Combining Logic and Control to Characterize Global Invariants of Prolog Programs

Livio Colussi and Massimo Marchiori,* Elena Marchiori,†

Abstract

The behaviour of logic programs (with various built-in's) is described in terms of sets of substitutions associated with the control points of the program. These sets are defined by means of an operational semantics $\mathcal{O}$, based on a description of unification as predicate transformer. It is shown that $\mathcal{O}$ subsumes the semantics of logic programs consisting of the set of computed substitutions obtained from finite prefixes of SLD-derivations with Prolog selection rule. Moreover $\mathcal{O}$ is used as base semantics for performing dataflow analysis of logic programs.

1 Introduction

The standard view of logic programming is declarative, i.e., a program is a static description of some predicate or function, and there is no reference to the computational mechanism used to compute it. In the declarative interpretation, the meaning of a program is given model-theoretically as the set of all its logical consequences in the *Herbrand domain* of ground atoms ([vEK76]). Nevertheless, despite of its simplicity and elegance, this approach has some drawbacks. The soundness and completeness results relating the declarative and operational semantics show that there is a mismatch between them, in the sense that some programs that are operationally different are identified by the declarative semantics. Moreover, the declarative semantics does not apply when extending logic programs with extra-logical features, which are introduced in the programming language for efficiency reasons. Finally, the standard declarative semantics is not suitable for dataflow analysis, since the latter needs also information on the flow of control.

This last argument is the fundamental motivation for this paper: we are interested in methods for dataflow analysis of logic programs which are systematically derived from the semantics. The aim of this paper is to develop a suitable semantics that allows to reason about properties of logic programs which depend on the execution and on the presence of some built-in relations, like `var`. To this end, it is convenient to understand a logic program from an imperative point of view, where a clause is viewed as a sequence of procedure calls, corresponding to the sequence of atoms in its body. The different nature of logic and imperative programs, due to a different notion of variable and of the basic computational mechanism, does not provide a direct formalization of such a procedural interpretation. As a consequence various approaches have been proposed and a number of semantics for Prolog programs have been introduced to perform dataflow analysis (cf. [AH87]).

This paper does not introduce a new approach: instead, it gives a novel use of techniques originally developed for imperative programs, to describe the meaning of Prolog programs. The novelty amounts to formalize the logic and the control part of a logic program separately, by means of the concepts of predicate transformer and of path, respectively. More precisely, a program is annotated with (labels of) control points, such that each program clause consists, alternatingly, of atoms and control points. Then the flow of control of possible program executions is described in terms of sequences of control points, called *paths*, while a (strongest postcondition) *predicate transformer*, acting on sets of substitutions, is used to describe the declarative behaviour of the program. These formalizations of logic and control are combined in the key notion of *path strongest postcondition* $psp.\pi.\phi$ of a path π with respect to a precondition ϕ to define an operational semantics $\mathcal{O}$. Given a set of substitutions ϕ, describing the initial values of the variables of the goal-clause, $\mathcal{O}(\mathcal{P}, \phi)$ associates with each control point a set of substitutions ψ, describing

*Dip. di Matematica Pura ed Applicata, Via Belzoni 7, 35131 Padova, Italy, e-mail: colussi,max@euler.math.unipd.it,
†CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, e-mail: elena@cwi.nl,

the possible values of the variables of the program when the execution of the goal-clause starting from an initial value reaches that control point. Such ψ is defined as the union of all the $psp.\pi.\phi$, obtained considering all paths π of the program ending in that control point. A characterization of $\mathcal{O}(\mathcal{P},\phi)$ as union $\cup_{k=0}^{\omega}\mathcal{O}(k)$ of the partial semantics $\mathcal{O}(k)$ is given, by means of the least fixpoint of a continuous operator F_ϕ. In $\mathcal{O}(k)$, the meaning of a control point is the union $\cup_\pi psp.\pi.\phi$ obtained considering all paths π whose last component is equal to that control point and whose length is less or equal then k.

It is shown that $\mathcal{O}$ subsumes the semantics of logic programs consisting of the set of computed substitutions obtained from finite prefixes of SLD-derivations with Prolog selection rule.

Then it is shown how $\mathcal{O}$ can be used as base semantics to perform automatically dataflow analysis of logic programs at compile-time. Dataflow information is useful for debugging, code optimization, program transformation and even for correctness proofs. We introduce a practical framework based on the approach of P. and R. Cousot ([CC77], [CC79]) to derive a global invariant for every control point of the program. An abstraction $\mathcal{O}(k)^a$ of the partial semantics $\mathcal{O}(k)$ is given by applying a proper abstraction to the ϕ's, to the path strongest postcondition *psp* and to the operator $\cup$. Then an approximation $\mathcal{O}^a$ of $\mathcal{O}$ is obtained by iterating $\mathcal{O}(k)^a$ from $k=0$ until a fixpoint is reached.

Related work

Various alternative semantics for logic programming languages have been given. Non-standard declarative semantics have been introduced in [FLMP89] and [AMP92], the first to fill the gap between declarative and operational semantics of logic programs, the second to deal also with some metalogical, built-in relations. Our contribution is related to those approaches which employ mathematical concepts and techniques traditionally applied in the imperative setting. In particular, in [dV90], [dBK90] and [dB91] the semantics of various logic programming languages is investigated by distinguishing between the logic and the control part of a program. Since their aim is to compare logic programming languages with various control features, they focus on the study of the control part, by abstracting completely from the logical part, i.e., from the structure of the atom and the semantical concepts of (S)LD-resolution. A number of alternative semantics of logic programs based on control points have been introduced, as base semantics to perform dataflow analysis. In [Mel87] control points are associated with procedure definitions and a semantics is given, that associates a set of (equivalence classes under variance of) logical terms to a control point. This semantics describes a superset of the observable logical terms. In [JS87] only the entry point of clauses are considered and a denotational semantics of Prolog programs is defined. Our approach presents similarities with those of [Bru91] and [Nil90], where the same notion of control point is considered as here. In [Bru91] an SLD-derivation of a goal is described by means of a proof tree, which is then generalized to consider executions of a class of goals. In [Nil90] a transition system is used, where states are traces whose elements are pairs consisting of one control point and one substitution. The separation of the logic from the control part used in our formalization allows to give a simple and elegant description of (partial) derivation and of state by means of the notions of path and substitutions, respectively and it allows to apply directly techniques for dataflow analysis originally developed in the imperative setting (e.g., [CC79]).

Plan of the Paper

The paper is organized as follows. The next section contains some preliminaries, and the definition of strongest postcondition for the unification. In Section 3, the semantics of various built-in's of Prolog is given. In Section 4, the concepts of control point and of path are introduced. In Section 5, the top-down semantics $\mathcal{O}$ is given and its relation with the standard operational semantics of logic programs is investigated. In Section 6, $\mathcal{O}$ is used as base semantics to perform dataflow analysis. Finally Section 7 gives some conclusions. Due to lack of space, proofs are omitted or only sketched.

2 Preliminaries

We consider logic programs with various built-in's, like arithmetic relations such as $<$ or some metalogical relations like `var`. A *program*, denoted by $\mathcal{P}$, is a (finite) set of clauses $H \leftarrow A_1,\ldots,A_n$, together with one goal-clause $\leftarrow B_1,\ldots,B_m$, where H and the A_i's and B_i's are atoms and H is not a built-in. Clauses are denoted by C, D, while G denotes the goal-clause. A clause with empty body is called *unitary clause*. We consider as execution model LD-resolution, i.e., SLD-resolution with Prolog selection rule where at every resolution step the leftmost atom of a goal is chosen. We call a finite prefix of an LD-derivation

a *partial LD-derivation*, and the composition of the substitutions computed in a partial LD-derivation ξ is called the *computed substitution for* ξ. Then $\mathcal{O}_{LD}(\mathcal{P})$ denotes the set of all computed substitutions of partial LD-derivations of the goal-clause of $\mathcal{P}$.

