# SEQUENCE-BASED ABSTRACT INTERPRETATION OF PROLOG[1]

BAUDOUIN LE CHARLIER, SABINA ROSSI, AND PASCAL VAN HENTENRYCK

▷

Abstract interpretation is a general methodology for systematic development of program analyses. An abstract interpretation framework is centered around a parametrized non-standard semantics that can be instantiated by various domains to approximate different program properties.
Many abstract interpretation frameworks and analyses for Prolog have been proposed, which seek to extract information useful for program optimization. Although motivated by practical considerations, notably making Prolog competitive with imperative languages, such frameworks fail to capture some of the control structures of existing implementations of the language. In this paper we propose a novel framework for the abstract interpretation of Prolog which handles the depth-first search rule and the cut operator. It relies on the notion of *substitution sequence* to model the result of the execution of a goal. The framework consists of (i) a denotational concrete semantics, (ii) a safe abstraction of the concrete semantics defined in terms of a class of post-fixpoints, and (iii) a generic abstract interpretation algorithm. We show that traditional abstract domains of substitutions may easily be adapted to the new framework, and provide experimental evidence of the effectiveness of our approach. We also show that previous work on determinacy analysis, that was not expressible by existing abstract interpretation frameworks, can be seen as an instance of our framework.
The ideas developed in this paper can be applied to other logic languages, notably to constraint logic languages, and the theoretical approach should be of general interest for the analysis of many non-deterministic programming languages.

◁

[1]Preliminary versions of this work appeared in the Proceedings of ILPS'94, Ithaca, NY [6, 58].
*Address correspondence to* Baudouin Le Charlier, Institut d'Informatique, University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium. E-mail: `ble@info.fundp.ac.be`

## 1. INTRODUCTION

*Abstract interpretation* [19] is a general methodology for systematic development of program analyses. It has been applied to various formalisms and paradigms including flow-charts and imperative, functional, logic, and constraint programming.

Abstract interpretation of Prolog and, more generally, of logic programming was initiated by Mellish [71] and further developed by numerous researchers (e.g., [8, 21, 47, 53, 69]). Many different kinds of practical analyses and optimizations have been proposed, a detailed description of which can be found in [21, 37]. Briefly, mode [16, 27, 29, 83], type [2, 18, 36, 44, 49, 50, 51, 63, 77, 96, 97], and aliasing [12, 42] analyses collect information about the state of variables during the execution and are useful to speed up term unification and make memory allocation more efficient [41, 94]. Sharing analysis [14, 15, 52, 75] is similar to aliasing except that it refers to the sharing of memory structures to which program variables are instantiated; it is useful to perform compile-time garbage collection [45, 52, 74] and automatic parallelization [10, 11, 38, 43]. Reference chain analysis [66, 92] attempts to determine an upper bound to the length of the pointer chain for a program variable. Trailing analysis [88] aims at detecting variables which do not need to be trailed. Liveness analysis [73] determines when memory structures can be reused. All these analyses approximate the set of values (i.e., terms or memory structures) to which program variables can be instantiated at some given program point.
It is thus not surprising that almost all frameworks for the abstract interpretation of Prolog (e.g., [3, 8, 47, 67, 69, 71, 78]) are based on abstractions of sets of substitutions. Such frameworks ignore important control features of the language, like the depth-first search strategy and the cut operator. Indeed, the latter are difficult to model accurately, and yet not strictly necessary for a *variable level* analysis. However, modeling Prolog control features has two main advantages. First, it allows one to perform so-called *predicate level* analyses, like determinacy [31, 39, 81, 89, 91] and local stack [65, 70] analyses. These analyses are not captured by traditional abstract interpretation frameworks; they usually rely on some ad hoc technique and require special-purpose proofs of correctness (e.g., [30, 81]). They are useful to perform optimizations, such as the choice point removal and the simplification of environment creation. Second, the analysis of some classes of programs, like programs containing multi-directional procedures which use cuts and meta-predicates to select among different versions, may be widely improved. The compiler may be allowed to perform various important optimizations such as dead-code elimination.

Abstract interpretation of Prolog with control has been investigated by other authors. In particular, we know of three main different approaches. The approach of R. Barbuti *et al.* [4] is based on an abstract semantics for logic programs with control which is parametric with respect to a "termination theory". The latter is intended to be provided from outside, for instance by applying proofs procedures. G. Filé and S. Rossi [34] propose an operational and non-compositional abstract interpretation framework for Prolog with cut consisting of a tabled interpreter to visit OLDT abstract trees decorated with information about sure success or failure of goals. Finally, F. Spoto and G. Levi [84] define an abstract goal-independent denotational semantics for Prolog handling control rules and cut. Program denotations are adorned with "observability" constraints giving information about divergent computations and cut executions. We know of no experimental results validating the effectiveness of these approaches.

In this paper we present a novel abstract interpretation framework for Prolog which models the depth-first search rule and the cut operator. It relies on the notion of *substitution sequence* which allows us to collect the solutions to a goal together with information such as sure success and failure, the number of solutions, and/or termination. The framework that we propose can be applied to perform predicate level analyses, such as determinacy, which were not expressible by classical frameworks, and can be also used to improve the accuracy of existing analyses. Experiments on a sample analysis, namely cardinality analysis, will be discussed.

### *1.1. Some Motivating Examples*

In this section we illustrate by means of small examples the functionality of our static analyzer and we discuss how it improves on previous abstract interpretation frameworks. Experimental results on medium-size programs will be reported later.

The first two examples show that predicate level properties, such as determinacy, which are out of the scope of traditional abstract interpretation frameworks can be captured by our analyzer. To the best of our knowledge, does not exist any specific analysis which can infer determinacy of all the programs discussed below.

Consider first the procedure `is_last`:

```
is_last(X,[X]).
is_last(X,[_|T]) :- is_last(X,T).
```

When given the input pattern `is_last(var,ground)`, where `var` and `ground` denote the set of all variables and the set of all ground terms respectively, our analysis returns the abstract sequence ⟨`is_last(ground,[ground|ground])`,`0`,`1`,`pt`⟩, where `is_last(ground,[ground|ground])` is the pattern characterizing the output substitutions, 0 and 1 are, respectively, the minimum and the maximum number of returned output substitutions, and `pt` stands for "possible termination".

Consider now the following two versions of the procedure `partition`.

```
partition([],P,[],[]).
partition([S|T],P,[S|Ss],Bs) :- S ≤ P, !, partition(T,P,Ss,Bs).
partition([B|T],P,Ss,[B|Bs]) :- partition(T,P,Ss,Bs).
```

```
partition([],P,[],[]).
partition([S|T],P,[S|Ss],Bs) :- leq(S,P), partition(T,P,Ss,Bs).
partition([B|T],P,Ss,[B|Bs]) :- gt(B,P), partition(T,P,Ss,Bs).
leq(K1-V1,K2-V2) :- K1 ≤ K2.
gt(K1-V1,K2-V2) :- K1 > K2.
```

Note that the second version of the procedure calls arithmetic predicates through an auxiliary predicate and is appropriate for a key sort. Given an input pattern `partition(ground,ground,var,var)`, our analysis returns in both cases the abstract sequence ⟨`partition(ground,ground,ground,ground)`,`0`,`1`,`pt`⟩. Input/output patterns are used to determine that the first clause and the two others are mutually exclusive in both programs, while the cut (in the first version) and the abstraction of arithmetic predicates (in the second version) determine the mutual exclusion of the second and the third clause. Thus we can infer determinacy of both versions of the procedure `partition`.

As stated above, we don't know of any static analysis for logic programs which can

infer determinacy of all these programs. For instance, the analysis developed by Debray and Warren in [30] to detect functional computations of a logic program cannot infer determinacy of the procedure `is_last`; the analyses proposed by Dawson *et al.* in [25] and by Giacobazzi and Ricci in [39] cannot handle the first version of the procedure `partition`, since they do not deal with the cut; and the cardinality analysis defined by Sahlin in [81] cannot handle any of the examples discussed above since it ignores predicate arguments. Finally, we know of no implemented system that can handle the second version of the procedure `partition`.

The next example shows that the use of abstract sequences can improve on the analysis of variable level properties such as modes.

Consider the procedure `compress(L,Lc)`, which relates two lists `Lc` and `L` such that `Lc` is a compressed version of `L`. For instance, the compressed version of `[a, b, b, c, c, c]` is `[a, 1, b, 2, c, 3]`. A library can contain the definition of a single procedure to handle both compression and decompression as follows.

```
compress(A,B) :- var(A), !, decomp(A,B).
compress(A,B) :- comp(A,B).

comp([],[]).
comp([C],[C,1]).
comp([C1,C2|T],[C1,1,C2,N|Rest]):-C1<>C2, comp([C2|T],[C2,N|Rest]).
comp([C1,C1|T],[C1,N1|Rest]) :- comp([C1|T],[C1,N|Rest]), N1 := N + 1.

decomp([],[]).
decomp([C],[C,1]).
decomp([C1,C2|T],[C1,1,C2,N|Rest]):-decomp([C2|T],[C2,N|Rest]),C1<>C2.
decomp([C1,C1|T],[C1,N1|Rest]):-N1>1,N:=N1-1,
                                decomp([C1|T],[C1,N|Rest]).
```

Given the input patterns `compress(ground,var)` and `compress(var,ground)`, our analysis returns the abstract sequence ⟨`compress(ground,ground),0,1,pt`⟩ for both the inputs. This example illustrates many of the functionalities of our system, including input/output patterns, abstraction of arithmetic and meta-predicates, and the cut, all of which are necessary to obtain the optimal precision. In addition, it shows that taking the cut into account improves the analysis of modes. Indeed, a mode analysis ignoring the cut would return the output pattern `compress(novar,ground)` for the input pattern `compress(var,ground)`, losing the groundness information. None of the abstract interpretation algorithms for logic programs we know of can handle this example with an optimal result. Moreover, if a program only uses the input pattern `compress(var,ground)`, our analysis detects that the second clause of `compress` is dead code without any extra processing since no input/output pattern exists for `comp`. The second clause, the test `var`, and the cut of the first clause can then be removed by an optimizer.

Notice that there exist implemented tools for the static analysis of Prolog programs, such as PLAI [76], which can achieve as accurate success and dead-code information as our analyzer. However, such tools usually integrate several analyses based on different techniques which are not all justified by the abstract interpretation framework. The example of the procedure `compress` shows that our analyzer can handle control features of the language within the abstract interpretation framework without the need of any extra consideration.

### *1.2. Sequence-Based Abstract Interpretation of Prolog*

An abstract interpretation framework [22] is centered around the definition of a non-standard semantics approximating a concrete semantics of the language.

Most top-down abstract interpretation frameworks for logic programs [8, 13, 47, 61, 68, 71, 76, 78, 93, 95] can be viewed as abstractions of a concrete structural operational semantics [79]. Such a semantics defines the meaning of a program as a *transition relation* described in terms of transition rules of the form $\langle \theta, o \rangle \longmapsto \theta'$, where the latter expresses the fact that $\theta'$ is a possible output from the execution of the construct $o$ (i.e., a procedure, a clause, etc.) called with input $\theta$. This structural operational semantics can easily be rephrased as a fixpoint semantics mapping any input pattern $\langle \theta, o \rangle$ to the set of all corresponding outputs $\theta'$. The fixpoint semantics can then be lifted to a *collecting semantics* that maps sets of inputs to sets of outputs and is defined as the least fixpoint of a set-based transformation. The *non-standard* (or *abstract*) *semantics* is identical to the collecting one except that it uses abstract values instead of sets and abstract operations instead of operations over sets. Finally, an abstract interpretation algorithm can be derived by instantiating a generic fixpoint algorithm (e.g., [60]) to the abstract semantics.

The limitations of traditional top-down frameworks for Prolog stem from the fact that structural operational semantics are unable to take the depth-first search rule into account. Control operators such as the cut cannot be modeled and are thus simply ignored. To overcome these limitations, we propose a concrete semantics of Prolog which describes the result of program executions in terms of substitution sequences. This allows us to model the depth-first search rule and the cut operator. The semantics is defined in the denotational setting to deal with sequences resulting from the execution of infinite computations. Moreover, it is still compositional allowing us to reuse most of the material of our previous works, i.e., the abstract domains and the generic algorithm [61]. However, technical problems arise when applying the abstract interpretation approach described above. Let us informally explain the main ideas behind the definition of our framework.

