION 4.7 *Let $\mathcal{T}_{GP}$ be the set of and-trees for a goal G in a program P. Then,*

$$ans_P\ G = \{ans\ T\ true \mid T \in \mathcal{T}_{GP}\} \setminus \{false\}.$$

PROOF. Straightforward by induction on the length of a derivation. □

5 A denotational semantics based on closure operators

In this section we construct a denotational semantics for programs with dynamic scheduling. The main feature of this semantics is that it is based on sets of closure operators on the underlying Constraint System, which, as we will see, allow to describe the operators of the language in a very simple way.

We first review some properties of closure operators which will be useful. See for example [Gierz et al. 1980] for more details.

Let $(X, \preceq, \sqcap)$ be a complete lattice, where $\sqcap$ is the meet operation, and consider a mapping $f : X \rightarrow X$:

- f is *monotonic* if $\forall x, x' \in X,\ x \preceq x' \Rightarrow f\ x \preceq f\ x'$;
- f is *idempotent* if $\forall x \in X,\ f\ (f\ x) = f\ x$;
- f is *extensive* if $\forall x \in X,\ x \preceq f\ x$; and
- f is a *closure operator* if it is monotonic, idempotent and extensive.

One fundamental property is that, given a closure operator f, the image of f, $\hat{f}$, coincides with the set of fixpoints (or resting points) of f, and this set is a complete meet sublattice of X. Namely for all $Y \in \hat{f}$, $\sqcap Y \in \hat{f}$ holds (the meet of the empty set is defined as the bottom element of X). Furthermore, for each $x \in X$, $f\ x = \sqcap\{z \in \hat{f} \mid x \preceq z\}$ holds. Vice versa, given a complete meet sublattice Z of X, the function $\lambda x.\sqcap\{z \in Z \mid x \preceq z\}$ is a closure operator. Furthermore, the set of fixpoints of this operator coincides with Z. In other words, there is a bijection between closure operators and complete meet sublattices of X, and we can represent a closure operator f by the set of its fixpoints. The result of f, applied to x, is the least fixpoint above x.

Closure operators on constraints will be used in our semantics as denotations of sequences of literals. The main reason for this choice is that the basic operation in our language, the concatenation of literals, can simply be modeled as intersection of the resting points of the corresponding closure operators. As stated before, the reason why we have introduced and-trees in previous section is that they can be viewed as closure operators which map initial constraints into the corresponding answers.

LEMMA 5.1 *For every and-tree T, ans T is a closure operator.*

PROOF. Let G be the root of T. Extensivity is immediate, since either *ans* $T\ \theta = false$, or *ans* $T\ \theta = \theta \wedge \exists_{(\bar{vars}\ G)}\theta'$, for some constraint θ'.

Concerning monotonicity, consider θ, θ' with $\theta \leq \theta'$, and consider the two cases:

1. There is no complete and-tree derivation based on T for θ, and therefore *ans* $T\ \theta = false$, or
2. There is a complete and-tree derivation ξ based on T for θ with last state $\langle G_m, \theta_m \rangle$, and therefore *ans* $T\ \theta = \theta \wedge \exists_{(\bar{vars}\ G)}\theta_m$.

In the first case, the reasons why there exist no complete derivation can be: (a) there are no maximal and-tree derivations, or (b) the maximal and-tree derivations end in a state $\langle G_m, \theta_m \rangle$ such that for some atom A in G_m, *delay* $A\ \theta_m$ is false. If (a) holds, then it is possible to construct an infinite and-tree derivation ξ' based on T for θ', just mimicking one and-tree derivation ξ based on T for θ. By induction we can easily show that at each state the constraint in ξ' will be bigger than or equal to the constraint in the corresponding state in ξ. Therefore, due to Condition (4) on the *delay* function, at each state the delayed atoms will be less than or equal to the delayed atoms in the corresponding state in ξ'. If (b) holds, then we can construct an and-tree derivation based on T for θ', whose final state is $\langle G_m, \theta'_m \rangle$ with $\theta_m \leq \theta'_m$. Again due to Condition (4) on the *delay* function, for some atom A in G_m, *delay* $A\ \theta'_m$ is false. So, either in (a) and in (b), ξ' is maximal but not complete, and therefore *ans* $T\ \theta' = false$.

In the second case, by mimicking ξ we can construct an and-tree derivation ξ' based on T for θ with last state $\langle G_m, \theta'_m \rangle$, where $\theta_m \leq \theta'_m$. This might be maximal or not maximal. Consider a maximal and-tree derivation ξ'' obtained by extending ξ'. If ξ'' is infinite, or not complete, then *ans* $T\ \theta' = false$ and we are done. Otherwise it will have a last state $\langle G_n, \theta'_n \rangle$, with $\theta'_m \leq \theta'_n$, and *ans* $T\ \theta' = \theta' \wedge \exists_{(\bar{vars}\ G)}\theta'_n$, which is bigger than or equal to $\theta \wedge \exists_{(\bar{vars}\ G)}\theta_m =$ *ans* $T\ \theta$.