Variables are denoted by u, v, w, x, y, z, functions by f, g, h and terms by r, s, t. Sequences are denoted by $\underline{t}$ or $\langle t_1, \ldots, t_k \rangle$, and $\cdot$ denotes the operator of *concatenation of sequences*. The number of elements of a sequence/set E is denoted by $|E|$. For a syntactic object O, the set of variables that occur in O is denoted by $vars(O)$.

2.1 Substitutions

A *substitution* is a mapping from variables to terms such that its *domain*, $dom(\vartheta) \stackrel{\text{def}}{=} \{v \mid v\vartheta \neq v\}$, is finite. For a set of variables X, the *restriction of* ϑ *to* X, denoted by $\vartheta_{|X}$, is the substitution which is obtained from ϑ by restricting its domain to X.

$Subs$ denotes the set of all substitutions, whose elements are denoted by lower-case Greek letters. In particular ϵ is used to denote the substitution with empty domain. A substitution ρ is called *renaming* if there exists ρ' such that $\rho\rho' = \epsilon$. For two syntactic expressions E, E', we call E' a *variant* of E if there is a renaming ρ s.t. $E' = E\rho$.

It is well known that the set 2^{Subs} of subsets of $Subs$ with $(\cap, \cup, Subs, \emptyset)$ is a complete lattice. Elements of 2^{Subs} are denoted by ϕ, ψ. Sometimes we shall write $true$ for $Subs$ and $false$ for $\emptyset$.

We shall make the following assumption on 2^{Subs}.

Assumption 2.1 *Sets of substitutions are closed with respect to renaming, i.e.,* $\alpha \in \phi$ *if and only if* $\alpha\rho \in \phi$, *for every renaming* ρ.

Note that the collection of sets of substitutions which are closed under renaming is again a complete lattice, since closedness under renaming is preserved by both intersection and union.

Below some formulas are given, which describe sets of substitutions that will be used in the later sections. Here Var denotes the set of variables, E denotes an expression, $\underline{x} = \underline{t}$ is a shorthand for $x_1 = t_1 \cap \ldots \cap x_k = t_k$, and $mgu(s,t)$ denotes the set of idempotent most general unifiers of s and t.

$$
\begin{array}{lcl}
free(x) & \stackrel{\text{def}}{=} & \{\alpha \mid x\alpha \in Var \text{ and } x\alpha \notin vars(y\alpha) \text{ for all } y \in dom(\alpha), \text{ with } y \neq x\};\\
var(x) & \stackrel{\text{def}}{=} & \{\alpha \mid x\alpha \in Var\};\\
nonvar(x) & \stackrel{\text{def}}{=} & \{\alpha \mid x\alpha \notin Var\};\\
inst(x,y,z) & \stackrel{\text{def}}{=} & \{\alpha \mid x\alpha = y\alpha\mu, \text{ where } \mu \in mgu(y\alpha, z\alpha)\};\\
gae(x) & \stackrel{\text{def}}{=} & \{\alpha \mid x\alpha \text{ is a ground arithmetic expression}\};\\
x < y & \stackrel{\text{def}}{=} & \{\alpha \mid x\alpha, y\alpha \text{ are ground arithmetic expressions and } x\alpha < y\alpha\};\\
(t_1 = t_2) & \stackrel{\text{def}}{=} & \{\alpha \mid t_1\alpha = t_2\alpha\};\\
\exists x\phi & \stackrel{\text{def}}{=} & \{\alpha \mid \text{ there exists } \beta \in \phi \text{ s.t. } y\alpha = y\beta \text{ for all } y \neq x\};\\
\exists \overline{E}\phi & \stackrel{\text{def}}{=} & \{\alpha \mid \text{ there exists } \beta \in \phi \text{ s.t. } x\alpha = x\beta \text{ for all } x \in vars(E)\}.
\end{array}
$$

2.2 Unification

In the imperative setting, the basic statements of a programming language can be described by means of a mapping $\tau : 2^{State} \to 2^{State}$ acting on sets of states ([Flo67], [Dij76], [dB80]). This map is called predicate transformer because a set of states can be represented by means of a predicate of a suitable assertion language. In logic programming, the basic computational mechanism is unification ([Her30], [Rob65]). As observed in [CM93], (elementary) unification can be viewed as a predicate transformer, where states are replaced with substitutions. Formally, for a pair $U = (A, B)$ of atoms define a *unifier of* U to be a substitution ϑ such that $A\vartheta = B\vartheta$. A *most general unifier* for U is a unifier ϑ such that $\vartheta\gamma = \sigma$ for every other unifier σ, for some substitution γ. The *set of idempotent most general unifiers* for U is denoted by $mgu(U)$. Notice that substitutions in $mgu(U)$ are equivalent up to renaming, i.e., for every $\alpha, \beta \in mgu(U)$, there exists a renaming ρ such that $\alpha\rho = \beta$. For a pair $U = (A, B)$ of atoms, unification of instances $U\alpha = (A\alpha, B\alpha)$ of U can be described by a map transforming a substitution into a set of substitutions,

$$\lambda\alpha.\ \{\alpha\mu \mid \mu \in mgu(U\alpha)\}.$$

By applying this function to sets of substitutions, unification can be described as a predicate transformer $sp.U : 2^{Subs} \to 2^{Subs}$ defined as follows.

Definition 2.2 (Strongest Postcondition of U wrt ϕ) For a pair U of atoms and for ϕ in 2^{Subs},

$$sp.U.\phi \stackrel{\text{def}}{=} \{\alpha\mu \mid \alpha \in \phi \text{ and } \mu \in mgu(U\alpha)\}.$$

□

The name *strongest postcondition* is used for the following reason: suppose we have an assertion language, of which the assertions are interpreted as sets of substitutions, and where implication is interpreted as set inclusion. Suppose that an assertion is true if its interpretation equals the set $Subs$. Then ϕ is a precondition for U and $sp.U.\phi$ is the strongest postconditions (w.r.t. implication) that holds after the unification of U with respect to the precondition ϕ.

The following section deals with the semantics of some built-in relations, whose behaviour does not depend on the program in which they occur.

3 Predicate Transformer Semantics of Built-in's

We consider here the built-in's of Prolog that include the arithmetic relations like `<`, and some metalogical relations like `var` and `nonvar`. The semantics of a logic program is influenced by the presence of such predefined relations (see e.g. [AMP92] for an extended discussion on the complications that they cause). However, the meaning of such built-in's is independent of the program in which they occur. Therefore we can describe their semantics by means of a (strongest postcondition) predicate transformer sp, which computes the strongest postcondition $sp.A.\phi$ of a built-in atom A with respect to the precondition ϕ. Table 1 contains the semantics of some of these built-in's. Notice that here sp takes a built-in atom as argument while in the previous section it takes a pair of atoms. It will be clear by the context which sp is intended.