First, we define a *concrete semantics* as the least fixpoint of a concrete transformation $TCB$ mapping every so-called concrete behavior $\longmapsto$ to another concrete behavior $\overset{TCB}{\longmapsto}$. The notion of concrete behavior is our denotation choice for a Prolog program: it is a function that maps pairs of the form $\langle \theta, p \rangle$ to a substitution sequence $S$, which intuitively represents the sequence of computed answer substitutions returned by the query $p(x_1, \ldots, x_n)\theta$. The fixpoint construction of the concrete semantics relies on a suitable ordering $\sqsubseteq$ defined on sequences.

Second, a collecting transformation $TCD$ is obtained by lifting the concrete transformation $TCB$ to sets of substitutions and sets of sequences. The transformation $TCD$ is monotonic with respect to set inclusion. However, its least fixpoint does not safely approximate the concrete semantics. In fact, the least set with respect to inclusion, that is the empty set $\{\}$, does not contain the least substitution sequence with respect to $\sqsubseteq$, which is a special sequence denoted by $< \bot >$. The problem relies on the fact that an ordering on sets of sequences that "combines" both the ordering $\sqsubseteq$ on sequences and the ordering $\subseteq$ on sets is needed. This is an instance of the power domain construction problem [82], which is difficult in general.
We choose a more pragmatic solution which consists in restricting to *chain-closed* sets of sequences, i.e., sets containing the limit of every increasing chain, with respect to $\sqsubseteq$, of their elements. We also introduce the notion of *pre-consistent*

collecting behavior which, roughly speaking, contains a lower approximation, with respect to $\sqsubseteq$, of the concrete semantics (the least fixpoint of *TCB*). The transformation *TCD* maps pre-consistent collecting behaviors to other pre-consistent ones. Moreover, assuming that sets of sequences are *chain-closed*, any pre-consistent post-fixpoint, with respect to set inclusion, of *TCD* safely approximates the concrete semantics. These results imply that a safe collecting behavior can be constructed by iterating on *TCD* from any initial pre-consistent collecting behavior and by applying some widening techniques [23] in order to reach a post-fixpoint.

Third, the *abstract semantics* is defined exactly as the collecting one except that it is parametric with respect to the abstract domains. In fact, we do not explicitly distinguish between the collecting and the abstract semantics: in our presentation, the collecting transformation *TCD* is just a particular instance of the (generic) abstract transformation *TAB*.

Finally, a generic abstract interpretation algorithm is derived from the abstract semantics. The algorithm is essentially an instantiation of the universal fixpoint algorithm described in [60].

### *1.3. Plan of the Paper*

The paper is organized as follows. Section 2 and Section 3 describe, respectively, our concrete and abstract semantics for pure Prolog augmented with the cut. The generic abstract interpretation algorithm is discussed in Section 4. Section 5 is a revised and extended version of [6]. It describes an instantiation of our abstract interpretation framework to approximate the number of solutions to a goal. Experimental results are reported. In Section 6 we consider related works on determinacy analysis. Section 7 concludes the paper.

## 2. CONCRETE SEMANTICS

This section describes a concrete semantics for pure Prolog augmented with the cut. The concrete semantics is the link between the standard semantics of the language and the abstract one. Our concrete semantics is denotational and is based on the notion of substitution sequence. Correctness of the concrete semantics with respect to Prolog standard semantics, i.e., OLD-resolution, is discussed. Most proofs are omitted here; all details can be found in [57].

### *2.1. Syntax*

The abstract interpretation framework presented in this paper assumes that programs are normalized according to the abstract syntax given in Figure 2.1. The variables occurring in a literal are distinct; distinct procedures have distinct names; all clauses of a procedure have exactly the same head; if a clause uses $m$ different program variables, these variables are $x_1, \ldots, x_m$.

| | | | | | |
|---|---|---|---|---|---|
| $P$ | $\in$ | *Programs* | $P$ | $::=$ | $pr \mid pr\ P$ |
| $pr$ | $\in$ | *Procedures* | $pr$ | $::=$ | $c \mid c\ pr$ |
| $c$ | $\in$ | *Clauses* | $c$ | $::=$ | $h$ :- $g$. |
| $h$ | $\in$ | *ClauseHeads* | $h$ | $::=$ | $p(x_1, \ldots, x_n)$ |
| $g$ | $\in$ | *ClauseBodyPrefixes* | $g$ | $::=$ | $<> \mid g\ ,\ l$ |
| $l$ | $\in$ | *Literals* | $l$ | $::=$ | $p(x_{i_1}, \ldots, x_{i_n}) \mid b$ |
| $b$ | $\in$ | *Built-ins* | $b$ | $::=$ | $x_i{=}x_j \mid x_{i_1}{=}f(x_{i_2}, \ldots, x_{i_n}) \mid\ !$ |
| $p$ | $\in$ | *ProcedureNames* | | | |
| $f$ | $\in$ | *Functors* | | | |
| $x_i$ | $\in$ | *ProgramVariables (PV)* | | | |

**FIGURE 2.1.** Abstract Syntax of Normalized Programs

## *2.2. Basic Semantic Domains*

This section presents the basic semantic domains of substitutions. Note that we assume a preliminary knowledge of logic programming (see, for instance [1, 64]).

**Variables and Terms.** We assume the existence of two disjoint and infinite sets of variables, denoted by $PV$ and $SV$. Elements of $PV$ are called *program variables* and are denoted by $x_1, x_2, \ldots, x_i, \ldots$. The set $PV$ is totally ordered; $x_i$ is the $i$-th element of $PV$. Elements of $SV$ are called *standard variables* and are denoted by letters $y$ and $z$ (possibly subscripted). Terms are built using standard variables only.

**Standard Substitutions.** Standard substitutions are substitutions in the usual sense [1, 64] which use standard variables only. The set of standard substitutions is denoted by $SS$. Renamings are standard substitutions that define a permutation of standard variables. The domain and the codomain of a standard substitution $\sigma$ are denoted by $dom(\sigma)$ and $codom(\sigma)$, respectively. We denote by $mgu(t_1, t_2)$ the set of standard substitutions that are a most general unifier of terms $t_1$ and $t_2$.

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

all $x_i \in D$. We denote by $PS_D$ the set of program substitutions whose domain is $D$.

**Canonical Program Substitutions.** We say that two program substitutions $\theta$ and $\theta'$ are *equivalent* if and only if there exists a renaming $\rho$ such that $\theta\rho = \theta'$. We assume that, for each program substitution $\theta$, we are given a canonical representative, denoted by $[\![\theta]\!]$, of the set of all program substitutions that are equivalent to $\theta$. We denote by $CPS$ the set of all *canonical program substitutions* $[\![\theta]\!]$. For any finite set of program variables $D$, we denote by $CPS_D$ the set $PS_D \cap CPS$.

## *2.3. Program Substitution Sequences*

Program substitution sequences are intended to model the sequence of computed answer substitutions returned by a goal, a clause, or a procedure.

**Program Substitution Sequences.** Let us denote by $\mathbf{N}^\star$ the set of positive natural numbers. A program substitution sequence is either a *finite* sequence of the form $< \theta_1, \ldots, \theta_n >$ $(n \geq 0)$ or an *incomplete* sequence of the form $< \theta_1, \ldots, \theta_n, \bot >$ $(n \geq 0)$ or an *infinite* sequence of the form $< \theta_1, \ldots, \theta_i, \ldots >$ $(i \in \mathbf{N}^\star)$, where the $\theta_i$ are program substitutions with the same domain. We use the notation $< \theta_1, \ldots, \theta_i, \_ >$ to represent a program substitution sequence when it is not known whether it is finite, incomplete or infinite. Let $S$ be a program substitution sequence. We denote by $Subst(S)$ the set of program substitutions that are elements of $S$. The domain of $S$ is defined when $S \neq <>$ and $S \neq < \bot >$. In this case, $dom(S)$ is the domain of the program substitutions belonging to $Subst(S)$. The set of all program substitution sequences is denoted by $PSS$. Let $D$ be a finite set of program variables. We denote by $PSS_D$ the set of all program substitution sequences with domain $D$ augmented with $<>$ and $< \bot >$. Let $S \in PSS_D$ be a sequence $< \theta_1, \ldots, \theta_i, \_ >$ and $D' \subseteq D$. The *restriction* of $S$ to $D'$, denoted by $S_{/D'}$, is the program substitution sequence $< \theta_{1/D'}, \ldots, \theta_{i/D'}, \_ >$. The number of elements of $S$, including the special element $\bot$, is denoted by $Ne(S)$. The number of elements of $S$ that are substitutions is denoted by $Ns(S)$. Sequence concatenation is denoted by :: and it is used only when its first argument is a finite sequence.

**Canonical Substitution Sequences.** The canonical mapping $[\![\cdot]\!]$ is lifted to sequences as follows. Let $S$ be a program substitution sequence $< \theta_1, \ldots, \theta_i, \_ >$. We define $[\![S]\!] =< [\![\theta_1]\!], \ldots, [\![\theta_i]\!], \_ >$. We denote by $CPSS$ the set of all *canonical substitution sequences* $[\![S]\!]$ and by $CPSS_D$ the set $PSS_D \cap CPSS$, for any finite subset $D$ of $PV$.

**CPO's of Program Substitution Sequences.** The sets $PSS$, $PSS_D$, $CPSS$ and $CPSS_D$ can be endowed with a structure of *pointed cpo* as described below.

*Definition 2.1.* [Relation $\sqsubseteq$ on Program Substitution Sequences]
Let $S_1, S_2 \in PSS$. We define

$$S_1 \sqsubseteq S_2 \quad \text{iff} \quad \begin{array}{ll} \text{either} & S_1 = S_2 \\ \text{or} & \text{there exists } S, S' \in PSS \text{ such that } S \text{ is finite,} \\ & S_1 = S ::< \bot > \text{ and } S_2 = S :: S'. \end{array}$$

The relation $\sqsubseteq$ on program substitution sequences is an ordering and the pairs $\langle PSS, \sqsubseteq\rangle$, $\langle CPSS, \sqsubseteq\rangle$, $\langle PSS_D, \sqsubseteq\rangle$, and $\langle CPSS_D, \sqsubseteq\rangle$ are all pointed cpo's (see [57]).

We denote by $(S_i)_{i\in\mathbf{N}}$ an increasing chain, $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_i \sqsubseteq \ldots$ in $PSS$; whereas we denote by $\{S_i\}_{i\in\mathbf{N}}$ a, non necessarily increasing, sequence of elements of $PSS$.

**Lazy Concatenation.** Program substitution sequences are combined through the operation $\Box$ and its extensions $\Box_{k=1}^{n}$ and $\Box_{k=1}^{\infty}$ defined below.

*Definition 2.2.* [Operation $\Box$]
Let $S_1,\ S_2 \in PSS$.

$$\begin{array}{rcccl} S_1 \Box S_2 & = & S_1 :: S_2 & \text{if} & S_1 \text{ is finite} \\ & = & S_1 & \text{if} & S_1 \text{ is incomplete or infinite.} \end{array}$$

*Definition 2.3.* [Operation $\Box_{k=1}^{n}$]
Let $\{S_k\}_{k\in\mathbf{N}^*}$ be an infinite sequence of program substitution sequences (not necessarily a chain). For any $n \geq 1$, we define:

$$\begin{array}{rcl} \Box_{k=1}^{0} S_k & = & < > \\ \Box_{k=1}^{n} S_k & = & (\Box_{k=1}^{n-1} S_k) \Box S_n. \end{array}$$

*Definition 2.4.* [Operation $\Box_{k=1}^{\infty}$]
Let $\{S_k\}_{k\in\mathbf{N}^*}$ be an infinite sequence of program substitution sequences. The infinite sequence $\{S'_i\}_{i\in\mathbf{N}}$ where $S'_i = (\Box_{k=1}^{i} S_k)\Box < \bot >$ $(i \in \mathbf{N})$ is a chain. So we are allowed to define:

$$\Box_{k=1}^{\infty} S_k \quad = \quad \sqcup_{i=0}^{\infty} S'_i \quad = \quad \sqcup_{i=0}^{\infty}((\Box_{k=1}^{i} S_k)\Box < \bot >).$$

The operation $\Box$ is associative; hence, it is meaningful to write $S_1 \Box \ldots \Box S_n$ instead of $\Box_{k=1}^{n} S_k$. Operations $\Box$, $\Box_{k=1}^{n}$, and $\Box_{k=1}^{\infty}$ are continuous with respect to the ordering $\sqsubseteq$ on program substitution sequences.

**Program Substitution Sequences with Cut Information.** Program substitution sequences with cut information are used to model the result of a clause together with information on cut executions.
Let $CF$ be the set of cut flags $\{cut, nocut\}$. A program substitution sequence with cut information is a pair $\langle S, cf\rangle$ where $S \in PSS$ and $cf \in CF$.