Concerning idempotency, assume that *ans* $T\ \theta = \theta'$. If $\theta' = false$ then we are done. Otherwise, there exists a complete and-tree derivation ξ based on T for θ with final state $\langle G_m, \theta_m \rangle$, and $\theta' = \theta \wedge \exists_{(\bar{vars}\ G)}\theta_m$. Then by mimicking ξ we can construct an and-tree derivation ξ', still based on T, whose initial constraint is $\exists_{(\bar{vars}\ G)}\theta' = (\exists_{(\bar{vars}\ G)}\theta) \wedge (\exists_{(\bar{vars}\ G)}\theta_m)$. By induction it is easy to show that the constraint in the final state of ξ' will be $\theta_m \wedge (\exists_{(\bar{vars}\ G)}\theta_m)$, which is logically equivalent to θ_m. Thus *ans* $T\ \theta' = \theta'$, which completes the proof. □

We say that *ans* T is the closure operator *associated* with T. We now develop a denotational semantics which is based on the closure operators associated with and-trees.

Intuitively, in our semantics the denotation of a sequence of literals is a set of closure operators, corresponding to the and-trees which have this sequence as root. We represent each closure operator by its set of resting points. Thus the basis of our semantics is the set:

$$Clos = \{C \subseteq Con \mid C \text{ is a complete meet sublattice of } Con\}.$$

There are, however, some subtleties to do with ordering closures and the sets of closures. As is usual in a denotational semantics, we wish the order to reflect their information content – the more information contained in a closure operator or set of closure operators, the higher in the ordering it should be.

Recall that if $C \in Clos$ maps a constraint θ to *false*, then this indicates that we know nothing about the answer to θ. Thus we have the ordering on $C, C' \in Clos$ given by $C \leq_c C'$ iff

$$\forall \theta \in Con.(C\ \theta) \neq false \Rightarrow (C'\ \theta) = (C\ \theta).$$

The least element of *Clos* is $\{false\}$, and the greatest elements are the closure operators in which only *false* is mapped to *false*. Note that this ordering is *not* the pointwise ordering induced from the ordering on *Con*.

Now consider the ordering on sets of closure operators. As we are dealing with sets, we require some type of powerdomain construction in which the ordering on the sets also reflects the ordering on the elements in the sets. Given two sets of and-trees $\mathcal{T}$ and $\mathcal{T}'$, intuitively $\mathcal{T}'$ has more information than $\mathcal{T}$, if for every and-tree $T \in \mathcal{T}$ there is an and-tree in $\mathcal{T}'$ which extends T. This intuition is captured by the Hoare preordering: namely, for $\mathcal{C}, \mathcal{C}' \subseteq Clos$,

$$\mathcal{C} \subseteq_H \mathcal{C}' \quad \text{iff} \quad \forall C \in \mathcal{C}.\ \exists C' \in \mathcal{C}'.C \leq_c C'.$$

Note that from this definition it follows that $\mathcal{C} \subseteq_H \mathcal{C}'$ iff $\downarrow\mathcal{C} \subseteq \downarrow\mathcal{C}'$, where $\downarrow$ denotes the downwards closure, i.e.

$$\downarrow\mathcal{C} = \{C' \in Clos \mid \text{there exists } C \in \mathcal{C} \text{ such that } C' \leq_c C\}.$$

As a consequence, in the Hoare preordering, two sets with the same downward closure are equivalent (they have the same information content). Hence we can identify all such sets, and take as representative of their class their downward closure. Thus we consider as semantic domain the complete lattice of downward closed subsets of $\wp(Clos)$ ordered by $\subseteq$, namely those sets $\mathcal{C}$ such that $\mathcal{C} = \downarrow \mathcal{C}$. Furthermore, as it is usual in the construction of the Hoare powerdomain, we discard the empty set since it has no information content, and the absence of information content is already represented by the set $\{\{false\}\}$. In summary, the denotational domain is the complete lattice $(\Re\ Clos, \subseteq)$, where

$$\Re\ Clos = \{\mathcal{C} \subseteq Clos | \mathcal{C} \text{ is downwards closed and } \mathcal{C} \neq \emptyset\}.$$

with least element $\{\{false\}\}$.

In the denotational semantics, each syntactic category – atom, primitive constraint, literal, and body, is associated with an element of $\Re\ Clos$. We assume that if A is not defined in P then P is extended by adding the clause $A \leftarrow false$. The semantic equations are given in Figure 5. The definition makes use of the auxiliary functions

$$\begin{array}{lcl} \theta\uparrow & = & \{\theta' \in Con \mid \theta \leq \theta'\} \\ restrict\ W\ C & = & \{\theta' \in Con \mid \text{ there exists } \theta \in C \text{ such that } \exists_{\overline{W}}\theta = \exists_{\overline{W}}\theta'\} \\ delays\ A & = & \{\theta \in Con \mid delay\ A\ \theta \text{ holds}\}. \end{array}$$

To understand the equations, it is easier to think in terms of resting points. For an atom A, a constraint θ is a resting point if A is delayed with θ, or else θ is a resting point in a closure for the body of some rule defining A. For a primitive constraint θ, the denotation is (the downward closure of) the single closure operator $\{\theta\uparrow\}$. This is because a primitive constraint has a single and-tree, which corresponds to a single closure operator. The closure operator is $\theta\uparrow$, because these are exactly the constraints which remain unchanged when θ is conjoined with them. The meaning of a body of literals, $L_1 : ... : L_n$, is obtained as follows. First assume that each L_i is deterministic, with closure operator C_i. Then a constraint is a resting point for the body iff it is a resting point for each literal in the body. That is, the closure operator for $L_1 : ... : L_n$ is $\bigcap_{i=1}^{n} C_i$. In the case that the L_i are nondeterministic, then the denotation is obtained by combining all possible choices for each L_i.