$sp.$ `var(x).` ϕ	$\stackrel{\text{def}}{=}$	$\phi \cap var(x)$
$sp.$ `nonvar(x).` ϕ	$\stackrel{\text{def}}{=}$	$\phi \cap nonvar(x)$
$sp.$ `x < y.` ϕ	$\stackrel{\text{def}}{=}$	$\phi \cap (x < y)$
$sp.$ `x is y.` ϕ	$\stackrel{\text{def}}{=}$	$\exists x'(\exists x(\phi \cap x = x') \cap gae(y) \cap inst(x, x', y))$

Table 1: Semantics of some built-in relations

Example 3.1 Using Table 1, one obtains the following equalities for the built-in `var` :
$sp.$ `var(x).` $var(x) = var(x)$,
$sp.$ `var(x).` $nonvar(x) = nonvar(x) \cap var(x) = \emptyset$,
$sp.$ `var(x).` $true = var(x)$. □

Notice that the definition of predicate transformer semantics can be extended to include not only built-in relations, but also *constraints*. Indeed, one can consider a constraint as representing the set of substitutions which satisfies it, and define the strongest postcondition of a constraint A with respect to a precondition ϕ as the set of substitutions α which are in ϕ and which satisfy A. In such a way the semantics of logic programs defined in the following section can be easily extended to logic programs containing constraints (cf. [JM94]).

4 Control Points and Paths

In the procedural interpretation of logic programming, a clause is viewed as a sequence of procedure calls, corresponding to the sequence of atoms of its body. This can be formalized by viewing a clause $C : H \leftarrow A_1, \ldots, A_n$ as a sequence consisting alternatingly of control points $C(i-1)$ and atoms A_i of the body of C,

$$H \leftarrow C(0)\, A_1,\, C(1)\, \ldots,\, C(n-1)\, A_n\, C(n).$$

ending with a control point $C(n)$. Control points are described by means of natural numbers. We suppose that $C(0), \ldots, C(n)$ are ordered progressively, i.e., $C(i+1) = C(i)+1$ for $i \in \{1, \ldots, n-1\}$, that distinct clauses of a program are decorated with different control points and that the control points of a program form an initial segment $\{1, 2, \ldots, n\}$ of the natural numbers, where 1 denotes the entry point of the goal-clause. $C(0)$ and $C(n)$ are called the *entry point* and *exit point* of C, denoted also by $en(C)$ and $ex(C)$, respectively. For an atom A_i in the body of a (goal-)clause C, $C(i-1)$ and $C(i)$ are called the *calling point* and *success point* of A_i, denoted also by $c(A_i)$ and $s(A_i)$, respectively. It is convenient to assume that all atoms occurring in a program are distinct (one can alternatively use some unambiguous way to refer to occurrences of atoms in the program). $G(0)$ is called the *entry point of the program*. Notice that $C(0)$ is both the entry point of C and the calling point of A_1 and $C(n)$ is both the exit point of C and the success point of A_n. Moreover if C is a unitary clause (i.e., with empty body), then $C(0)$ is both the entry point and the exit point of C.

Example 4.1 The following program *Length* is explicitly labelled with its control points.

```
G:  ← 1 length(u,v) 2 .
C1:  length([x|y],z) ← 3 length(y,z1), 4 z is z1+1 5 .
C2:  length([ ],0) ← 6 .
```

Then 1 denotes the entry point of *Length*. The control point 4 is both the success point s(`length(y,z1)`) of `length(y,z1)` and the calling point c(`z is z1+1`) of `z is z1+1`. The control point 3 is both the entry point $en(C1)$ of $C1$ and the calling point c(`length(y,z1)`) of `length(y,z1)`; 5 is both the exit point $ex(C1)$ of $C1$ and the success point s(`z is z1+1`) of `z is z1+1`. Finally 6 is both the entry point $en(C2)$ and the exit point $ex(C2)$ of the clause $C2$. □

Partial LD-derivations of a program can be described abstractly as follows by means of sequences of control points called paths.

Definition 4.2 (Path) A *path of a program* $\mathcal{P}$ is a sequence $\langle p_1, \ldots, p_m \rangle$ of control points of $\mathcal{P}$, with $m \geq 1$, such that p_1 is a calling point and, for every $k \in [1, m-1]$, one of the following conditions is satisfied:

(i) $p_k = c(A)$, A not built-in, and $p_{k+1} = en(C)$, such that A and a variant of the head H of C are unifiable;

(ii) $p_k = ex(C)$, $p_{k+1} = s(A)$, A not built-in, and there exists $k' < k$ s.t.

- $p_{k'} = c(A)$ (that is $p_{k'} = p_{k+1} - 1$) and
- for every atom B occurring in $\mathcal{P}$

$$|\{j \ : \ k' < j \leq k, p_j = c(B)\}| = |\{j \ : \ k' < j \leq k, p_j = s(B)\}|;$$

(iii) $p_k = c(A)$ and $p_{k+1} = s(A)$ (that is $p_{k+1} = p_k + 1$), A built-in.

A *path from i to j* is a path $\langle p_1, \ldots, p_m \rangle$ such that $p_1 = i$ and $p_m = j$. The set of paths from 1 to i is denoted by $path(i)$; the k-th element of a path π is denoted by π_k, and the number of elements of a path π is called *length of* π, denoted by $|\pi|$. □

So paths describe possible but not necessary semantically possible partial derivations. In the above definition, condition (i) is satisfied when A is an atom of the body of a clause, H is the head of a variant of a clause and A and H are unifiable. Condition (ii) ensures that when the exit point of a clause C is followed by the success point of an atom A, then $c(A)$ was previously reached as well as every control point of all the clauses actived from the call of A until the exit of C. Finally, condition (iii) describes the execution of a built-in, which is independent of the program in which it occurs.

Example 4.3 Consider again the program *Length*. Then $\langle 1,\ 3,\ 3\rangle$ is a path from 1 to 3. $path(6) = \{\langle 1,\ 6\rangle, \langle 1,\ 3,\ 6\rangle, \langle 1,\ 3,\ 3,\ 6\rangle, \ldots\}$ is the set of paths to 6. $\langle 1,\ 3,\ 6,\ 4,\ 5,\ 2\rangle$ is a path from 1 to 2. But the sequence $\pi = \langle 1,\ 3,\ 6,\ 2\rangle$ is not a path: we have that $2 = s(\texttt{length(u,v)})$ and there is only one occurrence of $1 = c(\texttt{length(u,v)})$ in π. The subsequence $\langle 3,\ 6\rangle$ contains $3 = c(\texttt{length(y,z1)})$, but it does not contain $4 = s(\texttt{length(y,z1)})$. So the second condition of (ii) is not satisfied. Also the sequence $\pi = \langle 1,\ 3,\ 6,\ 4,\ 4,\ 2\rangle$ is not a path, because the subsequence $\langle 3,\ 6,\ 4,\ 4\rangle$ contains one occurrence of 3 and two occurrences of 4, thus violating the second condition of (ii). □

5 The Semantics $\mathcal{O}$

In this section, we introduce a top-down semantics $\mathcal{O}$ for logic programs with built-in's, by combining the concept of path with the concepts of strongest postcondition $sp.U$ (of a pair U of atoms) given in Definition 2.2, and of predicate transformer semantics of built-in's $sp.A$ described in Table 1. This combination allows to define the concept of (path) strongest postcondition $psp.\pi.\phi$ of a path π with respect to a set of substitutions ϕ: $psp.\pi.\phi$ represents the result of the symbolic execution of the goal-clause with input ϕ, by using as computation π and as computational mechanism sp. Notice that if π is not a semantical possible derivation, then the result of the symbolic execution of the goal-clause with input ϕ yields the empty set of substitutions. Informally, $psp.\pi.\phi$ is computed by transforming ϕ through π, applying repeatedly the predicate transformer sp. The semantics of a program is then defined with respect to a set ϕ associated with its entry point, by determining for every control point i a set ϕ_i of substitutions, where ϕ_i is obtained by taking the union of all the $psp.\pi.\phi$'s of paths π in $path(i)$.

To define $psp.\pi.\phi$ stepwise the elements of π are processed from left to right until the end of π is reached. To this end an index k is used to indicate the k-th element π_k of π.

To guarantee the independence of the result from the name of the variables, in LD-resolution a technique called *standardization apart* is used, which consists of choosing as input clause a variant of the selected clause which is disjoint with all previous input clauses and with the goal-clause.