*Definition 2.5.* [Relation $\sqsubseteq$ on Substitution Sequences with Cut Information]
Let $\langle S_1, cf_1\rangle, \langle S_2, cf_2\rangle \in PSS \times CF$. We define

$$\begin{array}{lcll} \langle S_1, cf_1\rangle \sqsubseteq \langle S_2, cf_2\rangle & \text{iff} & \text{either} & S_1 \sqsubseteq S_2 \text{ and } cf_1 = cf_2 \\ & & \text{or} & S_1 =< \bot > \text{ and } cf_1 = nocut. \end{array}$$

The relation $\sqsubseteq$ on program substitution sequences with cut information is an ordering. Moreover, the pairs $\langle PSS \times CF, \sqsubseteq\rangle$, $\langle PSS_D \times CF, \sqsubseteq\rangle$, $\langle CPSS \times CF, \sqsubseteq\rangle$ and $\langle CPSS_D \times CF, \sqsubseteq\rangle$ are all pointed cpo's.
We extend the definition of the operation $\Box$ to program substitution sequences with cut information. The extension is continuous in both the arguments.

*Definition 2.6.* [Operation $\Box$ with Cut Information]
Let $\langle S_1, cf\rangle \in PSS \times CF$ and $S_2 \in PSS$. We define

$$\langle S_1, cf\rangle \Box S_2 \quad = \quad \begin{array}{ll} S_1 \Box S_2 & \text{if} \quad cf = nocut \\ S_1 & \text{if} \quad cf = cut. \end{array}$$

## *2.4. Concrete Behaviors*

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

*Definition 2.7.* [Concrete Underlying Domain]
The *concrete underlying domain*, denoted by $CUD$, is the set of all pairs $\langle \theta, p\rangle$ such that $p$ is the name of a procedure $pr$ of $P$ and $\theta \in CPS_{\{x_1,\ldots,x_n\}}$, where $x_1, \ldots, x_n$ are the variables occurring in the head of every clause of $pr$.

Concrete behaviors are functions but we denote them by the relation symbol $\longmapsto$ in order to stress the similarities between the concrete semantics and a structural operational semantics for logic programs defined in [62].

*Definition 2.8.* [Concrete Behaviors]
A *concrete behavior* is a total function $\longmapsto$: $CUD \longrightarrow CPSS$ mapping every pair $\langle \theta, p\rangle \in CUD$ to a canonical program substitution sequence $S$ such that, for every $\theta' \in Subst(S)$, there exists a standard substitution $\sigma$ such that $\theta' = \theta\sigma$. We denote by $\langle \theta, p\rangle \longmapsto S$ the fact that $\longmapsto$ maps the pair $\langle \theta, p\rangle$ to $S$. The set of all concrete behaviors is denoted by $CB$.

The ordering $\sqsubseteq$ on program substitution sequences is lifted to concrete behaviors in a standard way [82].

*Definition 2.9.* [Relation $\sqsubseteq$ on Concrete Behaviors]
Let $\longmapsto_1, \longmapsto_2 \in CB$. We define

$$\longmapsto_1 \sqsubseteq \longmapsto_2 \quad \text{iff} \quad (\langle \theta, p\rangle \longmapsto_1 S_1 \text{ and } \langle \theta, p\rangle \longmapsto_2 S_2) \text{ imply } S_1 \sqsubseteq S_2, \text{ for all } \langle \theta, p\rangle \in CUD.$$

The following result is straightforward.

*Proposition 2.1.* *$\langle CB, \sqsubseteq\rangle$ is a pointed cpo, i.e.,*

1. *the relation $\sqsubseteq$ on $CB$ is a partial order;*
2. *$CB$ has a minimum element, which is the concrete behavior $\longmapsto_\perp$ such that for all $\langle \theta, p\rangle \in CUD$, $\langle \theta, p\rangle \longmapsto_\perp < \perp >$;*
3. *every chain $(\longmapsto_i)_{i\in\mathbf{N}}$ in $CB$ has a least upper bound, denoted by $\sqcup_{i=0}^{\infty} \longmapsto_i$; $\sqcup_{i=0}^{\infty} \longmapsto_i$ is the concrete behavior $\longmapsto$ such that, for all $\langle \theta, p\rangle \in CUD$, $\langle \theta, p\rangle \longmapsto \sqcup_{i=0}^{\infty} S_i$, where $\langle \theta, p\rangle \longmapsto_i S_i$ $(\forall i \in \mathbf{N})$.*

### *2.5. Concrete Operations*

We specify here the concrete operations which are used in the definition of the concrete semantics. The choice of these particular operations is motivated by the fact that they have useful (i.e., practical) abstract counterparts (see Sections 3, 4 and 5). The concrete operations are polymorphic since their exact signature depends on a clause $c$ or a literal $l$ or both.
Let $c$ be a clause, $D = \{x_1, \ldots, x_n\}$ be the set of all variables occurring in the head of $c$, and $D' = \{x_1, \ldots, x_m\}$ $(n \leq m)$ be the set of all variables occurring in $c$.

**Extension at Clause Entry** : $\texttt{EXTC}(c, \cdot) : CPS_D \rightarrow (CPSS_{D'} \times CF)$
This operation extends a substitution $\theta$ on the set of variables in $D$ to the set of variables in $D'$. Let $\theta \in CPS_D$.

$$\texttt{EXTC}(c, \theta) \quad = \quad \langle < [\![\theta']\!] >, nocut \rangle$$

where $x_i\theta' = x_i\theta$ $(\forall i : 1 \leq i \leq n)$ and $x_{n+1}\theta'$, ..., $x_m\theta'$ are distinct standard variables not belonging to $codom(\theta)$.

**Restriction at Clause Exit** : $\texttt{RESTRC}(c, \cdot) : (CPSS_{D'} \times CF) \rightarrow (CPSS_D \times CF)$
This operation restricts a pair $\langle S, cf \rangle$, representing the result of the execution of $c$ on the set of variables in $D'$, to the set of variables in $D$. Let $\langle S, cf \rangle \in (CPSS'_D \times CF)$.

$$\texttt{RESTRC}(c, \langle S, cf \rangle) \quad = \quad \langle [\![S']\!], cf \rangle \quad \text{where } S' = S_{/D}.$$

Let $l$ be a literal occurring in the body of $c$, $D'' = \{x_{i_1}, \ldots, x_{i_r}\}$ be the set of variables occurring in $l$, and $D'''$ be equal to $\{x_1, \ldots, x_r\}$.

**Restriction before a Call** : $\texttt{RESTRG}(l, \cdot) : CPS_{D''} \rightarrow CPS_{D'''}$
This operation expresses a substitution $\theta$ on the parameters $x_{i_1}, \ldots, x_{i_r}$ of a call $l$ in terms of the formal parameters $x_1, \ldots, x_r$ of $l$. Let $\theta \in CPS_{D''}$.

$$\texttt{RESTRG}(l, \theta) \quad = \quad [\![\{x_1/x_{i_1}\theta, \ldots, x_r/x_{i_r}\theta\}]\!].$$

**Extension of the Result of a Call** : $\texttt{EXTG}(l, \cdot, \cdot) : CPS_{D'} \times CPSS_{D'''} \not\rightarrow CPSS_{D'}$
This operation extends a substitution $\theta$ with a substitution sequence $S$ representing the result of executing a call $l$ on $\theta$. Hence, it is only used in contexts where the substitutions that are elements of $S$ are (roughly speaking) instances of $\theta$. Let $\theta \in CPS_{D'}$. Let $S \in CPSS_{D'''}$ be of the form $< \theta'\sigma_1, \ldots, \theta'\sigma_i, \_ >$ where $x_j\theta' = x_{i_j}\theta$ $(1 \leq j \leq r)$ and the $\sigma_i$ are standard substitutions such that $dom(\sigma_i) \subseteq codom(\theta')$. Let $\{z_1, \ldots, z_s\} = codom(\theta) \setminus codom(\theta')$. Let $y_{i,1}, \ldots, y_{i,s}$ be distinct standard variables not belonging to $codom(\theta) \cup codom(\sigma_i)$ $(1 \leq i \leq Ns(S))$. Let $\rho_i$ be a renaming of the form $\{z_1/y_{i,1}, \ldots, z_s/y_{i,s}, y_{i,1}/z_1, \ldots, y_{i,s}/z_s\}$.

$$\texttt{EXTG}(l, \theta, S) \quad = \quad [\![< \theta\rho_1\sigma_1, \ldots, \theta\rho_i\sigma_i, \_ >]\!].$$

It is easy to see that the value of $\texttt{EXTG}(l, \theta, S)$ does not depend on the choice of the $y_{i,j}$. Moreover, it is not defined when $S$ is not of the above mentioned form.

**Unification of Two Variables** : UNIF-VAR: $CPS_{\{x_1, x_2\}} \rightarrow CPSS_{\{x_1, x_2\}}$
Let $\theta \in CPS_{\{x_1, x_2\}}$. This operation unifies $x_1\theta$ with $x_2\theta$.

$$\begin{array}{lcll} \texttt{UNIF-VAR}(\theta) & = & <> & \text{if } x_1\theta \text{ and } x_2\theta \text{ are not unifiable,} \\ & = & [\![< \theta\sigma >]\!] & \text{where } \sigma \in mgu(x_1\theta, x_2\theta), \text{ otherwise.} \end{array}$$

**Unification of a Variable and a Functor** : $\texttt{UNIF-FUNC}(f, \cdot) : CPS_D \rightarrow CPSS_D$
Given a functor $f$ of arity $n-1$ and a substitution $\theta \in CPS_D$ where $D = \{x_1, \ldots, x_n\}$, the UNIF-FUNC operation unifies $x_1\theta$ with $f(x_2, \ldots, x_n)\theta$.

$$\begin{array}{lcll} \texttt{UNIF-FUNC}(f, \theta) & = & <> & \text{if } x_1\theta \text{ and } f(x_2, \ldots, x_n\theta) \text{ are not unifiable,} \\ & = & [\![< \theta\sigma >]\!] & \text{where } \sigma \in mgu(x_1\theta, f(x_2, \ldots, x_n)\theta), \text{ otherwise.} \end{array}$$

All operations defined in this section are monotonic and continuous with respect to the orderings defined in the previous sections. Sets of program substitutions are endowed with the trivial ordering $\sqsubseteq$ such that $\theta \sqsubseteq \theta'$ iff $\theta = \theta'$.

## *2.6. Concrete Semantic Rules*

The concrete semantics of the underlying program $P$ is the least fixpoint of a continuous transformation on $CB$ (the set of concrete behaviors). This transformation is defined in terms of a set of semantic rules that naturally extend a concrete behavior to a continuous function defining the input/output behavior of every prefix of the body of a clause, every clause, every suffix of a procedure and every procedure of $P$. This function is called *extended concrete behavior* and maps each element of the extended concrete underlying domain to a substitution sequence, possibly with cut information, as defined below.

*Definition 2.10.* [Extended Concrete Underlying Domain]
The *extended concrete underlying domain*, denoted by $ECUD$, consists of

1. all triples $\langle \theta, g, c \rangle$, where $c$ is a clause of $P$, $g$ is a prefix of the body of $c$, and $\theta$ is a canonical program substitution over the variables in the head of $c$;
2. all pairs $\langle \theta, c \rangle$, where $c$ is a clause of $P$ and $\theta$ is a canonical program substitution over the variables in the head of $c$;
3. all pairs $\langle \theta, pr \rangle$, where $pr$ is a procedure of $P$ or a suffix of a procedure of $P$ and $\theta$ is a canonical program substitution over the variables in the head of the clauses of $pr$.

*Definition 2.11.* [Extended Concrete Behaviors]
An *extended concrete behavior* is a total function from $ECUD$ to the set $CPSS \cup (CPSS \times CF)$ such that

1. every triple $\langle \theta, g, c \rangle$ from $ECUD$ is mapped to a program substitution sequence with cut information $\langle S, cf \rangle$ such that $dom(S)$ is the set of all variables in the clause $c$;
2. every pair $\langle \theta, c \rangle$ from $ECUD$ is mapped to a program substitution sequence with cut information $\langle S, cf \rangle$ such that $dom(S)$ is the set of variables in the head of the clause $c$;
3. every pair $\langle \theta, pr \rangle$ from $ECUD$ is mapped to a program substitution sequence $S$ such that $dom(S)$ is the set of variables in the head of the clauses of the procedure $pr$.

The set of extended concrete behaviors is endowed with a structure of pointed cpo in the obvious way. It is denoted by *ECB*; its elements are denoted by $\longmapsto$.