As an example, consider the case when P is the program `path` from Section 2. We show the denotation of `arc`; the intended environment, omitted for simplicity, is of course $\textbf{fix}\ \Gamma_P$.

$$\begin{array}{lcl} \mathbf{L}_P\ [\![X = a]\!] & = & \downarrow\{(X = a)\uparrow\} \\ \mathbf{L}_P\ [\![Y = b]\!] & = & \downarrow\{(Y = b)\uparrow\} \end{array}$$

Thus,

$$\begin{array}{rl} \mathbf{B}_P\ [\![X = a : Y = b]\!] & = \downarrow\{(X = a)\uparrow \cap (Y = b)\uparrow\} \\ & = \downarrow\{(X = a \wedge Y = b)\uparrow\}. \end{array}$$

Similarly,

$$\mathbf{B}_P\ [\![X = b : Y = c]\!] = \downarrow\{(X = b \wedge Y = c)\uparrow\}.$$

Thus,

$$\mathbf{A}_P\ [\![arc(X, Y)]\!] = \downarrow\{C_1, C_2\}$$

where

$$\begin{array}{lcl} C_1 & = & C_{ng} \cup (X = a \wedge Y = b)\uparrow \\ C_2 & = & C_{ng} \cup (X = b \wedge Y = c)\uparrow \end{array}$$

with

$$C_{ng} = \{\theta \in Con \mid \theta \text{ does not ground } X \text{ and does not ground } Y\}.$$

It is easy to see that the semantic equations are well defined, i.e. that they return elements of $\Re Clos$, and that Γ_P is a continuous function (since the underlying ordering is set inclusion, the proof is analogous to the proof of continuity of the standard T_p operator of logic programming). Let us denote by $\Gamma_P \uparrow n$ the natural powers of Γ_P applied to the least element of the denotational domain ($\{\{false\}\}$). Namely:

$$\begin{array}{lcl} \Gamma_P \uparrow 0 & = & \lambda G.\ \{\{false\}\} \\ \Gamma_P \uparrow n+1 & = & \Gamma_P(\Gamma_P \uparrow n). \end{array}$$

By the well-known theorem on the least fixpoint of continuous functions, we have

$$\textbf{fix}\ \Gamma_P\ =\ \lambda G.\ \bigcup_{n \geq 0} \Gamma_P \uparrow n\ G.$$

This characterization of $\textbf{fix}\ \Gamma_P$ will be used to prove the correctness of the denotational semantics with respect to the operational one, based on and-trees. We first need some preliminary results. The first one states the relations between the fixpoint of a tree T whose root is labeled by an atom and those of the immediate

The semantic functions for a given program P are:

$$\begin{array}{lcl} \mathbf{A}_P & : & Atom \to Env \to \Re Clos \\ \mathbf{B}_P & : & Lit^\star \to Env \to \Re Clos \\ \Gamma_P & : & Env \to Env, \end{array} \qquad \begin{array}{lcl} \mathbf{L}_P & : & Lit \to Env \to \Re Clos \\ \mathbf{R}_P & : & Rule \to Env \to \Re Clos \end{array}$$

where $Env = [Lit^\star \to \Re Clos]$. These functions are defined as

$$\begin{array}{llcl} D1 & \mathbf{A}_P\ [\![A]\!]\ e & = & \bigcup\{\mathbf{R}_P\ [\![R]\!]\ e \mid R \in defn_P\ A\} \\ D2 & \mathbf{R}_P\ [\![A \leftarrow B]\!]\ e & = & \downarrow\{\ (delays\ A) \cup (restrict\ (vars\ A)\ C) \mid \\ & & & \quad C \in (e\ B)\ \} \\ D3 & \mathbf{B}_P\ [\![nil]\!]\ e & = & \downarrow\{Con\} \\ D4 & \mathbf{B}_P\ [\![L : B]\!]\ e & = & \{C \cap C' \mid C \in \mathbf{L}_P\ [\![L]\!]\ e \text{ and } C' \in \mathbf{B}_P\ [\![B]\!]\ e\} \\ D5 & \mathbf{L}_P\ [\![L]\!]\ e & = & \text{if } L \in Atom \text{ then } \mathbf{A}_P\ [\![L]\!]\ e \text{ else } \downarrow\{L\uparrow\} \\ D6 & \Gamma_P\ e\ G & = & \mathbf{B}_P\ [\![G]\!]\ e. \end{array}$$

The denotational semantics for a sequence of literals G is given by $\mathbf{B}_P\ [\![G]\!]\ (\mathbf{fix}\ \Gamma_P)$, or equivalently by $\mathbf{fix}\ \Gamma_P\ G$, where $\mathbf{fix}\ \Gamma_P$ is the least fixpoint of Γ_P.