For a natural number i, we shall define the i-th variant of a clause or atom. To this end, it is convenient to suppose that clauses of the program $\mathcal{P}$ have disjoint sets of variables and that with every variable x of the program there corresponds a countable set of fresh (i.e., not in $\mathcal{P}$) variables x_i, $i = 1, \ldots$. Then the *i-th variant of an atom A*, written A^i, is the atom obtained by replacing every occurrence of a variable x in A with x_i. The i-th variant C^i of a clause C is defined analogously.

Now, consider the k-th step of the computation of $psp.\pi.\phi$, where the control point π_k is selected. Here we use the index k to guarantee that input clauses are standardized apart: if π_k is the calling point of an atom A and π_{k+1} is the entry point of a clause C, then the k-th variant C^k of C is considered as input clause. Moreover, the j-th variant A^j of A is considered, where if D is the (goal-)clause containing A, then D^j is the variant of D which has been more recently used, among the variants of D which have been used in the previous steps and whose (symbolic) execution is not terminated. Such variants are collected in the set S: then j is equal to the greatest index i such that D^i is in S. If such index is undefined then A^j is taken to be equal to A. Formally, for a clause D, an index k and a set S of indexed variants of clauses, we define

$$call(k, D, S) \stackrel{\text{def}}{=} max_{1 \le i < k}\{i \mid D^i \in S\},$$

the greatest index j below k with which D^j occurs in S. So $call(k, D, S)$ is used to recover the variant of the atom of the program which is actually called (hence the name *call*). Notice that $call(k, D, S)$ is undefined when S does not contain any variant D^i of D. Therefore the following convention is adopted:

$$A^{call(k,D,S)} = \begin{cases} \text{the } call(k, D, S)\text{-th variant of } A \text{ if } call(k, D, S) \text{ is defined} \\ A \text{ otherwise} \end{cases}$$

Thus A^j is equal to $A^{call(k,D,S)}$. So to define $psp.\pi.\phi$, the index k and the set S are used, initialized to 1 and to $\{\ \}$, respectively, and $psp_k.\ \pi.\ (S, \phi)$ is computed stepwise until k becomes equal to $|\pi|$. In the k-th step the actual set of substitutions is transformed using sp; moreover if π_k is the calling point of an atom and π_{k+1} is the entry point of C then C^k is added to the actual set S of variants; if π_k is the exit point of C then $C^{call(k,C,S)}$ is removed from S.

Note that a control point is either the calling point of an atom or it is the exit point of a (goal-)clause. Thus $psp_k.\ \pi.\ (S, \phi)$ can be defined by cases as follows:

- If $k = |\pi|$ then $psp_k.\ \pi.\ (S, \phi) = (S, \phi)$.
- If $k < |\pi|$ then:

  (i) If $\pi_k = c(A)$, A not built-in, and $\pi_{k+1} = en(C)$, with D the clause containing A and H the head of C, then
  $$psp_k.\ \pi.\ (S, \phi) = psp_{k+1}.\ \pi.\ (S', \psi)$$
  where $S' = S \cup \{C^k\}$, $\psi = sp.\ (H^k, A^{call(k,D,S)}).\ (\phi \cap free(vars(C^k))$.

  (ii) If $\pi_k = ex(C)$ then
  $$psp_k.\ \pi.\ (S, \phi) = psp_{k+1}.\ \pi.\ (S', \phi),$$
  where $S' = S \setminus \{C^{call(k,C,S)}\}$.

  (iii) If $\pi_k = c(A)$, A built-in, with D the clause containing A, then
  $$psp_k.\ \pi.\ (S, \phi) = psp_{k+1}.\ \pi.\ (S, \psi),$$
  where $\psi = sp.\ A^{call(k,D,S)}.\ \phi$.

□

We call $psp_k.\ \pi.\ (\{\ \}, \phi)$ *the indexed strongest postcondition of* π *with respect to* ϕ *and* k, also denoted by $psp_k.\ \pi.\ \phi$.

Thus the index k in the definition of $psp_k.\ \pi.\ (S, \phi)$ has two functions: it keeps track of the control point of the path which is active in the symbolic execution, by considering the k-th component of the path, and it is used to guarantee that input clauses are standardized apart, by considering the $(k-1)$-th variant of the chosen clause. Notice also that exit points do not modify sets of substitutions. However exit points are relevant because they guarantee that sequences of calling points of atoms describe correctly the way the corresponding atoms may be called, by updating the set S.

The use of the set S in the computation of $psp_1.\ \pi.\ \phi$ is illustrated in the following example. The abbreviation $p(s_1, \ldots, s_m)$ is used for $p(s_1) \cap \ldots \cap p(s_m)$, with $p \in \{var,\ free,\ ground\}$, where $ground(u)$ denotes the set $\{\alpha \mid vars(u\alpha) = \emptyset\}$.

Example 5.1 The sequence $\pi = \langle 1, 3, 3, 6, 4, 5, 4\rangle$ is a path of the program *Length*.
Let $\phi = (u = [a, b] \cap var(v))$. Then $psp_1.\ \pi.\ \phi$ is computed as follows.

1. from case (i) we have that $psp_1.\ \pi.\ (\{\ \}, \phi) = psp_2.\ \pi.\ (\{C1^1\}, \psi^1)$,
   where $\psi^1 = sp.\ (length([x_1|y_1], z_1), length(u, v)).\ (u = [a, b] \cap var(v) \cap free(x_1, y_1, z_1, z1_1))$. It is not difficult to check that ψ^1 is equal to
   $$(u = [a, b] \cap x_1 = a \cap y_1 = [b] \cap v = z_1 \cap var(z_1, v) \cap free(z1_1)).$$
2. from case (i) we have that $psp_2.\ \pi.\ (\{C1^1\}, \psi^1) = psp_3.\ \pi.\ (\{C1^1, C1^2\}, \psi^2)$,
   where $\psi^2 = sp.\ (length([x_2|y_2], z_2), length(y_1, z1_1)).\ (\psi^1 \cap free(x_2, y_2, z_2, z1_2))$. It is not difficult to check that ψ^2 is equal to
   $$(u = [a, b] \cap x_1 = a \cap y_1 = [b] \cap v = z_1 \cap z1_1 = z_2 \cap x_2 = b \cap y_2 = [\,] \cap var(z_1, v, z1_1, z_2) \cap free(z1_2)).$$
3. from case (i) we have that $psp_3.\ \pi.\ (\{C1^1, C1^2\}, \psi^1) = psp_4.\ \pi.\ (\{C1^1, C1^2\}, \psi^3)$,
   where $\psi^3 = sp.\ (length([\,], 0), length(y_2, z1_2)).\ \psi^2)$. It is not difficult to check that ψ^3 is equal to
   $$(u = [a, b] \cap x_1 = a \cap y_1 = [b] \cap v = z_1 \cap x_2 = b \cap y_2 = [\,] \cap z1_2 = 0 \cap var(z_1, v, z1_1, z_2)).$$
4. from case (ii) we have that $psp_4.\ \pi.\ (\{C1^1, C1^2\}, \psi^3) = psp_5.\ \pi.\ (\{C1^1, C1^2\}, \psi^3)$.
5. from case (iii) we have that $psp_5.\ \pi.\ (\{C1^1, C1^2\}, \psi^3) = psp_6.\ \pi.\ (\{C1^1, C1^2\}, \psi^4)$,
   where $\psi^4 = sp.\ z_2\ is\ z1_2 + 1.\ \psi^3$. It is not difficult to check that ψ^4 is equal to
   $$(u = [a, b] \cap x_1 = a \cap y_1 = [b] \cap x_2 = b \cap y_2 = [\,] \cap v = z_1 \cap z1_1 = z_2 = 1 \cap z1_2 = 0 \cap var(z_1, v)).$$

6. from case (ii) we have that $psp_6.\pi.(\{C1^1, C1^2\}, \psi^4) = psp_7.\pi.(\{C1^1\}, \psi^4)$.