Let $\longmapsto$ be a concrete behavior. The concrete semantic rules depicted in Figure 2.2 define an extended concrete behavior derived from $\longmapsto$. This extended concrete behavior is denoted by the same symbol $\longmapsto$. This does not lead to confusion since the inputs of the two functions belong to different sets. The definition proceeds by induction on the syntactic structure of $P$.
The concrete semantic rules model Prolog operational semantics through the notion of program substitution sequence. Rule **R1** defines the program substitution sequence with cut information at the entry point of a clause. Rules **R2** and **R3** define the effect of the execution of a cut at the clause level. Rules **R4**, **R5** and **R6** deal with execution of literals; procedure calls are solved by using the concrete behavior $\longmapsto$ as an oracle. Rule **R7** defines the result of a clause. Rules **R8** and **R9** define the result of a procedure by structural induction on its suffixes. Rule **R8** deals with the suffix consisting of the last clause only: it simply forgets the cut information, which is not meaningful at the procedure level. Rule **R9** combines the result of a clause with the (combined) result of the next clauses in the same procedure: it deals with the execution of a cut at the procedure level. The expression $\Box_{k=1}^{Ne(S)} S_k$ used in Rules **R4**, **R5** and **R6** deserves an explanation: when the sequence $S$ is incomplete, it is assumed that $S_{Ne(S)} =< \bot >$. This convention is necessary to propagate the non-termination of $g'$ to $g$.
The following results are instrumental for proving the well-definedness of the concrete semantics.

*Proposition 2.2. [Properties of the Concrete Semantic Rules]*

1. *Given a concrete behavior, the concrete semantic rules define a unique extended concrete behavior, i.e., a unique mapping from CB to ECB. This mapping is continuous.*

2. *Rules* **R1** *to* **R6** *have a conclusion of the form* $\langle \theta, g, c\rangle \longmapsto \langle S, cf\rangle$. *In all cases,* $S$ *is of the form* $< \theta'\sigma_1, \ldots, \theta'\sigma_i, \_ >$, *where the* $\sigma_i$ *are standard substitutions and* $\langle \theta', nocut\rangle = \mathtt{EXTC}(c, \theta)$.
*Rules* **R7** *to* **R9** *have a conclusion of the form* $\langle \theta, \cdot\rangle \longmapsto S$. *In all cases,* $S$ *is of the form* $< \theta\sigma_1, \ldots, \theta\sigma_i, \_ >$, *where the* $\sigma_i$ *are standard substitutions.*

## *2.7. Concrete Semantics*

The concrete semantics of the underlying program $P$ is defined as the least fixpoint of the following concrete transformation.

*Definition 2.12.* [Concrete Transformation]
The transformation $TCB : CB \to CB$ is defined as follows: for all $\longmapsto \in CB$,

$$\textbf{T1}\quad \frac{\begin{array}{c} pr \text{ is a procedure of } P \\ p \text{ is the name of } pr \\ \langle \theta, pr\rangle \longmapsto S \end{array}}{\langle \theta, p\rangle \overset{TCB}{\longmapsto} S}$$

$$\textbf{R1}\quad \frac{g ::= <>}{\langle \theta, g, c\rangle \longmapsto \texttt{EXTC}(c, \theta)}$$

$$\textbf{R2}\quad \frac{\begin{array}{c} g ::= g' \text{ , } ! \\ \langle \theta, g', c\rangle \longmapsto \langle S, cf\rangle \\ S \in \{< \bot >, <>\} \end{array}}{\langle \theta, g, c\rangle \longmapsto \langle S, cf\rangle}$$

$$\textbf{R3}\quad \frac{\begin{array}{c} g ::= g' \text{ , } ! \\ \langle \theta, g', c\rangle \longmapsto \langle S, cf\rangle \\ S =< \theta' >:: S' \end{array}}{\langle \theta, g, c\rangle \longmapsto \langle < \theta' >, cut\rangle}$$

$$\textbf{R4}\quad \frac{\begin{array}{c} g ::= g' \text{ , } l \\ l ::= x_i = x_j \\ \langle \theta, g', c\rangle \longmapsto \langle S, cf\rangle \\ S =< \theta_1, \ldots, \theta_i, \_ > \\ \left\{ \begin{array}{c} \theta'_k = \texttt{RESTRG}(l, \theta_k) \\ S'_k = \texttt{UNIF-VAR}(\theta'_k) \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) \\ (1 \le k \le Ns(S)) \end{array} \right\} \end{array}}{\langle \theta, g, c\rangle \longmapsto \langle \Box_{k=1}^{Ne(S)} S_k, cf\rangle}$$

$$\textbf{R5}\quad \frac{\begin{array}{c} g ::= g' \text{ , } l \\ l ::= x_{i_1} = f(x_{i_2}, \ldots, x_{i_n}) \\ \langle \theta, g', c\rangle \longmapsto \langle S, cf\rangle \\ S =< \theta_1, \ldots, \theta_i, \_ > \\ \left\{ \begin{array}{c} \theta'_k = \texttt{RESTRG}(l, \theta_k) \\ S'_k = \texttt{UNIF-FUNC}(f, \theta'_k) \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) \\ (1 \le k \le Ns(S)) \end{array} \right\} \end{array}}{\langle \theta, g, c\rangle \longmapsto \langle \Box_{k=1}^{Ne(S)} S_k, cf\rangle}$$

$$\textbf{R6}\quad \frac{\begin{array}{c} g ::= g' \text{ , } l \\ l ::= p(x_{i_1}, \ldots, x_{i_n}) \\ \langle \theta, g', c\rangle \longmapsto \langle S, cf\rangle \\ S =< \theta_1, \ldots, \theta_i, \_ > \\ \left\{ \begin{array}{c} \theta'_k = \texttt{RESTRG}(l, \theta_k) \\ \langle \theta'_k, p\rangle \longmapsto S'_k \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) \\ (1 \le k \le Ns(S)) \end{array} \right\} \end{array}}{\langle \theta, g, c\rangle \longmapsto \langle \Box_{k=1}^{Ne(S)} S_k, cf\rangle}$$

$$\textbf{R7}\quad \frac{\begin{array}{c} c \ ::= h \text{ :- } g. \\ \langle \theta, g, c\rangle \longmapsto \langle S, cf\rangle \end{array}}{\langle \theta, c\rangle \longmapsto \texttt{RESTRC}(c, \langle S, cf\rangle)}$$

$$\textbf{R8}\quad \frac{\begin{array}{c} pr \ ::= c \\ \langle \theta, c\rangle \longmapsto \langle S, cf\rangle \end{array}}{\langle \theta, pr\rangle \longmapsto S}$$

$$\textbf{R9}\quad \frac{\begin{array}{c} pr \ ::= c \ pr' \\ \langle \theta, c\rangle \longmapsto \langle S, cf\rangle \\ \langle \theta, pr'\rangle \longmapsto S' \end{array}}{\langle \theta, pr\rangle \longmapsto \langle S, cf\rangle \Box S'}$$

**FIGURE 2.2.** Concrete Semantic Rules

where $\overset{TCB}{\longmapsto}$ stands for $TCB(\longmapsto)$. Remember that $\langle \theta, pr \rangle \longmapsto S$ is defined by means of the previous rules which use the concrete behavior $\longmapsto$ as an oracle to solve the procedure calls.

The transformation $TCB$ is well-defined and continuous.

*Definition 2.13.* [Concrete Semantics]
The concrete semantics of the underlying program $P$ is the least concrete behavior $\longmapsto$ such that

$$\longmapsto = \overset{TCB}{\longmapsto} .$$

## *2.8. Correctness of the Concrete Semantics*

Since OLD-resolution [64, 87] is the standard semantics of pure Prolog augmented with cut, our concrete semantics and OLD-resolution have to be proven equivalent. The proof is fairly complex because OLD-resolution is not compositional. Consequently, the two semantics do not naturally match. The equivalence proof is given in [57]. In this section, we only give the principle of the proof.

1. We assume that OLD-resolution uses standard variables to rename clauses apart. The initial queries are also assumed to contain standard variables only.

2. The notion of *incomplete OLD-tree limited to depth k* is defined ($IOLD_k$-tree, for short). Intuitively, an $IOLD_k$-tree is an OLD-tree modified according to the following rules:

   (a) procedure calls may be unfolded only down to depth $k$;
   (b) branches that end at a node whose leftmost literal may not be unfolded are called *incomplete*;
   (c) a depth-first left-to-right traversal of the tree is performed in order to determine the cuts that are reached by the standard execution and to prune the tree accordingly (see [64]);
   (d) the traversal ends when the whole tree has been visited or when a node that may not be unfolded is reached;
   (e) the branches on the right of the left-most incomplete branch are pruned (if such a branch exists).

3. Assuming a query of the form $p(t_1, \ldots, t_n)$ and denoting the concrete behavior $TCB^k(\longmapsto_\perp)$ by $\longmapsto_k$, it can be shown that the sequence of computed answer substitutions $< \sigma_1, \ldots, \sigma_i, \_ >$ for the $IOLD_k$-tree of $p(t_1, \ldots, t_n)$ is such that $\langle \theta, p \rangle \longmapsto_k [\![< \theta\sigma_1, \ldots, \theta\sigma_i, \_ >]\!]$ where $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$.

4. The equivalence of our concrete semantics and OLD-resolution is a simple consequence of the previous result.
   For every query $p(t_1, \ldots, t_n)$, $< \sigma_1, \ldots, \sigma_i, \_ >$ is the sequence of computed answer substitutions of $p(t_1, \ldots, t_n)$ according to OLD-resolution if and only if $\langle \theta, p \rangle \longmapsto [\![< \theta\sigma_1, \ldots, \theta\sigma_i, \_ >]\!]$ where $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ and $\longmapsto$ is the concrete behavior of the program according to our concrete semantics.

In fact, the correctness of our concrete semantics should be close to obvious to anyone who knows about both Prolog and denotational semantics. So, the equivalence proof is a formal technical exercise, which adds little to our basic understanding of the concrete semantics.

### *2.9. Related Works*

Denotational semantics for Prolog have been proposed before (e.g., [26, 28, 46]). Our concrete semantics is not intended to improve on these works from the language understanding standpoint. Instead, it is merely designed as a basis for an abstract interpretation framework; in particular, it uses concrete operations that are as close as possible to the operations used by the structural operational semantics presented in [62] upon which our previous frameworks are based. This allows us to reuse much of the material from our existing abstract domains and generic algorithms (e.g., [32, 53, 61, 62]). The idea of distinguishing between finite, incomplete, and infinite sequences is originally due to M. Baudinet [5].

## 3. ABSTRACT SEMANTICS

As we have just explained in the introduction, our abstract semantics is not defined as a least fixpoint of an abstract transformation but instead as a set of post-fixpoints that fulfill a safety requirement, namely pre-consistency. Moreover, the abstract domains are assumed to represent so-called chain-closed sets of concrete elements as specified below.

### *3.1. Abstract Domains*

We state here the mathematical assumptions that are required to be satisfied by the abstract domains. Specific abstract domains will be described in Section 5.

**Abstract Substitutions.** For every finite set $D$ of program variables, we denote by $CS_D$ the set $\wp(PS_D)$. A domain of abstract substitutions is a family of sets $AS_D$ indexed by the finite sets $D$ of program variables. Elements of $AS_D$ are called *abstract substitutions*; they are denoted by $\beta$. Each set $AS_D$ is endowed with a partial order $\leq$ and a monotonic *concretization* function $Cc : AS_D \to CS_D$ associating to each abstract substitution $\beta$ the set $Cc(\beta)$ of program substitutions it denotes.

**Abstract Sequences.** For every finite set $D$ of program variables, we denote by $CSS_D$ the set $\wp(PSS_D)$. Abstract sequences denote chain-closed subsets of $CS_D$. A domain of abstract sequences is a family of sets $ASS_D$ indexed by the finite sets $D$ of program variables. Elements of $ASS_D$ are called *abstract sequences*; they are denoted by $B$. Each set $ASS_D$ is endowed with a partial order $\leq$ and a monotonic *concretization* function $Cc : ASS_D \to CSS_D$. Moreover, the following properties are required to be satisfied: (1) every $ASS_D$ contains an abstract sequence $B_\perp$ such that $< \perp > \in Cc(B_\perp)$; (2) for every $B \in ASS_D$, $Cc(B)$ is *chain-closed*, i.e., for every chain $(S_i)_{i \in \mathbf{N}}$ of elements of $Cc(B)$, the limit $\sqcup_{i=0}^{\infty} S_i$ also belongs to $Cc(B)$. The disjoint union of all the $ASS_D$ is denoted by $ASS$.