Figure 5: Denotational semantics

subtree of T. In the following lemma, we need to assume that $(vars\ A) \subseteq (vars\ B)$ for any clause $A \leftarrow B$. Note that there is no loss of generality here, since a clause $p(\tilde{x}) \leftarrow B$ can always be equivalently rewritten as $p(\tilde{x}) \leftarrow \tilde{x} = \tilde{x}, B$. From this assumption we have that, for any constraint θ

$$\exists_{(\bar{vars}\ B)}\theta \Rightarrow \exists_{(\bar{vars}\ A)}\theta \tag{4}$$

and

$$\exists_{(\bar{vars}\ B)}\exists_{(\bar{vars}\ A)}\theta = \exists_{(\bar{vars}\ A)}\theta. \tag{5}$$

We recall that the closure operator $ans\ T$ is represented by the set of its fixpoints.

LEMMA 5.2 *Let T be a tree whose root is labeled by the atom A and let T' be the (unique) immediate subtree of T. Then*

$$ans\ T = (delays\ A) \cup (restrict\ (vars\ A)\ (ans\ T')).$$

PROOF. From the definition it is immediate to check that $false \in (ans\ T)$ for any and-tree T. So so we do not need to consider this case and in the following we assume $\theta \neq false$. We prove the two inclusions separately.

($\subseteq$) Consider $\theta \in (ans\ T)$. If $delay\ A\ \exists_{(\bar{vars}\ A)}\theta$ holds, then Condition (3) in the definition of the $delay$ function implies that $\theta \in delays\ A$. Assume now that $delay\ A\ \exists_{(\bar{vars}\ A)}\theta$ does not hold. Since θ is a fixpoint, from Definitions 4.2, 4.5, and 4.6, it follows that there exists a complete and-tree derivation for θ based on T of the form

$$\xi :\ \langle A, \exists_{(\bar{vars}\ A)}\theta\rangle \to \langle B, \exists_{(\bar{vars}\ A)}\theta\rangle \to^* \langle N, \theta'\rangle$$

where B is the label of the root of T'. Moreover, since $\theta \wedge \exists_{(\bar{vars}\ A)}\theta'$ is the answer of derivation ξ and θ is a fixpoint, we have $\theta \Rightarrow \exists_{(\bar{vars}\ A)}\theta'$, from which we derive

$$\exists_{(\bar{vars}\ A)}\theta \Rightarrow \exists_{(\bar{vars}\ A)}\theta' \tag{6}$$

From the existence of ξ and (5) it follows that there exists also a complete and-tree derivation for $\exists_{(\bar{vars}\ A)}\theta$ based on T'

$$\langle B, \exists_{(\bar{vars}\ B)}\exists_{(\bar{vars}\ A)}\theta\rangle \to^* \langle N, \theta'\rangle$$

with answer $\sigma \equiv \exists_{(\bar{vars}\ A)}\theta \wedge \exists_{(\bar{vars}\ B)}\theta'$.

By the definition of restrict, it remains only to prove that $\exists_{(\bar{vars}\ A)}\sigma \equiv \exists_{(\bar{vars}\ A)}\theta$. We first observe that $\exists_{(\bar{vars}\ A)}\sigma \equiv \exists_{(\bar{vars}\ A)}\theta \wedge \exists_{(\bar{vars}\ A)}\exists_{(\bar{vars}\ B)}\theta'$, from which we derive, by (5), that $\exists_{(\bar{vars}\ A)}\sigma \equiv \exists_{(\bar{vars}\ A)}\theta \wedge \exists_{(\bar{vars}\ A)}\theta'$. Finally, apply (6).

($\supseteq$) Let $\theta \in (delays\ A) \cup (restrict\ (vars\ A)\ (ans\ T'))$. If $\theta \in delays\ A$ the same argument as before shows that $\theta \in ans\ T$.
Consider then $\theta \in restrict\ (vars\ A)\ (ans\ T')$. By definition of $restrict$, there exists $\sigma \in (ans\ T') \setminus (delays\ A)$ such that

$$\exists_{(\bar{vars}\ A)}\theta \equiv \exists_{(\bar{vars}\ A)}\sigma \tag{7}$$

Consider the complete and-tree derivation for σ based on T'

$$\xi : \langle B, \exists_{(\bar{vars}\ B)}\sigma\rangle \rightarrow^* \langle N, (\exists_{(\bar{vars}\ B)}\sigma) \wedge \sigma'\rangle$$

(here we denote by σ' the conjunction of constraints which are added to $\exists_{(\bar{vars}\ B)}\sigma$ during the derivation). Since $\sigma \wedge \exists_{(\bar{vars}\ B)}\sigma'$ is the answer of the previous derivation and σ is a fixpoint, we have $\sigma \Rightarrow \exists_{(\bar{vars}\ B)}\sigma'$, from which we derive

$$\exists_{(\bar{vars}\ B)}\sigma \Rightarrow \exists_{(\bar{vars}\ B)}\sigma'. \tag{8}$$