7. from $7 = |\pi|$ we have that $psp_7.\pi.(\{C1^1\}, \psi^4) = (\{C1^1\}, \psi^4)$.

Then $psp_1.\pi.(\{\ \}, \phi)$ is equal to $(\{C1^1\}, \psi^4)$. □

To define the strongest postcondition $psp.\pi.\phi$ of a path π with respect to ϕ, we proceed as follows. Let $psp_1.\pi.\phi$ be equal to (S, ψ). For every clause C of the program, the values described by ψ of the variables of the variant C^j of C are passed into the corresponding variables of C, where C^j is the last input clause used in the computation of $psp_1.\pi.\phi$. The set S is used to determine C^j, namely $C^j = C^{call(|\pi|,C,S)}$. Moreover all the variables which do not occur in the program are hidden. To this end the following operator $[-]_{\mathcal{D}\to\mathcal{C}}$ is introduced, where $\mathcal{C}$ and $\mathcal{D}$ are sets of clauses such that $\mathcal{C}$ contains distinct clauses and $\mathcal{D}$ is obtained considering one indexed variant of every clause C of $\mathcal{C}$. Then for ϕ in 2^{Subs}:

$$[\phi]_{\mathcal{D}\to\mathcal{C}} \stackrel{\text{def}}{=} \left\{ \alpha \mid \exists \beta \in \phi \text{ s.t. } x\alpha = \begin{cases} x_i\beta & x \in vars(C),\ C \in \mathcal{C},\ x_i \in vars(C^i),\ C^i \in \mathcal{D}, \\ x\beta & x \in (vars(\mathcal{P}) \setminus vars(\mathcal{C})). \end{cases} \right\}.$$

Definition 5.2 (Strongest Postcondition of a Path) The *strongest postcondition of π with respect to ϕ* is the set of substitutions

$$psp.\pi.\phi \stackrel{\text{def}}{=} [\psi]_{\mathcal{D}\to\mathcal{C}},$$

with $(S, \psi) = psp_1.\pi.\phi$,
$\mathcal{C} = \{C \mid \pi_k \text{ is a control point of } C \text{ for some } k > 1\}$,
$\mathcal{D} = \{C^{call(|\pi|,C,S)} \mid C \in \mathcal{C}\}$.

□

Example 5.3 Consider the path $\pi = \langle 1,\ 3,\ 3\rangle$ of the program *Length* and the set ϕ of substitutions equal to $ground(u) \cap var(v)$. Then the strongest postcondition $psp.\pi.\phi$ is computed as follows.

1. $psp.\pi.\phi = [psp_1.\pi.\phi]_{\mathcal{D}\to\{C1\}}$;

2. $psp_1.\pi.\phi = psp_2.\pi.(\{C1^1\}, \psi^1)$,
   where $\psi^1 = sp.(length([x_1|y_1], z_1), length(u, v)).(ground(u) \cap var(v) \cap free(x_1, y_1, z_1, z1_1))$. It is not difficult to show that ψ^1 is equal to
   $$(z_1 = v \cap u = [x_1|y_1] \cap ground(u) \cap var(v) \cap free(z1_1));$$

3. $psp_2.\pi.(\{C1^1\}, \psi^1) = psp_3.\pi.(\{C1^1, C1^2\}, \psi^2)$,
   where $\psi^2 = sp.(length([x_2|y_2], z_2), length(y_1, z1_1)).(\psi^1 \cap free(x_2, y_2, z_2, z1_2))$. It is not difficult to show that ψ^2 is equal to
   $$(z_1 = v \cap u = [x_1|y_1] \cap z_2 = z1_1 \cap y_1 = [x_2|y_2] \cap ground(u) \cap var(v, z_2) \cap free(z1_2)).$$

4. $psp_3.\pi.(\{C1^1, C1^2\}, \psi^2) = (\{C1^1, C1^2\}, \psi^2)$.
   Then $psp_1.\pi.\phi = (\{C1^1, C1^2\}, \psi^2)$. Then $\mathcal{D}$ is equal to $\{C1^2\}$.

5. $[\psi^2]_{\{C1^2\}\to\{C1\}}$ is equal to
   $$\exists x_1, y_1\ (ground(u) \cap u = [x_1|y_1] \cap y_1 = [x|y] \cap var(v, z) \cap free(z1)).$$

Then $psp.\langle 1,\ 3,\ 3\rangle.(ground(u) \cap var(v))$ is equal to

$$\exists x_1, y_1\ (ground(u) \cap u = [x_1|y_1] \cap y_1 = [x|y] \cap var(v, z) \cap free(z1)).$$

□

The top-down semantics of a program $\mathcal{P}$ with respect to a set ϕ of substitutions associated with its entry point $G(0)$ is defined below.

Definition 5.4 (Top-Down Semantics of Control Points) Let $\{1,\ldots,n\}$ be the set of control points of $\mathcal{P}$, and ϕ be a set of substitutions. The *top-down semantics* $\mathcal{O}(\mathcal{P},\phi)$ *of* $\mathcal{P}$ *with respect to* ϕ is the tuple $(\phi_1,\ldots,\phi_n)$, where for $i \in \{1,\ldots,n\}$

$$\phi_i \stackrel{\text{def}}{=} \bigcup_{\pi \in path(i)} psp.\pi.\phi.$$

□

Example 5.5 For a program $\mathcal{P}$, there is only one path to $G(0)(=1)$, namely $\langle 1\rangle$. Moreover $psp_1.\langle 1\rangle.\phi = (\{\ \},\phi)$. Hence $\phi_{G(0)} = \exists\overline{X}\,\phi$, with $X = vars(\mathcal{P})$, as expected. □

The following property of indexed strongest postconditions of a path is useful. Notice that the set of paths of a program is closed under non-empty prefixing, i.e., if π is a path, then every non-empty prefix π' of π is also a path.

Lemma 5.6 (Compositionality) *Let* π *be a path in* $path(i)$ *and let* k *be an index such that* $1 \le k < |\pi|$. *Then*

$$psp_k.\pi.\phi = psp_{|\pi'|}.\pi.(psp_k.\pi'.\phi),$$

where $\pi = \pi' \cdot \langle i\rangle$.

Proof. One can prove by induction on $|\pi| - k$ that $psp_k.\,\pi.\,(S,\phi) = psp_{|\pi'|}.\,\pi.\,(psp_k.\,\pi'.\,(S,\phi))$, for every S. Then the result follows by considering $S = \{\ \}$. □

5.1 Bottom-Up Semantics $\mu(F_\phi)$

Now a characterization of the semantics $\mathcal{O}$ by means of the least fixpoint of a suitable continuous operator is given. Consider a program $\mathcal{P}$ and let $\{1,\ldots,n\}$ be the set of control points of $\mathcal{P}$. Let $Path = \cup_{i\in\{1,\ldots,n\}} path(i)$, and let $V(\mathcal{P}) = \cup_{i\ge 0}\mathcal{P}^i$ where for an index i, $\mathcal{P}^i = \{C^i \mid C \text{ clause of } \mathcal{P}\}$ is the set of i-variants of the clauses of $\mathcal{P}$. Then consider the set

$$D = 2^{Path \times V(\mathcal{P}) \times 2^{Subs}}.$$

D with $(\cap,\cup,D,\emptyset)$ is a complete lattice. For a set ϕ of 2^{Subs} define $F_\phi : D \to D$ such that, for $T \in D$:

$$\begin{aligned} F_\phi(T) = \{(\langle 1\rangle,\{\ \},\phi)\} \cup \\ \{(\pi',S',\psi') \mid \text{ there is } (\pi,S,\psi) \text{ in } T \text{ s.t. } \pi' = \pi\cdot\langle j\rangle \text{ is in } path(j) \text{ for some } j, \text{ and} \\ (S',\psi') = psp_{|\pi|}.\pi'.(S,\psi)\}. \end{aligned}$$

One can easily verify that F_ϕ is continuous. Then the least fixpoint $\mu(F_\phi)$ of F_ϕ is equal to $\bigcup_{k=0}^{\omega} F_\phi^k(\emptyset)$.