**Abstract Sequences with Cut Information.** Let $CSSC_D$ denote $\wp(PSS_D \times CF)$. A domain of abstract sequences with cut information is a family of sets $ASSC_D$ indexed by the finite sets $D$ of program variables. Elements of $ASSC_D$ are called *abstract sequences with cut information*; they are denoted by $C$. Every set $ASSC_D$ is endowed with a partial order $\leq$ and a monotonic *concretization* function $Cc : ASSC_D \to CSSC_D$. The disjoint union of all the $ASSC_D$ is denoted by $ASSC$.

**Abstract Behaviors.** Abstract behaviors are the abstract counterpart of the concrete behaviors introduced in Section 2.4. They are endowed with a weaker mathematical structure as described below. As in the case of concrete behaviors, a fixed underlying program $P$ is assumed.

*Definition 3.1.* [Abstract Underlying Domain]
The *abstract underlying domain*, denoted by $AUD$, is the set of all pairs $\langle \beta, p \rangle$ such that $p$ is a procedure name in $P$ of arity $n$ and $\beta \in AS_{\{x_1,\ldots,x_n\}}$.

*Definition 3.2.* [Abstract Behaviors]
An *abstract behavior* is a total function $sat : AUD \longrightarrow ASS$ mapping each pair $\langle \beta, p \rangle \in AUD$ to an abstract sequence $B$ with $B \in ASS_{\{x_1,\ldots,x_n\}}$, where $n$ is the arity of $p$. The set of all abstract behaviors is denoted by $AB$. The set $AB$ is endowed with the partial ordering $\leq$ such that, for all $sat_1, sat_2 \in AB$:

$$sat_1 \leq sat_2 \text{ iff } sat_1\langle \beta, p \rangle \leq sat_2\langle \beta, p \rangle, \ \forall \langle \beta, p \rangle \in AUD.$$

It would be reasonable to assume that abstract behaviors are monotonic functions but this is not necessary for the safety results. The notation *sat* stands for "set of abstract tuples". It is used because the abstract interpretation algorithm, derived from the abstract semantics, actually computes a set of tuples of the form $\langle \beta, p, B \rangle$, i.e., a part of the table of an abstract behavior.

## *3.2. Abstract Operations*

In this section, we give the specification of the primitive abstract operations used by the abstract semantics. The specifications are safety assumptions which, roughly speaking, state that the abstract operations safely simulate the corresponding concrete ones. In particular, operations `EXTC`, `RESTRG`, `RESTRC`, `UNIF-VAR`, `UNIF-FUNC` are faithful abstract counterparts of the corresponding concrete operations. Hence, their specification simply states that, if some concrete input belongs to the concretization of their (abstract) input, then the corresponding concrete output belongs to the concretization of their (abstract) output. Moreover, overloading the operation names is natural in these cases. Operation `AI-CUT` deals with the cut; its specification is also straightforward. Operations `EXTGS` and `CONC` are related to the concrete operations `EXTG` and $\square$ in a more involved way. We will discuss them in more detail. Finally, operations `SUBST` and `SEQ` are simple conversion operations to convert an abstract domain into another.
Let us specify the operations, using the notations of Section 2.5.

**Extension at Clause Entry** : $\texttt{EXTC}(c, \cdot) : AS_D \to ASSC_{D'}$
Let $\beta \in AS_D$ and $\theta \in CPS_D$. The following property is required to hold.

$$\theta \in Cc(\beta) \Rightarrow \texttt{EXTC}(c,\theta) \in Cc(\texttt{EXTC}(c,\beta)).$$

**Restriction at Clause Exit** : $\texttt{RESTRC}(c,\cdot) : ASSC_{D'} \rightarrow ASSC_D$
Let $C \in ASSC_{D'}$ and $\langle S, cf\rangle \in (CPSS'_D \times CF)$.

$$\langle S, cf\rangle \in Cc(C) \Rightarrow \texttt{RESTRC}(c,\langle S, cf\rangle) \in Cc(\texttt{RESTRC}(c,C)).$$

**Restriction before a Call** : $\texttt{RESTRG}(l,\cdot) : AS_{D''} \rightarrow AS_{D'''}$
Let $\beta \in AS_{D''}$ and $\theta \in CPS_{D''}$.

$$\theta \in Cc(\beta) \Rightarrow \texttt{RESTRG}(l,\theta) \in Cc(\texttt{RESTRG}(l,\beta)).$$

**Unification of Two Variables** : $\texttt{UNIF-VAR}: AS_{\{x_1,x_2\}} \rightarrow ASS_{\{x_1,x_2\}}$
Let $\beta \in AS_{\{x_1,x_2\}}$ and $\theta \in CPS_{\{x_1,x_2\}}$.

$$\theta \in Cc(\beta) \Rightarrow \texttt{UNIF-VAR}(\theta) \in Cc(\texttt{UNIF-VAR}(\beta)).$$

**Unification of a Variable and a Functor** : $\texttt{UNIF-FUNC}(f,\cdot) : AS_D \rightarrow ASS_D$
Let $\beta \in AS_D$ and $\theta \in CPS_D$. Let also $f$ be a functor of arity $n-1$.

$$\theta \in Cc(\beta) \Rightarrow \texttt{UNIF-FUNC}(f,\theta) \in Cc(\texttt{UNIF-FUNC}(f,\beta)).$$

**Abstract Interpretation of the Cut** : $\texttt{AI-CUT}: ASSC_{D'} \rightarrow ASSC_{D'}$
Let $C \in ASSC_{D'}$, $\theta \in CPS_{D'}$, $S \in CPSS_{D'}$, $cf \in CF$.

$$\begin{array}{c}
\langle <>, cf\rangle \in Cc(C) \Rightarrow \langle <>, cf\rangle \in Cc(\texttt{AI-CUT}(C)),\\
\langle < \bot >, cf\rangle \in Cc(C) \Rightarrow \langle < \bot >, cf\rangle \in Cc(\texttt{AI-CUT}(C)),\\
\langle < \theta >:: S, cf\rangle \in Cc(C) \Rightarrow \langle < \theta >, cut\rangle \in Cc(\texttt{AI-CUT}(C)).
\end{array}$$

**Extension of the Result of a Call** : $\texttt{EXTGS}(l,\cdot,\cdot) : ASSC_{D'} \times ASS_{D'''} \rightarrow ASSC_{D'}$
The specification of this operation is more complex because it abstracts in a single operation the calculation of all sequences $S_k = \texttt{EXTG}(l,\theta_k,S'_k)$ and of their concatenation $\square_{k=1}^{Ne(S)} S_k$, performed by the rules **R4**, **R5**, **R6** (see Figure 2.2). At the abstract level, it may be too expensive or even impossible to simulate the execution of $l$ for all elements of $S$, as defined in the rules. Therefore, we abstract $S$ to its substitutions, losing the ordering. The abstract execution will be the following. Assuming that $C$ abstracts the program substitution sequence with cut information $\langle S, cf\rangle$ before $l$, we compute $\beta = \texttt{SUBST}(C)$; then we compute $\beta' = \texttt{RESTRG}(l,\beta)$ and, subsequently, we get the abstract sequence $B$ resulting from the abstract execution of $l$ with input $\beta'$. The set $Cc(B)$ contains all sequences $S'_k$ of rules **R4**, **R5**, **R6**. Then, an over approximation of the set of all possible values $\square_{k=1}^{Ne(S)} S_k$ is computed from the information provided by $C$ and $B$. This is realized by the following operation EXTGS. Let $C \in ASSC_{D'}$, $B \in ASS_{D'''}$, $\langle S, cf\rangle \in (CPSS_{D'} \times CF)$ and $S'_1, \ldots, S'_{Ns(S)} \in CPSS_{D'''}$.

$$\left.\begin{array}{c}
\langle S, cf\rangle \in Cc(C),\\
S =< \theta_1, \ldots, \theta_i, _ >,\\
\left(\begin{array}{c}\forall k : 1 \le k \le Ns(S) : S'_k \in Cc(B)\\ \text{and } S_k = \texttt{EXTG}(l,\theta_k,S'_k)\end{array}\right)
\end{array}\right\} \Rightarrow \langle \square_{k=1}^{Ne(S)} S_k, cf\rangle \in Cc(\texttt{EXTGS}(l,C,B)).$$

**Abstract Lazy Concatenation** : $\texttt{CONC} : (AS_D \times ASSC_D \times ASS_D) \rightarrow ASS_D$
This operation is the abstract counterpart of the concatenation operation $\Box$. It is however extended with an additional argument to increase the accuracy. Let $B' = \texttt{CONC}(\beta, C, B)$ where $\beta$ describes a set of input substitutions for a procedure; $C$ describes the set of substitution sequences with cut information obtained by executing a clause of the procedure on $\beta$; $B$ describes the set of substitution sequences obtained by executing the subsequent clauses of the procedure on $\beta$. Then, $B'$ describes the set of substitution sequences obtained by concatenating the results according to the concrete concatenation operation $\Box$.
Let us discuss a simple example to understand the role of $\beta$. Assume that

$$Cc(C) = \{\langle <>, nocut\rangle, \langle < \{x_1/a\} >, nocut\rangle\} \text{ and } \; Cc(B) = \{<>, < \{x_1/b\} >\}.$$

If the input mode of $x_1$ is unknown, it must be assumed that all combinations of elements in $Cc(C)$ and $Cc(B)$ are possible. Thus,

$$Cc(B') = \{<>, < \{x_1/a\} >, < \{x_1/b\} >, < \{x_1/a\}, \{x_1/b\} >\}.$$

On the contrary, if the input mode of $x_1$ is known to be ground, the outputs $\langle < \{x_1/a\} >, nocut\rangle$ and $< \{x_1/b\} >$ are incompatible since $x_1$ cannot be bound to both $a$ and $b$ in the input substitution. In this case, we have

$$Cc(B') = \{<>, < \{x_1/a\} >, < \{x_1/b\} >\}.$$

The first argument $\beta$ of the operation CONC provides information on the input values: it may be useful to improve the accuracy of the result. The above discussion motivates the following specification of operation CONC. Note that the statement $(\exists \sigma \in SS : \theta' = \theta\sigma)$ is abbreviated by $\theta' \leq \theta$ in the specification. Let $\beta \in AS_D$, $C \in ASSC_D$, $B \in ASS_D$, $\theta \in CPS_D$, $\langle S_1, cf\rangle \in (CPSS_D \times CF)$ and $S_2 \in CPSS_D$.

$$\left.\begin{array}{c} \theta \in Cc(\beta), \\ \langle S_1, cf\rangle \in Cc(C), \\ S_2 \in Cc(B), \\ \forall \theta' \in Subst(S_1) \cup Subst(S_2) : \theta' \leq \theta \end{array}\right\} \Rightarrow \langle S_1, cf\rangle \Box S_2 \in Cc(\texttt{CONC}(\beta, C, B)).$$

**Operation** $\texttt{SEQ} : ASSC_D \rightarrow ASS_D$
This operation forgets the cut information contained in an abstract sequence with cut information $C$. It is applied to the result of the last clause of a procedure before combining this result with the results of the other clauses.
Let $C \in ASSC_D$ and $\langle S, cf\rangle \in (CPSS_D \times CF)$.

$$\langle S, cf\rangle \in Cc(C) \Rightarrow S \in Cc(\texttt{SEQ}(C)).$$

**Operation** $\texttt{SUBST} : ASSC_{D'} \rightarrow AS_{D'}$
This operation forgets still more information. It extracts the "abstract substitution part" of $C$. It is applied before executing a literal in a clause. See operation EXTGS.
Let $C \in ASSC_{D'}$ and $\langle S, cf\rangle \in (CPSS_{D'} \times CF)$.

$$\langle S, cf\rangle \in Cc(C) \Rightarrow Subst(S) \subseteq Cc(\texttt{SUBST}(C)).$$

### *3.3. Abstract Semantics*

We are now in position to present the abstract semantics. Note that we are not concerned with algorithmic issues here: they are dealt with in Section 4.

**Extended Abstract Behaviors.** Extended abstract behaviors are the abstract counterpart of the concrete extended behaviors defined in Section 2.6.

*Definition 3.3.* [Extended Abstract Underlying Domain]
The *extended abstract underlying domain*, denoted by $EAUD$, consists of

1. all triples $\langle \beta, g, c \rangle$, where $c$ is a clause of $P$, $g$ is a prefix of the body of $c$, $\beta \in AS_D$, and $D$ is the set of variables in the head of $c$;
2. all pairs $\langle \beta, c \rangle$, where $c$ is a clause of $P$, $\beta \in AS_D$, and $D$ is the set of variables in the head of $c$;
3. all pairs $\langle \beta, pr \rangle$, where $pr$ is a procedure of $P$ or a suffix of a procedure of $P$, $\beta \in AS_D$, and $D$ is the set of variables in the head of the clauses of $pr$.