Now consider a complete and-tree derivation for $\exists_{(\bar{vars}\ A)}\sigma$ based on T' following the steps of ξ, modulo the replacement of $\exists_{(\bar{vars}\ B)}\sigma$ for $\exists_{(\bar{vars}\ A)}\sigma$. Because of Condition 4 on the *delay* function and (4) we can at most reach the final state of ξ, possibly stopping before because of a delay condition. Hence there exists a complete and-tree derivation for $\exists_{(\bar{vars}\ A)}\sigma$ based on T'

$$\langle B, \exists_{(\bar{vars}\ A)}\sigma\rangle \rightarrow^* \langle N', (\exists_{(\bar{vars}\ A)}\sigma) \wedge \sigma''\rangle$$

such that

$$\sigma' \Rightarrow \sigma''. \tag{9}$$

Since by hypothesis $\theta \notin$ *delays* A and hence $\exists_{(\bar{vars}\ A)}\theta \notin$ *delays* A, we can construct a complete and-tree derivation for θ based on T

$$\begin{aligned}\xi' :\ & \langle A, \exists_{(\bar{vars}\ A)}\theta\rangle \equiv \\ & \langle A, \exists_{(\bar{vars}\ A)}\sigma\rangle \rightarrow \langle B, \exists_{(\bar{vars}\ A)}\sigma\rangle \rightarrow^* \\ & \rightarrow^* \langle N', (\exists_{(\bar{vars}\ A)})\sigma \wedge \sigma''\rangle\end{aligned}$$

whose answer is $\theta \wedge \exists_{(\bar{vars}\ A)}((\exists_{(\bar{vars}\ A)}\sigma) \wedge \sigma'') \equiv \theta \wedge \exists_{(\bar{vars}\ A)}\sigma \wedge \exists_{(\bar{vars}\ A)}\sigma''$. It remains to show that this answer coincides with θ. By (8) and (5) we obtain $\exists_{(\bar{vars}\ A)}\sigma \equiv \exists_{(\bar{vars}\ A)}\exists_{(\bar{vars}\ B)}\sigma \Rightarrow \exists_{(\bar{vars}\ A)}\exists_{(\bar{vars}\ B)}\sigma' \equiv \exists_{(\bar{vars}\ A)}\sigma'$. By (9) we hence derive $\exists_{(\bar{vars}\ A)}\sigma \Rightarrow \exists_{(\bar{vars}\ A)}\sigma''$. Therefore the answer of ξ' can be rewritten as $\theta \wedge \exists_{(\bar{vars}\ A)}\sigma$. Finally, apply (7).

□

We also need to consider the relations between the fixpoints of two different subtrees T' and T'' and those of the tree obtained by "merging" T' and T''. For merging to make sense, we require that the "local" variables in T' and T'' are disjoint.

DEFINITION 5.3 Let T' and T'' be two and-trees of P such that

$$(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B),$$

where L and B are the labels of the roots of T' and T'', respectively. We define $T = merge\ T'\ T''$ as the and-tree of P such that

(i) the label of the root of T is $L : B$,

(ii) the immediate subtrees of T are T' and T''.

LEMMA 5.4 *Let T' and T'' be two and-trees of P such that $(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B)$, where L and B are the labels of the roots of T' and T'', respectively, and let $T = merge\ T'\ T''$. Then $ans\ T = (ans\ T') \cap (ans\ T'')$.*

PROOF.
We prove the two inclusions separately.

($\supseteq$) Let $\theta \in (ans\ T') \cap (ans\ T'')$. From Definition 4.6 and Definition 4.5 it follows that there exist two complete and-tree derivations for θ

$$\langle L, \exists_{(\bar{vars}\ L)}\theta\rangle \rightarrow^* \langle M', (\exists_{(\bar{vars}\ L)}\theta) \wedge \theta'\rangle$$

and

$$\langle B, \exists_{(\bar{vars}\ B)}\theta\rangle \rightarrow^* \langle M'', (\exists_{(\bar{vars}\ B)}\theta) \wedge \theta''\rangle$$

based on T' and T'' respectively, such that

$$\theta \Rightarrow \exists_{(\bar{vars}\ L)}\theta' \quad \text{and} \quad \theta \Rightarrow \exists_{(\bar{vars}\ B)}\theta''. \tag{10}$$

From the existence of previous derivations, the assumption $(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B)$, and Condition (3) on *delay*, it follows that there exist also the complete and-tree derivations for $\exists_{(\bar{vars}\ L:B)}\theta$

$$\langle L, \exists_{(\bar{vars}\ L:B)}\theta\rangle \rightarrow^* \langle M', (\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta'\rangle$$

and

$$\langle B, \exists_{(\bar{vars}\ L:B)}\theta\rangle \rightarrow^* \langle M'', (\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta''\rangle$$

based on T' and T'' respectively.

Now, since the derivation leading to M' is complete, for every atom A' in M', *delay* A' $(\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta'$ holds. From (10) and the assumption $(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B)$, we have that $(\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta' \equiv (\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta' \wedge \exists_{(\bar{vars}\ L:B)}\theta''$, hence *delay* A' $(\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta' \wedge \exists_{(\bar{vars}\ L:B)}\theta'$ holds as well. From Condition (3) on *delay* we finally derive that *delay* A' $(\exists_{(\bar{vars}\ L:B)}\theta) \wedge \theta' \wedge \theta''$ holds too.