To characterize $\mathcal{O}$ using $\mu(F_\phi)$ the following lemma is used.

Lemma 5.7 *Let* $k \ge 0$. *Then* $\bigcup_{j=0}^{k} F_\phi^j(\emptyset) = \{(\pi,S,\psi) \mid (S,\psi) = psp_1.\,\pi.\,\phi, \text{with } |\pi| \le k\}$.

Proof. Induction on k, using Lemma 5.6. □

Theorem 5.8 *For* $i \in \{1,\ldots,n\}$, *let* $\mu(F_\phi)_{path(i)}$ *be the set of triples* (π,S,ψ) *of* $\mu(F_\phi)$ *such that* π *is in* $path(i)$. *Let* $\mathcal{O}(\phi,\mathcal{P}) = (\phi_1,\ldots,\phi_n)$. *Then*

$$\phi_i = \bigcup_{(\pi,S,\psi)\in\mu(F_\phi)_{path(i)}} [\psi]_{\mathcal{D}\to\mathcal{C}},$$

where $\mathcal{C} = \{C \mid \pi_k$ *is a control point of* C *for some* $k > 1\}$ *and* $\mathcal{D} = \{C^{call(|\pi|,C,S)} \mid C \in \mathcal{C}\}$.

The following result follows.

Corollary 5.9 *Let* $\mathcal{P}$ *be a program and let* ϕ *be a set of substitutions. Then*

$$\mathcal{O}(\mathcal{P},\phi) = \bigcup_{k=0}^{\omega} \mathcal{O}(k)(\mathcal{P},\phi),$$

where for $i \in \{1,\ldots,n\}$ *the* i*-th component of* $\mathcal{O}(k)(\mathcal{P},\phi)$ *is* $\mathcal{O}(k)(\mathcal{P},\phi)_i \stackrel{\text{def}}{=} \bigcup_{\pi\in path(i),|\pi|\le k} psp.\,\pi.\,\phi$.

5.2 Relation between $\mathcal{O}$ and $\mathcal{O}_{LD}$

We show that $\mathcal{O}$ subsumes $\mathcal{O}_{LD}$. Recall that, for a program $\mathcal{P}$, $\mathcal{O}_{LD}(\mathcal{P})$ is the set of computed substitutions of partial derivations of the goal-clause of $\mathcal{P}$. First, LD-resolvents of built-in's are described in accordance with their predicate transformer semantics. Next, a correspondence between paths and partial derivations is defined: a suitable partial derivation $\xi(\pi)$ is associated with a path π and conversely a suitable path $\pi(\xi)$ is associated with a partial derivation ξ. This correspondence is used to prove that $\mathcal{O}$ subsumes $\mathcal{O}_{LD}$.

LD-resolvents of built-in's are described as follows.

Definition 5.10 Let A be a built-in, let ϕ in 2^{Subs}. For every sequence $B_1, \ldots, B_m$ of atoms, for every α in ϕ, the goal $\leftarrow (A, B_1, \ldots, B_m)\alpha$ has the resolvent $\leftarrow (B_1, \ldots, B_m)\alpha\beta$ if and only if $\alpha\beta \in (sp.A.\phi)$. □

Next, a path $\pi(\xi)$ is constructed from a partial LD-derivation ξ as follows. The indexes j and k are used to indicate the j-th goal ξ_j of ξ and the k-th component π_k of π, respectively, while the index m is used to point to a suitable goal of ξ. Moreover for an atom A of a goal in a partial LD-derivation we use for simplicity $c(A)$ and $s(A)$ to denote, respectively, the calling point and the success point of the atom of the program from which A is obtained.

Definition 5.11 (Path of a Partial Derivation) Let ξ be a partial LD-derivation of G in $\mathcal{P}$. The *path* $\pi(\xi)$ *associated with* ξ is defined by the following algorithm. Initialize m, j and k to 1. Set $\pi(\xi)_1$ to $G(0)$. Repeat the following step until $j = |\xi|$ (recall that $|\xi|$ denotes the length of ξ).

- Set j to $j+1$ and k to $k+1$;
  1. If ξ_j is the *empty clause* then `(update)`.
  2. Otherwise, if ξ_j is obtained using a *unitary clause* C then
     - set $\pi(\xi)_k$ to $en(C)$;
     - `(update)`
     - Set k to $k+1$ and set $\pi(\xi)_k$ to $c(A)$, where A is the leftmost atom of ξ_j.
  3. Otherwise, if ξ_j is obtained by using a *non-unitary clause* C then set $\pi(\xi)_k$ to $en(C)$.
  4. Otherwise, if ξ_j is obtained by resolving a *built-in* B then
     - set $\pi(\xi)_k$ to $s(B)$; if $s(B)$ is the exit point of a clause, then `(update)`;
     - Set k to $k+1$ and $\pi(\xi)_k$ to $c(A)$, where A is the leftmost atom of ξ_j.

where `(update)` is the following step:

let $\langle A_m, \ldots, A_{j-1}\rangle$ be the sequence of the selected (i.e., leftmost) atoms of $\xi_m, \ldots, \xi_{j-1}$. For every $i \in [1, m-j+2]$ set $\pi(\xi)_{k+i}$ to $s(A_{i+j-2})$; set k to $k+(m-j)$ and set m to j. □

Note that $\pi(\xi)$ is well-defined, i.e., it is a path of $\mathcal{P}$. In fact, $\pi(\xi)_1 = G(0)$ and conditions (i),(iii) of Definition 4.2 are immediate. Moreover if a success point of an atom, say A, is introduced, say in $\pi(\xi)_k$, then $\pi(\xi)_{k-1}$ is the exit point of a clause and the calling point of A has been introduced in $\pi(\xi)_{k'}$, for some $k' < k$. Then k' satisfies condition (ii) because of the order in which success points are introduced in $\pi(\xi)$ by means of step `(update)`.

Now, given a path π, a partial LD-derivation $\xi(\pi)$ is constructed, by considering variants of input clauses obtained using π. To construct $\xi(\pi)$ components of π are possibly marked.

Definition 5.12 (Partial Derivation of a Path) Let π be a path of $\mathcal{P}$ such that π_1 is the entry point of $\mathcal{P}$. The following algorithm defines the *partial LD-derivation* $\xi(\pi)$ *associated with* π. Initially no component of π is marked.

- Set the first goal of $\xi(\pi)$ to G. Set k to 0.
- Repeat the following step until $k = |\pi|$ or $\xi(\pi)_k$ is marked as failed:

– Set k to $k+1$. Let A be the selected atom of $\xi(\pi)_k$. If A is a built-in then: if it has no resolvent then mark $\xi(\pi)_k$ as failed, otherwise set $\xi(\pi)_{k+1}$ to be the resolvent of $\xi(\pi)_k$. If A is not a built-in then: let j the least index such that π_j is not marked and $en(C) = \pi_j$; if there is not a resolvent of A and C^j then mark $\xi(\pi)_k$ as failed, otherwise set $\xi(\pi)_{k+1}$ to be the resolvent of $\xi(\pi)_k$ and mark π_j. □

By construction $\xi(\pi)$ is a partial LD-derivation of G in $\mathcal{P}$. Note that $\pi = \pi(\xi(\pi))$.