*Definition 3.4.* [Extended Abstract Behaviors]
An *extended abstract behavior* is a function from $EAUD$ to $ASS \cup ASSC$ such that

1. every triple $\langle \beta, g, c \rangle$ from $EAUD$ is mapped to an abstract sequence with cut information $C \in ASSC_{D'}$, where $D'$ is the set of all variables in $c$;
2. every pair $\langle \beta, c \rangle$ from $EAUD$ is mapped to an abstract sequence with cut information $C \in ASSC_D$, where $D$ is the set of variables in the head of $c$;
3. every pair $\langle \beta, pr \rangle$ from $EAUD$ is mapped to an abstract sequence $B \in ASS_D$, where $D$ is the set of variables in the head of the clauses of $pr$.

The set of extended abstract behaviors is endowed with a structure of partial order in the obvious way. It is denoted by $EAB$ and its elements are denoted by *esat*.

**Abstract Transformation.** The abstract semantics is defined in terms of two semantic functions that are depicted in Figure 3.1. The first function $E : AB \to EAB$ maps abstract behaviors to extended abstract behaviors. It is the abstract counterpart of the concrete semantic rules of Figure 2.2. The second function $TAB : AB \to AB$ transforms an abstract behavior into another abstract behavior. It is the abstract counterpart of Rule **T1** in Definition 2.12.

**Abstract Semantics.** The abstract semantics is defined as the set of all abstract behaviors that are both post-fixpoints of the abstract transformation $TAB$ and pre-consistent. The corresponding definitions are given first; then the rationale underlying the definitions is discussed.

*Definition 3.5.* [Post-Fixpoints of $TAB$]
An abstract behavior $sat \in AB$ is called a *post-fixpoint* of $TAB$ if and only if $TAB(sat) \leq sat$, i.e., if and only if

$$TAB(sat)\langle \beta, p \rangle \leq sat\langle \beta, p \rangle, \quad \forall \langle \beta, p \rangle \in AUD.$$

$TAB(sat)\langle\beta,p\rangle = E(sat)\langle\beta,pr\rangle$
**where** $pr$ is the procedure defining $p$,

$E(sat)\langle\beta,pr\rangle = \texttt{SEQ}(C)$
**where** $C = E(sat)\langle\beta,c\rangle$ if $pr ::= c$

$E(sat)\langle\beta,pr\rangle = \texttt{CONC}(\beta,C,B)$
**where** $B = E(sat)\langle\beta,pr'\rangle$
$C = E(sat)\langle\beta,c\rangle$ if $pr ::= c,pr'$

$E(sat)\langle\beta,c\rangle = \texttt{RESTRC}(c,C)$
**where** $C = E(sat)\langle\beta,g,c\rangle$
$g$ is the body of $c$

$E(sat)\langle\beta,<>,c\rangle = \texttt{EXTC}(c,\beta)$

$E(sat)\langle\beta,(g,!),c\rangle = \texttt{AI-CUT}(C)$
**where** $C = E(sat)\langle\beta,g,c\rangle$

$E(sat)\langle\beta,(g,l),c\rangle = \texttt{EXTGS}(l,C,B)$
**where** $B =$ $\texttt{UNIF-VAR}(\beta')$ if $l ::= x_i{=}x_j$
$\texttt{UNIF-FUNC}(f,\beta')$ if $l ::= x_i{=}f(\ldots)$
$sat\langle\beta',p\rangle$ if $l ::= p(\ldots)$
$\beta' = \texttt{RESTRG}(l,\beta'')$
$\beta'' = \texttt{SUBST}(C)$
$C = E(sat)\langle\beta,g,c\rangle$.

**FIGURE 3.1.** The Abstract Transformation

*Definition 3.6.* [Pre-Consistent Abstract Behaviors]
Let $\longmapsto$ be the concrete semantics of the underlying program, according to Definition 2.13. An abstract behavior $sat \in AB$ is said to be *pre-consistent* with respect to $\longmapsto$ if and only if there exists a concrete behavior $\longmapsto'$ such that

$$\longmapsto' \sqsubseteq \longmapsto$$

and such that, for all $\langle\beta,p\rangle \in AUD$ and $\langle\theta,p\rangle \in CUD$,

$$\left.\begin{array}{l}\theta \in Cc(\beta),\\ \langle\theta,p\rangle \longmapsto' S\end{array}\right\} \Rightarrow S \in Cc(sat\langle\beta,p\rangle).$$

In the next section, we show that any pre-consistent post-fixpoint $sat$ of $TAB$ is a safe approximation of the concrete semantics, i.e., it is such that for all $\langle\beta,p\rangle \in AUD$ and $\langle\theta,p\rangle \in CUD$,

$$\left.\begin{array}{l}\theta \in Cc(\beta),\\ \langle\theta,p\rangle \longmapsto S\end{array}\right\} \Rightarrow S \in Cc(sat\langle\beta,p\rangle).$$

The abstract semantics is defined as the set of all pre-consistent post-fixpoints. Indeed, under the current hypotheses on the abstract domains, there is no straight-

forward way to choose a "best" abstract behavior among all pre-consistent post-fixpoints. Thus, we consider the problem of computing a reasonably accurate post-fixpoint as a pragmatic issue to be solved at the algorithmic level. In fact, the abstract interpretation algorithm presented in Section 4 is an improvement of the following construction: define the abstract behavior $sat_\perp$ by

$$sat_\perp\langle\beta, p\rangle = B_\perp, \quad \forall\langle\beta, p\rangle \in AUD.$$

Assume that the domain of abstract sequences is endowed with an upper-bound operation $\mathtt{UB} : ASS_D \times ASS_D \to ASS_D$ (not necessarily a *least* upper bound). For every $sat_1, sat_2 \in AB$, we define $\mathtt{UB}(sat_1, sat_2)$ by

$$\mathtt{UB}(sat_1, sat_2)\langle\beta, p\rangle = \mathtt{UB}(sat_1\langle\beta, p\rangle, sat_2\langle\beta, p\rangle), \quad \forall\langle\beta, p\rangle \in AUD.$$

Let j be an arbitrarily chosen natural number. An infinite sequence of pre-consistent abstract behaviors $sat_0, \ldots, sat_i, \ldots$ is defined as follows:

$$\begin{array}{lll} sat_0 & = sat_\perp, & \\ sat_{i+1} & = TAB(sat_i) & (0 \leq i < j), \\ sat_{i+1} & = \mathtt{UB}(sat_i, TAB(sat_i)) & (j \leq i). \end{array}$$

The abstract behaviors $sat_i$ are all pre-consistent because $sat_\perp$ is pre-consistent by construction, every application of $TAB$ maintains pre-consistency (as proven in the next section), and each application of UB produces an abstract behavior whose concretization contains the concretizations of the arguments. Moreover, assuming that every partial order $ASS_D$ is finite or satisfies the finite ascending chain property, the sequence $sat_0, \ldots, sat_i, \ldots$ has a least upper bound which is the desired pre-consistent post-fixpoint. In case the $ASS_D$ contains chains with infinitely many distinct elements, UB must be a widening operator [23].
The sequence from $sat_0$ to $sat_j$ is not ascending in general. In fact, $sat_\perp$ is not the minimum of $AB$ and $TAB$ is not necessarily monotonic nor extensive (i.e., $sat \leq TAB(sat)$ does not always hold). From step 0 to $j$, the computation of the $sat_i$ simulates as closely as possible the computation of the least fixpoint of the concrete transformation. From step $j$ to convergence, all iterates are "lumped" together. All concrete behaviors $\longmapsto_j, \longmapsto_{j+1}, \ldots$ of the Kleene sequence of the concrete semantics, are thus included in the concretization of the final post-fixpoint $sat$. So $sat$ describes properties that are true not only for the concrete $\longmapsto$ semantics but also for its approximations $\longmapsto_j, \longmapsto_{j+1}, \ldots$. The choice of $j$ is a compromise: a low value ensures a faster convergence while a high value provides a better accuracy. The abstract interpretation algorithm presented in Section 4 does not iterate globally over $TAB$. It locally iterates over $E$ for every needed input pattern $\langle\beta, p\rangle$ and uses different values of $j$ for different input patterns. Depending on the particular abstract domain, the value can be guessed more or less cleverly. This is the role of the special widening operator of Definition 4.1. A sample widening operator is described in Section 5.2, showing how the value of $j$ can be guessed in the case of a practical abstract domain.

### *3.4. Safety of the Abstract Semantics*

We prove here the safety of our abstract semantics. First, we formally define the notion of safe approximation. Then, we show that the abstract transformation is

safe in the sense that, whenever *sat* safely approximates $\longmapsto$, $TAB(sat)$ safely approximates $\overset{TCB}{\longmapsto}$ (Theorem 3.1). From this basic result, we deduce that $TAB$ transforms pre-consistent abstract behaviors into other pre-consistent abstract behaviors (Theorem 3.2), and that, when *sat* is a post-fixpoint of the abstract transformation which safely approximates a concrete behavior $\longmapsto$, it also safely approximates the concrete behavior $\overset{TCB}{\longmapsto}$ (Theorem 3.3). Theorem 3.4 states that abstract behaviors are, roughly speaking, chain-closed with respect to concrete behaviors. Finally, Theorem 3.5 states our main result, i.e., every pre-consistent post-fixpoint of the abstract transformation safely approximates the concrete semantics.

*Definition 3.7.* [Safe Approximation]
Let $\longmapsto \in CB$ and $sat \in AB$. The abstract behavior *sat safely approximates* the concrete behavior $\longmapsto$ if and only if, for all $\langle \theta, p\rangle \in CUD$ and $\langle \beta, p\rangle \in AUD$, the following implication holds:

$$\left.\begin{array}{c} \theta \in Cc(\beta), \\ \langle \theta, p\rangle \longmapsto S \end{array}\right\} \Rightarrow S \in Cc(sat\langle \beta, p\rangle).$$

Similarly, let $\longmapsto \in ECB$ and $esat \in EAB$. The extended abstract behavior *esat safely approximates* $\longmapsto$ if and only if, for all $\langle \theta, pr\rangle, \langle \theta, c\rangle, \langle \theta, g, c\rangle \in ECUD$ and $\langle \beta, pr\rangle, \langle \beta, c\rangle, \langle \beta, g, c\rangle \in EAUD$, the following implications hold:

$$\left.\begin{array}{c} \theta \in Cc(\beta), \\ \langle \theta, pr\rangle \longmapsto S \end{array}\right\} \Rightarrow S \in Cc(esat\langle \beta, pr\rangle),$$

$$\left.\begin{array}{c} \theta \in Cc(\beta), \\ \langle \theta, c\rangle \longmapsto \langle S, cf\rangle \end{array}\right\} \Rightarrow \langle S, cf\rangle \in Cc(esat\langle \beta, c\rangle),$$

$$\left.\begin{array}{c} \theta \in Cc(\beta), \\ \langle \theta, g, c\rangle \longmapsto \langle S, cf\rangle \end{array}\right\} \Rightarrow \langle S, cf\rangle \in Cc(esat\langle \beta, g, c\rangle).$$

*Theorem 3.1.* [Safety of the Abstract Transformation]
*Let $\longmapsto \in CB$ and $sat \in AB$. If sat safely approximates $\longmapsto$, then $TAB(sat)$ safely approximates $\overset{TCB}{\longmapsto}$.*

We first establish the following result. Remember that if $\longmapsto \in CB$, its extension in $ECB$ is also denoted by $\longmapsto$ (see Section 2.6).

*Lemma 3.1.* [Safety of $E$]
*Let $\longmapsto \in CB$ and $sat \in AB$. If sat safely approximates $\longmapsto$, then $E(sat)$ safely approximates $\longmapsto$ (the extension of $\longmapsto$ in ECB).*

PROOF. [of Lemma 3.1]
The proof follows by structural induction on the syntax of the underlying program. It uses the concrete semantic rules of Figure 2.2, the definition of $E$ in Figure 3.1, and the specifications of the abstract operations given in Section 3.2. The proof is straightforward due to the close correspondence of the concrete and the abstract semantics. We only detail the reasoning for the base case and for the case of a goal $(g, l)$ where $l$ is an atom of the form $p(x_{i_1}, \ldots, x_{i_n})$. The other cases are similar.