Analogously, we can show that, for every atom A'' in M'', *delay* A'' $(\exists_{(vars^- L:B)}\theta) \wedge \theta' \wedge \theta''$ holds. Therefore, by combining the previous derivations we obtain a complete and-tree derivation for θ based on T:

$$\begin{array}{l}\langle L : B, \exists_{(vars^- L:B)}\theta\rangle \rightarrow^* \\ \langle M' : B, (\exists_{(vars^- L:B)}\theta) \wedge \theta'\rangle \rightarrow^* \\ \langle M' : M'', (\exists_{(vars^- L:B)}\theta) \wedge \theta' \wedge \theta''\rangle\end{array}$$

whose answer is $\sigma \equiv \theta \wedge \exists_{(vars^- L:B)}(\theta' \wedge \theta'')$.

It remains only to show that $\sigma \equiv \theta$. Let X' be the set of variables occurring in θ' which do not occur in L, and let X'' be the set of variables occurring in θ'' which do not occur in B. By the assumption $(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B)$, we know that the variables of X' do not occur in θ'', and vice versa the variables of X'' do not occur in θ'. Hence we can write $\sigma \equiv \theta \wedge \exists_{X'}\exists_{X''}(\exists_{X''}\theta' \wedge \exists_{X'}\theta'')$, from which we derive $\sigma \equiv \theta \wedge \exists_{X'}(\exists_{X''}\theta' \wedge \exists_{X''}\exists_{X'}\theta'') \equiv \theta \wedge \exists_{X'}\exists_{X''}\theta' \wedge \exists_{X'}\exists_{X''}\theta''$. From (10) it follows that $\sigma \equiv \theta$. Hence we conclude $\theta \in ans\ T$.

($\supseteq$) If $\theta \in ans\ T$, by using Proposition 4.3, we can construct a derivation $\xi = \xi' : \xi''$ for θ based on T which has θ as answer and such that in each step of ξ' the selected literal is connected to L is T, and in each step of ξ'' the selected literal is connected to B is T. From ξ' we can derive (by eliminating B from the states) an and-tree derivation for θ based on T', and analogously from ξ'' we can derive an and-tree derivation for θ based on T''. By using the same arguments as before we can show that ξ' and ξ'' are complete, thus concluding the proof.

□

We can now state the main lemma needed for the correctness result. In the following, the depth of a tree T is the maximal number of nodes contained in any root-leaf path in T.

LEMMA 5.5 *Let G be a goal and P be a program. If T is an and-tree for G in P, then there exists n such that $ans\ T \in (\Gamma_P \uparrow n)\ G$. Conversely, for any $n \geq 1$, if $C \in (\Gamma_P \uparrow n)\ G$ then there exists a tree T for G in P such that $C \leq_c (ans\ T)$.*

PROOF. The proof of the first part is by structural induction on G and by induction on the depth d of T. We have the following cases.

$G \equiv \theta$. Obvious.

$G \equiv nil$. Obvious.

$G \equiv L : B$. Consider a tree T for $L : B$. From Definition 4.1 it follows that $T = merge\ T'\ T''$ where T' is a tree for L, T'' is a tree for B and $(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B)$. By inductive hypothesis there exist n, m such that $C' \in (\Gamma_P \uparrow n)\ L$, $C'' \in (\Gamma_P \uparrow m)\ B$, $ans\ T' = C'$ and $ans\ T'' = C''$. This together with Lemma 5.4 implies that $ans\ T = C' \cap C''$. Since Γ_P is monotonic, we have that, for $k = max(n, m)$, $C' \in (\Gamma_P \uparrow k)\ L$ and $C'' \in (\Gamma_P \uparrow k)\ B$. Now from Equation $D4$ it follows that $ans\ T = C' \cap C'' \in (\Gamma_P \uparrow k)\ (L : B)$, thus completing the proof of this case.

$G \equiv A$. The proof here is by induction on d. For the base case $d = 1$, from Definition 4.1 it follows that the unique node of T is labeled by A. Moreover Definition 4.5 and Definition 4.6 imply

$$ans\ T \quad = \quad \{false\} \cup (delays\ A). \tag{11}$$

Now, from Equations $D1$ and $D2$ it follows that $\{false\} \cup (delays\ A) \in (\Gamma_P \uparrow 1)\ A)$ which completes the proof of the base case.

For the inductive step consider a tree T of depth $d > 1$ whose root is labeled by A. Let T' be the (unique) immediate subtree of T and let B be the label of the root of T', so $defn_P\ A$ contains the clause $A \leftarrow B$. From Lemma 5.2 it follows that

$$\begin{array}{rcl} ans\ T & = & (delays\ A) \cup \\ & & (restrict\ (vars\ A)\ (ans\ T')) \end{array} \tag{12}$$

By inductive hypothesis there exists n such that $(ans\ T') \in (\Gamma_P \uparrow n)\ B$. Now from Equations $D1$ and $D2$ it follows that

$$\begin{array}{l}(delays\ A) \cup (restrict\ (vars\ A)\ (ans\ T')) \\ \in (\Gamma_P \uparrow n+1)\ A,\end{array}$$

which, together with (12), concludes the proof of the first part.

The proof of the second part is by structural induction on G and by induction on n. We have the following cases.