Theorem 5.13 ($\mathcal{O}$ **subsumes** $\mathcal{O}_{LD}$)

(i) Let π be a path of $\mathcal{P}$ such that π_1 is the entry point of $\mathcal{P}$. Let $psp_1.\pi.\{\epsilon\} = (S, \psi)$ and suppose that η is in ψ. Then η is the computed substitution of $\xi(\pi)$.

(ii) Let ξ be a partial LD-derivation of G in $\mathcal{P}$. Suppose $\xi(\pi(\xi))$ has computed substitution η. Let $psp_1.\pi(\xi).\{\epsilon\} = (S, \psi)$. Then η is in ψ.

Proof.
(i) Induction on $|\pi|$, using Definition 5.10 and Lemma 5.6.
(ii) Induction on the number of goals of $\xi(\pi(\xi))$, using Lemma 5.6. □

As a consequence $\mathcal{O}$ subsumes the model-theoretic semantics by Apt et al. for logic programs with built-in's given in [AMP92] ($\mathcal{M}$ for short), in the following sense.

Corollary 5.14 *Let $\mathcal{P}$ be a program with goal-clause $\leftarrow A$. Consider $\phi_{s(A)}$ in $\mathcal{O}(\mathcal{P}, \{\epsilon\})$. If $\eta \in \phi_{s(A)}$ then $\langle A, \eta_{|vars(A)}\rangle \in \mathcal{M}(\mathcal{P})$. Conversely if $\langle A, \eta\rangle \in \mathcal{M}(\mathcal{P})$ then there exists $\eta' \in \phi_{s(A)}$ s.t. $(\eta')_{|vars(A)} = \eta$.*

Proof. From the soundness and completeness results for $\mathcal{M}$, proven in [AMP92], $\mathcal{M}(\mathcal{P})$ is the set of pairs $\langle A, \eta\rangle$ such that $\leftarrow A$ has an LD-refutation in $\mathcal{P}$ with computed answer substitution η. The result then follows by Theorem 5.13. □

6 Application

Here we illustrate how $\mathcal{O}$ can be used to define an abstract interpretation framework for dataflow analysis of logic programs. Although the original work on abstract interpretation was intended for imperative programs ([CC77]), it has been widely applied to declarative programming languages, due to the generality of its basic scheme (cf. [AH87], [CC92]). First, the concrete domain $Conc$ equipped with a partial order $\subseteq$ is approximated by an abstract domain Abs equipped with a partial order $\sqsubseteq$, such that there is a Galois connection (f_a, f_c) between $Conc$ and Abs, i.e., $f_a : Conc \to Abs$, $f_c : Abs \to Conc$ are total monotone functions such that $f_a f_c(\phi) \sqsubseteq \phi^a$ for all ϕ^a in Abs, and $\psi \subseteq f_c f_a(\psi)$ for all ψ in $Conc$. Next, given a concrete semantics of the program defined as the least fixpoint $\mu(F)$ of an operator F over $Conc$, an abstract interpretation framework computes an element $(\mu F)^a$ of Abs such that $f_a(\mu F) \sqsubseteq (\mu F)^a$.

This approach can be applied to the semantics $\mathcal{O}$, by using the characterization of $\mathcal{O}$ given in Corollary 5.9.

Example 6.1 We illustrate the application of the framework with a simple mode inferencing system. Let $Conc = 2^{Subs}$ and let Abs be the set of assertions of the assertion language $\mathcal{A}$ whose variables are those occurring in the considered program, with $ground(s)$ and $var(s)$ as atomic predicates and with $\wedge$ and $\neg$ as logical operators (also, $\phi^a \vee \psi^a$ is used as an abbreviation for $\neg(\phi^a \wedge \psi^a)$). Consider f_c defined as follows:

$$f_c(\phi^a) = \begin{cases} \{\,\} & \text{if } \phi^a = false \\ Subs & \text{if } \phi^a = true \\ \{\alpha \mid s\alpha \text{ is a ground term }\} & \text{if } \phi^a = ground(s) \\ \{\alpha \mid s\alpha \text{ is a variable }\} & \text{if } \phi^a = var(s) \\ f_c(\phi_1^a) \cap f_c(\phi_2^a) & \text{if } \phi^a = \phi_1^a \wedge \phi_2^a \\ Subs \setminus f_c(\phi_1^a) & \text{if } \phi^a = \neg\phi_1^a \end{cases}$$

Recall that for $i \in \{1, \ldots, n\}$ $\mathcal{O}(k)_i = \bigcup_{\pi \in path(i), |\pi| \leq k} psp.\pi.\phi$. Then one can define the n-tuple $\mathcal{O}(k)^a = (\mathcal{O}(k)^a_1, \ldots, \mathcal{O}(k)^a_n)$ of assertions of Abs as follows. For a path π and for ϕ^a in Abs consider the assertion $\overline{psp}.\pi.\phi^a \stackrel{\text{def}}{=} SP.(\mathcal{C}, \mathcal{D}).(\psi \wedge free(vars(\mathcal{C}))$, with $\psi = psp_1.\pi.\phi^a$ defined replacing sp with SP in the inductive definition of $psp_k.\pi.(S, \phi)$ given in Section 5. Here SP stands for an algorithm which, for a given pair U of atoms and for an assertion ϕ in Ass, computes an assertion ψ equivalent to $sp.U.\phi$ (equivalent means that the interpretation of ψ as sets of substitutions is equal to $sp.U.\phi$).

Such an algorithm was introduced in [CM93] where the set Ass of assertions there considered contains Abs; moreover for every set ϕ of substitutions used in Table 1 to define the semantics of built-in's, Ass contains an assertion equivalent to ϕ.

Note that the natural extension of SP to pairs of the form $(\mathcal{C}, \mathcal{D})$ is used in the definition of $\overline{psp}$, where $\mathcal{C}$ is a set of distinct clauses and $\mathcal{D}$ is obtained by considering one variant of every clause C of $\mathcal{C}$. Then $SP.(\mathcal{C}, \mathcal{D}).\phi$ computes an assertion equivalent to the set $sp.(\mathcal{C}, \mathcal{D}).\phi$ of substitutions $\alpha\mu$ such that $\alpha \in \phi$ and μ is in $mgu(\mathcal{C}\alpha, \mathcal{D}\alpha)$: a unifier of such a pair is defined as a unifier of all the $(C\alpha, D\alpha)$'s, with C in $\mathcal{C}$ and D the corresponding variant of C in $\mathcal{D}$.

Moreover for a built-in atom A define $SP.A.\phi$ to be equal to the assertion of Ass equivalent to the set of substitutions $sp.A.\phi$, defined using Table 1.

Now, $\overline{psp}.\pi.\phi^a$ is not in general an assertion of Abs, because it can contain also the predicate symbols $free$, gae, $<$, $=$ and $inst$ and the quantifier $\exists$. Then it can be transformed in an element $(\overline{psp}.\pi.\phi^a)^a$ of Abs as follows: every atomic predicate $free(s)$ is transformed in $var(s)$, $gae(s)$ is transformed in $ground(s)$, $s < t$ is transformed in $ground(s) \wedge ground(t)$, and $s = t$ becomes $(ground(s) \Leftrightarrow ground(t)) \wedge (var(s) \Leftrightarrow var(t))$; while the predicate $inst$ and the quantifier $\exists$ are simply deleted as well as the corresponding bounded variables.