**Base case**. Let $\langle\theta, <>, c\rangle \in ECUD$ and $\langle\beta, <>, c\rangle \in EAUD$. Assume that $\theta \in Cc(\beta)$ and $\langle\theta, <>, c\rangle \longmapsto \langle S, cf\rangle$. It must be proven that

$$\langle S, cf\rangle \in Cc(E(sat)\langle\beta, <>, c\rangle).$$

This relation holds because of the three following facts:

$$\begin{array}{rcll} \langle S, cf\rangle & = & \texttt{EXTC}(c, \theta) & \text{(by } \mathbf{R2}\text{)}, \\ \texttt{EXTC}(c, \theta) & \in & Cc(\texttt{EXTC}(c, \beta)) & \text{(by specification of EXTC)}, \\ E(sat)\langle\beta, <>, c\rangle & = & \texttt{EXTC}(c, \beta) & \text{(by definition of } E\text{)}. \end{array}$$

**Induction step**. Let $\langle\theta, (g, l), c\rangle \in ECUD$ and $\langle\beta, (g, l), c\rangle \in EAUD$, where $l$ is an atom of the form $p(x_{i_1}, \ldots, x_{i_n})$. Assume that $\theta \in Cc(\beta)$ and $\langle\theta, (g, l), c\rangle \longmapsto \langle S, cf\rangle$. It must be proven that

$$\langle S, cf\rangle \in Cc(C), \text{ where } C = E(sat)\langle\beta, (g, l), c\rangle.$$

By Rule **R6**, there exist program substitutions and program sequences such that

$$\begin{array}{lll} \langle\theta, g, c\rangle \longmapsto \langle S', cf\rangle & & (C1) \\ S' =< \theta_1, \ldots, \theta_i, \_ > & & (C2) \\ \theta'_k = \texttt{RESTRG}(l, \theta_k) & (1 \leq k \leq Ns(S)) & (C3) \\ \langle\theta'_k, p\rangle \longmapsto S'_k & (1 \leq k \leq Ns(S)) & (C4) \\ S_k = \texttt{EXTG}(l, \theta_k, S'_k) & (1 \leq k \leq Ns(S)) & (C5) \\ S = \square_{k=1}^{Ne(S)} S_k & & (C6) \end{array}$$

Moreover, by definition of $E(sat)$, there exist abstract values such that

$$\begin{array}{ll} C = \texttt{EXTGS}(l, C', B) & (A1) \\ B = sat\langle\beta', p\rangle & (A2) \\ \beta' = \texttt{RESTRG}(l, \beta'') & (A3) \\ \beta'' = \texttt{SUBST}(C') & (A4) \\ C' = E(sat)\langle\beta, g, c\rangle & (A5) \end{array}$$

The following assertions hold. By $A5$, $C1$, and the induction hypothesis,

$$\langle S', cf\rangle \in Cc(C') \ \ (B1).$$

By $A4$, $B1$, $C2$, and the specification of SUBST,

$$\theta_k \in Cc(\beta'') \ (1 \leq k \leq Ns(S)) \ \ (B2).$$

By $A3$, $B2$, $C3$, and the specification of RESTRG,

$$\theta'_k \in Cc(\beta') \ (1 \leq k \leq Ns(S)) \ \ (B3).$$

By $A2$, $B3$, $C4$, and the hypothesis that *sat* safely approximates $\longmapsto$,

$$S'_k \in Cc(B) \ (1 \leq k \leq Ns(S)) \ \ (B4).$$

Finally, by $A1$, $B1$, $B4$, $C2$, $C5$, $C6$, and specification of EXTGS,

$$\langle S, cf\rangle \in Cc(C).$$

■

PROOF. [of Theorem 3.1]
The result follows from the definition of *TAB* in Figure 3.1, the definition of *TCB* in Section 2.12, and Lemma 3.1. ■

The next theorem states that the transformation *TAB* maintains pre-consistency.

*Theorem 3.2. Let $sat \in AB$. If sat is pre-consistent, then $TAB(sat)$ is also pre-consistent.*

PROOF. Let $\longmapsto$ be the concrete semantics of the underlying program. Since *sat* is pre-consistent, there exists a concrete behavior $\longmapsto'$ such that

1. $\longmapsto' \sqsubseteq \longmapsto$, and
2. *sat* safely approximates $\longmapsto'$.

The first condition implies that

$$\overset{TCB}{\longmapsto}' \sqsubseteq \longmapsto,$$

since *TCB* is monotonic and $\overset{TCB}{\longmapsto} = \longmapsto$.
The second condition and Theorem 3.1 imply that

$$TAB(sat) \text{ safely approximates } \overset{TCB}{\longmapsto}'.$$

The result follows from the two implied statements and Definition 3.6. ■

The next two theorems state closure properties of abstract behaviors, which are used to prove the safety of the abstract semantics.

*Theorem 3.3. Let sat be a post-fixpoint of TAB. Let $\longmapsto \in CB$. If sat safely approximates $\longmapsto$, then sat also safely approximates $\overset{TCB}{\longmapsto}$.*

PROOF. Let *sat* safely approximate $\longmapsto$. Let $\langle \theta, p \rangle \in CUD$ and $\langle \beta, p \rangle \in AUD$. It must be proven that

$$\left.\begin{array}{c} \theta \in Cc(\beta), \\ \langle \theta, p \rangle \overset{TCB}{\longmapsto} S \end{array}\right\} \Rightarrow S \in Cc(sat\langle \beta, p \rangle).$$

Assume that the left part of the implication holds. Theorem 3.1 implies that

$$S \in Cc(TAB(sat)\langle \beta, p \rangle).$$

Since *sat* is a post-fixpoint and *Cc* is monotonic,

$$Cc(TAB(sat)\langle \beta, p \rangle) \subseteq Cc(sat\langle \beta, p \rangle),$$

and then

$$S \in Cc(sat\langle \beta, p \rangle).$$

■

*Theorem 3.4. Let $(\longmapsto_i)_{i \in \mathbf{N}}$ be a chain of concrete behaviors. Let $sat \in AB$. If sat safely approximates $\longmapsto_i$, for all $i \in \mathbf{N}$, then sat safely approximates $(\sqcup_{i=0}^{\infty} \longmapsto_i)$.*

PROOF. Let us abbreviate $(\sqcup_{i=0}^{\infty} \longmapsto_i)$ by $\longmapsto$. It is sufficient to prove that, for any $\langle \beta, p \rangle \in AUD$ and any $\langle \theta, p \rangle \in CUD$,

$$\left.\begin{array}{l} \theta \in Cc(\beta), \\ \langle \theta, p \rangle \longmapsto S \end{array}\right\} \Rightarrow S \in Cc(sat\langle \beta, p \rangle).$$

Fix $\langle \beta, p \rangle$, $\langle \theta, p \rangle$, and $S$ satisfying the left part of the implication. By Theorem 2.1,

$$S = \sqcup_{i=0}^{\infty} S_i \quad \text{where} \quad \langle \theta, p \rangle \longmapsto_i S_i \ \forall i \in \mathbf{N}.$$

Since *sat* safely approximates every $\longmapsto_i$,

$$S_i \in Cc(sat\langle \beta, p \rangle) \text{ for all } i \in \mathbf{N}.$$

Finally, since $Cc(sat\langle \beta, p \rangle)$ is chained-closed,

$$S \in Cc(sat\langle \beta, p \rangle).$$

■

The last theorem states our main result.

*Theorem 3.5. [Safety of the Abstract Semantics]*
*Let sat be a pre-consistent post-fixpoint of TAB. Then sat safely approximates* $\longmapsto$ *where* $\longmapsto$ *is the concrete semantics of the underlying program.*

We first establish the following statement.

*Lemma 3.2. Let sat be a pre-consistent post-fixpoint of TAB. There exists a chain of concrete behaviors* $(\longmapsto_i)_{i \in \mathbf{N}}$ *such that sat safely approximates* $\longmapsto_i$*, for all* $i \in \mathbf{N}$ *and* $(\sqcup_{i=0}^{\infty} \longmapsto_i) = \longmapsto$ *where* $\longmapsto$ *is the concrete semantics of the underlying program.*

PROOF. [of Lemma 3.2]
The proof is in three steps. First we construct a sequence $\{\longmapsto'_i\}_{i \in \mathbf{N}}$ of lower-approximations of $\longmapsto$ which is not necessarily a chain; then we modify it to get a chain $(\longmapsto_i)_{i \in \mathbf{N}}$; finally, we show that $(\sqcup_{i=0}^{\infty} \longmapsto_i) = \longmapsto$. The proof uses the following property of program substitution sequences, whose proof is left to the reader. If $S_1$, $S_2$ and $S$ are program substitution sequences such that $S_1 \sqsubseteq S$ and $S_2 \sqsubseteq S$, then $S_1$ and $S_2$ have a least upper-bound, which is either $S_1$ or $S_2$. The least upper-bound is denoted by $S_1 \sqcup S_2$ in the proof.

1. Since *sat* is pre-consistent, there exists a concrete behavior $\longmapsto'$ such that *sat* safely approximate $\longmapsto'$ and $\longmapsto' \sqsubseteq \longmapsto$. The sequence $\{\longmapsto'_i\}_{i \in \mathbf{N}}$ is defined by
$$\longmapsto'_0 = \longmapsto' \quad \text{and} \quad \longmapsto'_{i+1} = \overset{TCB}{\longmapsto'_i} \ (i \in \mathbf{N}).$$
Since $\longmapsto' \sqsubseteq \longmapsto$, *TCB* is monotonic and $\longmapsto$ is a fixpoint, it follows that
$$\longmapsto'_i \sqsubseteq \longmapsto \ (\forall i \in \mathbf{N}).$$
Moreover, by Theorem 3.3, *sat* safely approximates every $\longmapsto'_i$.

2. $(\longmapsto_i)_{i \in \mathbf{N}}$ is now constructed by induction over $i$. The correctness of the construction process requires to prove that, after each induction step, the relation $\longmapsto_i \sqsubseteq \longmapsto$ holds. We first define
$$\longmapsto_0 = \longmapsto'_0 .$$

Let $i \in \mathbf{N}$. Assume, by induction, that $\longmapsto_0 \sqsubseteq \ \ldots \ \sqsubseteq \longmapsto_i \sqsubseteq \longmapsto$. For every $\langle \theta, p \rangle \in CUD$, we define

$$\langle \theta, p \rangle \longmapsto_{i+1} (S_1 \sqcup S_2) \text{ where } \begin{cases} \langle \theta, p \rangle \longmapsto_i S_1, \\ \langle \theta, p \rangle \longmapsto'_{i+1} S_2. \end{cases}$$

Since $\longmapsto'_{i+1} \sqsubseteq \longmapsto$ and $\longmapsto_i \sqsubseteq \longmapsto$, we have that $\longmapsto_{i+1}$ is well-defined and $\longmapsto_{i+1} \sqsubseteq \longmapsto$. Moreover, since *sat* safely approximates $\longmapsto_i$ (by induction) and $\longmapsto'_{i+1}$, and $S_1 \sqcup S_2$ is equal either to $S_1$ or $S_2$, in the definition of $\longmapsto_{i+1}$, we have that *sat* safely approximates every $\longmapsto_{i+1}$.

3. The Kleene sequence of the concrete semantics is a chain $(\longmapsto''_i)_{i \in \mathbf{N}}$ defined as follows:
$$\longmapsto''_0 = \longmapsto_\bot \text{ and } \longmapsto''_{i+1} = \overset{TCB}{\longmapsto}''_i \ (i \in \mathbf{N}).$$
Since $\longmapsto_\bot \sqsubseteq \longmapsto'$ and $TCB$ is monotonic, it follows, by induction, that
$$\longmapsto''_i \sqsubseteq \longmapsto'_i \sqsubseteq \longmapsto_i \sqsubseteq \longmapsto \ (\forall i \in \mathbf{N}).$$
Therefore, by definition of the least upper bound and since the least fixpoint is the limit of the Kleene sequence,
$$\longmapsto = \ (\sqcup_{i=0}^{\infty} \longmapsto''_i) \ \sqsubseteq (\sqcup_{i=0}^{\infty} \longmapsto_i) \ \sqsubseteq \longmapsto .$$
Thus,
$$\longmapsto = \ (\sqcup_{i=0}^{\infty} \longmapsto_i).$$

■

PROOF. [of Theorem 3.5]
The result is an immediate consequence of Theorem 3.4 and Lemma 3.2 ■

## *3.5. Related Works*

In this section we first discuss the mathematical approach underlying our abstract semantics and relate it with the higher-order abstract interpretation frameworks advocated by P. Cousot and R. Cousot in [24]. Then, we compare our approach with the abstract semantics for Prolog with control proposed by R. Barbuti *et al.* in [4], by G. Filé and S. Rossi in [34], and by F. Spoto and G. Levi in [84].