$G \equiv \theta$. Obvious.

$G \equiv nil$. Obvious.

$G \equiv L : B$. Consider $C \in (\Gamma_P \uparrow n)\ (L : B)$. By Equation $D4$ there exist $C' \in (\Gamma_P \uparrow n)\ L$ and $C'' \in (\Gamma_P \uparrow n)\ B$ such that $C = C' \cap C''$. By inductive hypothesis there exist a tree T' for L in P such that $C' \leq_c ans\ T'$ and a tree T'' for B in P such that $C'' \leq_c ans\ T''$. Note that we can assume without loss of generality that $(vars\ T') \cap (vars\ T'') \subseteq (vars\ L) \cap (vars\ B)$. Consider now $T = merge\ T'\ T''$ which, by definition, is an and-tree for $L : B$ in P. From Lemma 5.4, it follows that $ans\ T = (ans\ T') \cap (ans\ T'')$, and therefore we obtain $C = C' \cap C'' \leq_c (ans\ T') \cap (ans\ T'') = ans\ T$.

$G \equiv A$. Consider $C \in (\Gamma_P \uparrow n)\ A$. The proof is by induction on n. The base case, $n = 1$, is immediate. For the inductive step, $n > 1$, Equations $D1$ and $D2$ imply that $C \leq_c (delays\ A) \cup (restrict\ (vars\ A)\ C')$ where $(A \leftarrow B) \in defn_P\ A$ and $C' \in (\Gamma_P \uparrow n-1)\ B$.

By the inductive hypothesis, there exists a tree T' for B in P such that $C' \leq_c ans\ T'$. Note that we can assume without loss of generality that $(vars\ T') \cap (vars A) \subseteq (vars\ B)$. Consider now the tree T such that

(i) the label of the root of T is A,

(ii) T' is the (only) immediate subtree of T,

Since $(A \leftarrow B) \in defn_P\ A$ and $(vars\ T') \cap (vars\ A) \subseteq (vars\ B)$, from Definition 4.1 it follows that T is an and-tree for A in P. Then Lemma 5.2 implies that $ans\ T = (delays\ A) \cup restrict\ (vars\ A)\ (ans\ T')$ which, together with the hypothesis $C' \leq_c ans\ T'$, shows that $C \leq_c ans\ T$ and completes the proof.

□

We can now prove the correctness of the denotational semantics with respect to the operational one:

THEOREM 5.6 *Let G be a goal and P a program. Then*

$$ans_P\ G = \{\sqcap C \mid C \in \textbf{fix}\ \Gamma_P\ G\} \setminus \{false\}.$$

PROOF.

($\subseteq$) If $\theta \in (ans_P\ G)$, there is a tree T for G in P such that $\theta = \sqcap(ans\ T)$. By Lemma 5.5 there exists an n such that $ans\ T \in (\Gamma_P \uparrow n)\ G$. By monotonicity of Γ_P, we have $ans\ T \in \textbf{fix}\ \Gamma_P\ G$, which concludes this direction of the proof.

($\supseteq$) Let $\theta = \sqcap C$, $\theta \neq false$, for some $C \in \textbf{fix}\ \Gamma_P\ G$. Then by continuity of Γ_P we have that there exists n such that $C \in (\Gamma_P \uparrow n)\ G$. By Lemma 5.5 there exists a tree T for G in P such that $C \leq_c ans\ T$. By definition of $\leq_c$, and since $\theta \neq false$, we derive $\theta = \sqcap(ans\ T)$, which concludes the proof.

□

It is interesting to consider the special case when no literals can delay since this is the traditional constraint logic program operational semantics. In this case each closure will be constructed by intersecting the closures associated with the constraints encountered in the derivation. That is, if the constraints $\theta_1, ..., \theta_n$ are encountered it will be $\bigcap_{i=1}^{n}(\theta_i \uparrow)$. But this is just $(\bigwedge_{i=1}^{n} \theta_i) \uparrow$. Thus in this simpler case the closure semantics is equivalent to a semantics which maps atoms to sets of constraints. This indicates that our closure based semantics is the analogue for (constraint) logic languages with dynamic scheduling of the S-semantics [Falaschi et al. 1989] for logic programming with fixed atom scheduling and of its extension to CLP [Gabbrielli et al. 1995].

5.1 About full abstraction

We recall that a denotational semantics is *fully abstract* if, whenever it distinguishes two syntactic objects, then there is a context in which the observational behavior of these two objects is different. In our case, the syntactic objects are essentially the literals, and the contexts are the goals in which these literals can be placed. The next example shows that our semantics is not fully abstract:

EXAMPLE 5.7 Consider the constraint domain *Con* in which there are two distinct constants, a and b and equalities are allowed. Define the predicates

$$\begin{array}{rcl} pa(X,Y) & \leftarrow & Y = b. \\ pb(X,Y) & \leftarrow & Y = a. \\ \\ p(X,Y) & \leftarrow & pa(X,Y). \\ p(X,Y) & \leftarrow & pb(X,Y). \\ \\ q(X,Y) & \leftarrow & p(X,Y). \\ q(X,Y) & \leftarrow & r(X,Y). \end{array}$$