Thus $\mathcal{O}(k)^a_i = \bigvee_{\pi \in path(i), |\pi| \leq k} (\overline{psp}.\pi.\phi^a)^a$. Hence the $\mathcal{O}^a$ is obtained by iterating $\mathcal{O}(k)^a$ from $k = 0$ until a fixpoint is reached.

We apply this system for abstract interpretation to the program $Length$:

```
G:  ← 1 length(u,v) 2 .
C1:  length([x|y],z) ← 3 length(y,z1), 4 z is z1+1 5 .
C2:  length([ ],0) ← 6 .
```

Let $\phi = ground(u) \wedge var(v)$. Then $\mathcal{O}(\phi, \mathcal{P})^a$ is computed as follows, where $p(x_1, \ldots, x_n)$ is used as an abbreviation for $p(x_1) \wedge \ldots \wedge p(x_n)$, with $p \in \{var, ground\}$. Denote by $ipath$ the set of paths of length i whose first element is $G(0)$.

- $0path = \emptyset$. Then $\mathcal{O}(0)^a$ is equal to $(false, \ldots, false)$;

- $1path = \{\langle 1 \rangle\}$. Then $\mathcal{O}(1)^a$ is equal to $(ground(u) \wedge var(v), false, \ldots, false)$;

- $2path = \{\langle 1, 3 \rangle, \langle 1, 6 \rangle\}$. Then $\mathcal{O}(2)^a$ is equal to

$(ground(u) \wedge var(v), false, ground(u, x, y) \wedge var(v, z, z1), false, false, ground(u, v))$;

- $3path = \{\langle 1, 3, 3 \rangle, \langle 1, 3, 6 \rangle, \langle 1, 6, 2 \rangle\}$. Then $\mathcal{O}(3)^a$ is equal to

$(ground(u) \wedge var(v), ground(u, v), ground(u, x, y) \wedge var(v, z, z1),$
$false, false, (ground(u, v) \vee ground(u, x, y, z1) \wedge var(v, z))$;

- $4path = \{\langle 1, 3, 3, 3 \rangle, \langle 1, 3, 3, 6 \rangle, \langle 1, 3, 6, 4 \rangle\}$. Then $\mathcal{O}(4)^a$ is equal to

$(ground(u) \wedge var(v), ground(u, v), ground(u, x, y) \wedge var(v, z, z1), ground(u, x, y, z1) \wedge var(v, z),$
$false, (ground(u, v) \vee ground(u, x, y, z1) \wedge var(v, z))$;

- $5path = \{\langle 1, 3, 3, 3, 3 \rangle, \langle 1, 3, 3, 3, 6 \rangle, \langle 1, 3, 3, 6, 4 \rangle, \langle 1, 3, 6, 4, 5 \rangle\}$. Then $\mathcal{O}(5)^a$ is equal to

$(ground(u) \wedge var(v), ground(u, v), ground(u, x, y) \wedge var(v, z, z1), ground(u, x, y, z1) \wedge var(v, z),$
$ground(u, v, x, y, z, z1), (ground(u, v) \vee ground(u, x, y, z1) \wedge var(v, z))$;

- $6path = \{\langle 1, 3, 3, 3, 3, 3 \rangle, \langle 1, 3, 3, 3, 3, 6 \rangle, \langle 1, 3, 3, 3, 6, 4 \rangle, \langle 1, 3, 3, 6, 4, 5 \rangle, \langle 1, 3, 6, 4, 5, 2 \rangle\}$. Then $\mathcal{O}(6)^a$ is equal to

$(ground(u) \wedge var(v), (ground(u, v) \vee ground(u, v, x, y, z, z1)), ground(u, x, y) \wedge var(v, z, z1),$
$ground(u, x, y, z1) \wedge var(v, z), ground(u, v, x, y, z, z1), (ground(u, v) \vee ground(u, x, y, z1) \wedge var(v, z))$;

- $\mathcal{O}(7)^a = \mathcal{O}(6)^a$.

Then $\mathcal{O}^a(\phi, \mathcal{P})$ is equal to $\mathcal{O}(6)^a$.

□

7 Conclusion

In this paper we studied the behaviour of a logic program with various built-in's by means of invariants associated with its control points, by introducing the novel concepts of path and of strongest postcondition of a path with respect to a precondition. We developed an operational model $\mathcal{O}$ and proved that $\mathcal{O}$ subsumes the semantics of logic programs consisting of the set of computed substitutions of finite prefixes of LD-derivations. Next, we derived from $\mathcal{O}$ an abstract interpretation framework for dataflow analysis. The quality of a system for an abstract interpretation is based on the precision of the information the system gives and on its efficiency. The art of abstract interpretation consists in finding systems where both requirements are sufficiently fulfilled. The system for mode inference sketched in the final example of this paper is good in precision but not very efficient, due to the use of the algorithm SP. So, it remains to be investigated how good systems for abstract interpretation can be obtained from our framework.

Another interesting topic for future research is to use the semantics to study termination of logic programs with respect to a precondition. Moreover we would like to investigate the use of a *weakest precondition* predicate transformer wp to define the semantics of a logic program with respect to a set of output substitutions, by using the notion of path. Finally, how this semantics could be extended to describe the behaviour of logic programs containing negation is an open question.

The contribution of this paper should be considered as a novel use of well-known techniques from the imperative setting to describe the meaning of a logic program. Its relevance is both theoretical and practical: from the theoretical point of view, we showed that, despite of the different nature of logic and imperative programs, due to a different notion of variable and of the basic computational mechanism, logic programs naturally support programming styles and techniques originally developed for imperative programs. From the practical point of view, we provided a suitable base semantics to perform dataflow analysis of logic programs.

Acknowledgments We would like to thank Jan Rutten for useful comments on the exposition. The research of E. Marchiori was partly supported by the ESPRIT Basic Research Action 6810 (Compulog 2).

References

[AH87] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of declarative languages*, pages 9–31. Ellis Horwood, 1987.

[AMP92] K.R. Apt, E. Marchiori, and C. Palamidessi. A theory of first order built-in's of Prolog. Report CS-R9216, CWI, Amsterdam, 1992.

[Bru91] M. Bruynooghe. A practical framework for abstract interpretation of logic programs. *J. Logic Programming*, 10(2):91–124, 1991.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–251, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming*, pages 13: 103–180, 1992.

[CM93] L. Colussi and E. Marchiori. Unification as predicate transformer. *Submitted*, 1993. Preliminary version in Proceedings JICSLP' 92, 67-85.

[dB80] J.W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[dB91] J.W. de Bakker. Comparative semantics for flow of control in Logic Programming. *Information and Computation*, 94:2:123–179, 1991.

[dBK90] J.W. de Bakker and J. N. Kok. Comparative metric semantics of concurrent Prolog. *Theoretical Computer Science*, 75:15–43, 1990.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[dV90] E.P. de Vink. Comparative metric semantics of concurrent prolog. *Science of Computer Programming*, 13:237–264, 1990.

[FLMP89] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behaviour of logic languages. *Theoretical Computer Science*, 69:289–318, 1989.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings Symposium on Applied Mathematics, 19, Math. Aspects in Computer Science*, pages 19–32. American Society, 1967.

[Her30] J. Herbrand. *Recherches sur la theorie de la demonstration.* Universite de Paris (These), 1930.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. Logic Programming*, 1994. to appear.

[JS87] N.D. Jones and H. Sondegaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of declarative languages*, pages 123–142. Ellis Horwood, 1987.

[Mel87] C. Mellish. Abstract interpretation of Prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of declarative languages*, pages 181–198. Ellis Horwood, 1987.

[Nil90] U. Nilsson. Systematic semantics approximations of logic programs. In *Proceedings of PLILP '90.* Springer-Verlag, 1990.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, 1965.

[vEK76] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.