The delaying conditions are

$$\begin{array}{rcl} delay\ pa(X,Y)\ c & \Leftrightarrow & X = a \not\leq c \\ delay\ pb(X,Y)\ c & \Leftrightarrow & X = b \not\leq c \\ delay\ r(X,Y)\ c & \Leftrightarrow & true \end{array}$$

The atoms $p(X, Y)$ and $q(X, Y)$ are observationally equivalent because, in whatever goal they are inserted, they will deliver the same answers for any initial constraint. However, the denotation of $p(X, Y)$ and $q(X, Y)$ is different. In fact, the denotation of $pa(X, Y)$ is $\downarrow \{C_a\}$ where

$$C_a = \overline{(X = a)\uparrow} \cup (X = a \wedge Y = b) \uparrow,$$

the denotation of $pb(X, Y)$ is $\downarrow \{C_b\}$ where

$$C_b = \overline{(X = b)\uparrow} \cup (X = b \wedge Y = a) \uparrow,$$

and the denotation of $p(X, Y)$ is $\downarrow \{C_a, C_b\}$. On the other hand, the denotation of $r(X, Y)$ is $\downarrow \{Con\}$, and the denotation of $q(X, Y)$ is $\downarrow \{Con, C_a, C_b\}$, which is different from $\downarrow \{C_a, C_b\}$.

A natural question which arises at this point is whether full abstraction could be achieved by performing a saturation similar to the one applied in [Saraswat et al. 1991]. In our setting, this would mean to add to the denotation of an atom A all the closure operators C such that $C \supseteq C'$ for some C' in the denotation of A. However, in our case this operation would be unsound. For instance, in previous example the denotation of $p(X, Y)$ would be forced to contain also the closure operator

$$C_{ab} = \overline{(X = a)\uparrow} \cup (X = a \wedge Y = b) \uparrow \cup (X = a \wedge Y = a) \uparrow,$$

which, when trying to retrieve the observables according to Theorem 5.6 in a context which provides the constraint $X = a$, would deliver (among others) the wrong answer $X = a \wedge Y = a$.

6 Conclusion

We have given a simple denotational semantics for logic programs with dynamic scheduling. The semantics is based on sets of closure operators. In a sense this is the analogue of the bottom-up S-semantics for usual logic programs, since, if no atoms are allowed to delay, then the semantic definitions are equivalent to the semantics in which atoms are mapped to their set of answer constraints.

The semantics can be used as a basis for the analysis of logic programs with dynamic scheduling. The idea is that closure operators can be abstracted by descriptions which capture their behavior. This idea has been pursued in [de la Banda et al. 1995] to give the first practical framework for the analysis of logic programs with dynamic scheduling.

References

DE BOER, F. S. AND PALAMIDESSI, C. 1991. A Fully Abstract Model for Concurrent Constraint Programming. In *Proc. TAPSOFT'91*, S. Abramsky and T. Maibaum, Eds. Lecture Notes in Computer Science, vol. 493. Springer-Verlag, Berlin, 296–319.

DE LA BANDA, M. G., MARRIOTT, K., AND STUCKEY, P. 1995. Efficient analysis of logic programs with dynamic scheduling. In *Proc. Twelfth International Logic Programming Symposium*, J. Lloyd, Ed. MIT Press, 417–431.

DEBRAY, S., GUDEMANN, D., AND BIGOT, P. 1994. Detection and optimization of suspension-free logic programs. In *Proc. Eleventh International Logic Programming Symposium*, M. Bruynooghe, Ed. The MIT Press, Cambridge, Mass., 487–504.

FALASCHI, M., LEVI, G., MARTELLI, M., AND PALAMIDESSI, C. 1989. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science 69*, 3, 289–318.

GABBRIELLI, M., DORE, G., AND LEVI, G. 1995. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation 5*, 2, 133–171.

GIERZ, G., HOFMANN, K., KEIMEL, K., LAWSON, J., MISLOVE, M., AND SCOTT, D. 1980. *A Compendium of Continuous Lattices*. Springer-Verlag.

JAFFAR, J. AND LASSEZ, J.-L. 1987. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 111–119.

JAGADEESAN, R., SHANBHOGUE, V., AND SARASWAT, V. 1991. Angelic non-determinism in concurrent constraint programming. Tech. rep., System Science Lab., Xerox PARC.

MARRIOTT, K., DE LA BANDA, M. G., AND HERMENEGILDO, M. 1994. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 240–253.

NAISH, L. 1986. *Negation and Control in Prolog*. Lecture Notes in Computer Science, vol. 238. Springer-Verlag, Berlin.

NAISH, L. 1992. Coroutining and the construction of terminating logic programs. Tech. Rep. 92/5, Department of Computer Science, University of Melbourne.

SARASWAT, V. A. 1989. Concurrent constraint programming languages. Ph.D. thesis, Carnegie-Mellon University.

SARASWAT, V. A., RINARD, M., AND PANANGADEN, P. 1991. Semantic Foundation of Concurrent Constraint Programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 333–353.

VAN EMDEN, M. H. AND KOWALSKI, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23*, 4, 733–742.

YELICK, K. AND ZACHARY, J. 1989. Moded type systems for logic programming. In *Proc. Sixteenth POPL*. ACM Press, 116–124.