**Cousot and Cousot's Higher-order Abstract Interpretation Frameworks.** As mentioned in the introduction, the traditional approach to abstract interpretation can not be applied to approximate the concrete semantics of Section 2. Indeed, we can define a set-based collecting transformation by lifting the concrete semantics to sets of program substitution sequences. However, the least fixpoint of the collecting transformation does not safely approximate the concrete semantics. The problem can be solved by restricting to sets of $\wp(CPS_D)$ and $\wp(CPSS_D)$ that enjoy some closure properties ensuring safeness of the least fixpoint. This solution is similar to the choice of a power-domain structure in denotational semantics [82, 86]: the needed constructions can in fact be viewed as power-domains. However there is no best way to choose the closure properties. Different closure properties are adequate for different sorts of information. It is therefore advocated by P. Cousot

and R. Cousot in [24] that, for higher-order languages, different collecting semantics should be defined for the same language depending on the kind of properties to be inferred. In our case, at least two dual collecting semantics could be defined. Both of them use sets of program substitution sequences that are chain-closed.

1. The first semantics considers *downwards-closed* sets of program substitution sequences, i.e., such that for any $S, S' \in CPSS_D$,

$$\left.\begin{array}{l} S \in \Sigma, \\ S' \sqsubseteq S \end{array}\right\} \Rightarrow S' \in \Sigma.$$

   This domain is ordered by inclusion and its minimum is $\{< \bot >\}$. It is adequate to infer non-termination and upper bounds to the length of sequences. In particular, it is adequate for determinacy analysis. However, it is unable to infer termination since $< \bot >$ belongs to any set of sequences.

2. The second semantics considers *upwards-closed* sets of program substitution sequences, i.e., such that for any $S, S' \in CPSS_D$,

$$\left.\begin{array}{l} S \in \Sigma, \\ S \sqsubseteq S' \end{array}\right\} \Rightarrow S' \in \Sigma.$$

   This domain is ordered by $\Sigma \leq \Sigma' \Leftrightarrow \Sigma' \subseteq \Sigma$ and its minimum is $CPSS_D$. It is able to infer termination and lower bounds to the length of sequences. It is less adequate than the previous one to infer precise information about the substitutions in the sequences because its least fixpoint corresponds to a greatest fixpoint in a traditional framework ignoring the sequence structure.

In both cases, the least fixpoint is well-defined because the collecting versions of the operations are monotonic, since they have to ensure the closure properties. Moreover, the least fixpoint of the collecting semantics safely approximates the concrete semantics because all iterates are pre-consistent and the sets are chain-closed. Nevertheless, our formalization has some advantages.

1. It can be more efficient: a single analysis is able to infer all the information that can be inferred by the two collecting semantics.
2. It can be more accurate: there are pre-consistent post-fixpoints that are more precise than the intersection of the two collecting semantics.

**Barbuti et al.'s Abstract Semantics**. The abstract semantics proposed by R. Barbuti *et al.* in [4] aims at modeling control aspects of logic programs such as search strategy and selection rule. Their semantics is parametric with respect to a "termination theory". The meaning of a program is obtained by composing the meaning of its "logic component" together with a corresponding "termination theory" (the "control component"). The latter can be provided either by applying techniques of abstract interpretation or by applying proof procedures. In all cases, control information is deduced from outside in the form of a separated termination analysis. This is the main difference with our framework, where control information, i.e. information relative to termination or non-termination, is modeled within the semantic domains through the notion of substitution sequence.

**Filé and Rossi's Abstract Interpretation Framework**. The framework proposed by G. Filé and S. Rossi in [34] consists of a tabled interpreter which explores OLDT abstract trees decorated with control information about sure success or failure of the goals. Such information is used by the cut operation to prune the OLDT-tree whenever a cut is reached. Sure success is modeled in our framework by abstract sequences representing only non-empty sequences. The abstract semantics defined by Filé and Rossi is operational and non-compositional while ours is compositional and based on the fixpoint approach. Moreover, the abstract execution of a goal $(g, !)$ is different. Whenever is known that $g$ surely succeeds, their framework stops after generating the first "sure" solution, while ours computes the entire abstract sequence for $g$ and then cuts it to maintain at most one solution. Our approach may thus imply some redundant work. However, if $g$ is used in several contexts, their framework should recognize this situation and expand the OLDT-tree further.

**Spoto and Levi's Denotational Abstract Semantics**. The related work closest to ours is the denotational abstract semantics proposed by F. Spoto and G. Levi in [84]. They define a goal-independent and compositional abstract semantics of Prolog modeling the depth-first search rule and the cut. Their semantics associates to any Prolog program a sequence of pairs consisting of a "kernel" constraint and its "observability" part. Intuitively, kernel constraints denote computed answers, while observability constraints give information about divergent computations and cut executions. The main difference with our approach is that their semantics is goal-independent while ours is not. This is due to the fact that our abstract semantics is *functional*, i.e., it associates to each program P a function (an abstract behavior) mapping every pair $\langle \beta, p \rangle$ to an abstract sequence $B$. However, this choice is unrelated to our concrete semantics: we could as well abstract the concrete semantics by a *relational* abstract semantics [22], making it possible to express dependencies between input substitutions and the corresponding output substitution sequences. This is the approach of [56] where we express dependencies between the size of input terms and the number of corresponding output substitutions. We will go back to this issue at the end of Section 6.2.

## 4. GENERIC ABSTRACT INTERPRETATION ALGORITHM

A generic abstract interpretation algorithm is an algorithm that is parametric with respect to the abstract domains. It can be instantiated by various domains to obtain different data-flow analyses. Several such algorithms have been proposed for Prolog [8, 32, 53, 54, 61, 62, 71, 76], but they do not handle the control features of the language such that Prolog search rule and cut.

The algorithm presented here is essentially an instantiation of the universal fixpoint algorithm described in [60] to the abstract semantics of Section 3. In particular, it is quite similar to the algorithm presented in [53, 61]: in fact, the abstract semantics of Section 3 can be viewed as a proper generalization of the abstract semantics described in [53, 61], where the sequences of computed answer substitutions are no longer abstracted to sets of substitutions.
The universal algorithm in [60] is top-down, i.e., it computes a subset of the fixpoint (in the form of a set of tuples) containing the output value corresponding to a distinguished input together with all the tuples needed to compute it. Top-down

algorithms are naturally used to perform data-flow analyses, where one is interested in collecting the abstract information corresponding to a class of initial queries described by the distinguished input. It is more efficient in general to compute a part of the fixpoint only and this allows one to use infinite abstract domains, which are more expressive [23]. Although the instantiation of [60] to our abstract semantics is as mechanical as in our previous works (a slightly more general widening operator is needed however), the correctness of the algorithm involves some new theoretical issues: the pre-consistency of the post-fixpoint has now to be proven. Nevertheless, since the novel algorithm is in practice very similar to the algorithm presented in [61], we only discuss here the extended widening operator which ensures a good compromise between efficiency and accuracy. A detailed description of the algorithm and its correctness proof can be found in [59].

### *4.1. Extended Widening*

The extended widening operation used by the novel algorithm is defined as follows.

*Definition 4.1.* [Extended Widening] An *extended widening* on abstract sequences is a (polymorphic[1]) operation $\nabla : ASS_D \times ASS_D \to ASS_D$ that enjoys the following properties. Let $\{B_i\}_{i\in\mathbf{N}}$ be a sequence of elements of $ASS_D$. Consider the sequence $\{B'_i\}_{i\in\mathbf{N}}$ defined by

$$\begin{aligned} B'_0 &= B_0, \\ B'_{i+1} &= B_{i+1} \nabla B'_i \ (i \in \mathbf{N}). \end{aligned}$$

The following conditions hold:

1. $B'_i \geq B_i$ $(i \in \mathbf{N})$;
2. the sequence $\{B'_i\}_{i\in\mathbf{N}}$ is stationary, i.e., there exists $j \geq 0$ such that $B'_i = B'_j$ for all $i$ such that $j \leq i$.

An extended widening is slightly more general than a widening [23] because the sequence $\{B'_i\}_{i\in\mathbf{N}}$ is *not* required to be a chain.

Let us now explain how the extended widening is used by the algorithm. Given an input pair $\langle\beta, p\rangle$, the algorithm iterates on the computation of $TAB(sat)\langle\beta, p\rangle$ until convergence, and concurrently updates *sat*, as follows (recursive calls – which also modify *sat* – are ignored in the discussion):

1. $B'_0 = B_\perp$ is stored in the initial *sat* as the output for $\langle\beta, p\rangle$;
2. $B_i$ results from the *i-th* execution of $TAB(sat)\langle\beta, p\rangle$;
3. $B'_i = B_i \nabla B'_{i-1}$ is stored in the current *sat* after the *i-th* execution of $TAB(sat)\langle\beta, p\rangle$;
4. the loop is exited when $B_{i+1} \leq B'_i$.

The loop terminates because there must be some $i$ such that $B'_{i+1} = B'_i$ (otherwise Condition 2 of Definition 4.1 would be violated), and, hence, $B_{i+1} \leq B'_i$ since $B'_{i+1} \geq B_{i+1}$ by Condition 1. The loop can be resumed later on because some

[1] It is parametrized over $D$.

values in *sat* have been updated (Step 1 is omitted in these subsequent executions); all re-executions of the loop terminate for the same reasons as the first one; moreover, the loop can only be resumed finitely many times because no element in *sat* can be improved infinitely many often, since there is a $j$ such that $B'_i = B'_j$ for all $i$ greater or equal to $j$. Note that a local post-fixpoint is attained each time the loop is exited. Thus a global post-fixpoint is obtained when all loops are terminated for all values in *sat*. The formal characterization of Definition 4.1 elegantly captures the idea that the algorithm sticks as closely as possible to the abstract semantics during the first iterations, and starts lumping the results together only when enough accuracy is obtained, in order to ensure convergence. The advantage of this characterization is that no particular value of $j$ is fixed. So we can think of "intelligent" extended widenings that observe how the successive iterates behave and that enforce convergence exactly at the right time. The extended widening used in our experimental evaluation is based on this intuitive idea (see Section 5.2).

## 5. CARDINALITY ANALYSIS

The abstract interpretation framework for Prolog presented in previous sections has been instantiated by a domain of abstract sequences to perform so-called cardinality analysis [6]. Cardinality analysis approximates the number of solutions to a goal and is useful for many purposes such as indexing, cut insertion and elimination [27, 81], dead code elimination, and memory management and scheduling in parallel systems [9, 40]. The analysis subsumes traditional determinacy analysis [25, 27, 39, 81].

This section is organized as follows. First we describe how a generic abstract domain for cardinality analysis, which is parametric with respect to *any* domain of abstract substitutions, can be built. Then, we instantiate this generic domain to the domain of abstract substitutions `Pattern` [61]. Finally, we discuss experimental evaluations of the analysis from both accuracy and efficiency standpoints.

### *5.1. Generic Abstract Domains for Cardinality Analysis*

In this section, generic domains of abstract sequences and abstract sequences with cut information are built. The domains are generic with respect to the information on the substitutions in the sequences, but they provide specific information about the sequence structure. The latter consists of lower and upper bounds to the number of substitutions in the sequences and information about the nature (i.e., finite, incomplete or infinite) of the sequences. This information allows us to perform non-termination analysis and a limited form of termination analysis. Predicate level analyses, like determinacy and functionality [30], which were previously considered falling outside the scope of abstract interpretation, can be performed.

**Abstract Substitutions.** The substitution part of our generic domain of abstract sequences is assumed to be an element of an arbitrary domain of abstract substitutions $AS_D$. The only requirement on $AS_D$ is that it contains a minimum element $\beta_\emptyset$ such that $Cc(\beta_\emptyset) = \emptyset$. An abstract domain can always be enhanced with such an element.

**Abstract Sequences.** The generic domain of abstract sequences manipulates termination information whose domain is defined below.

*Definition 5.1.* [Termination Information]
A termination information $t$ is an element of the set $TI = \{st,\ snt,\ pt\}$ endowed with the ordering $\leq$ defined by

$$t_1 \leq t_2 \Leftrightarrow \text{either } t_1 = t_2 \text{ or } t_2 = pt \quad \forall t_1, t_2 \in TI.$$

The symbol *st* stands for "sure termination" and it characterizes finite sequences; *snt* stands for "sure non termination" and characterizes incomplete and infinite sequences; *pt* stands for "possible termination" and corresponds to absence of information. The domain of abstract substitution sequences is defined as follows.

*Definition 5.2.* [Abstract Sequences]
Let $D$ be a finite set of program variables. We denote by $ASS_D$ the set of all 4-tuples $\langle \beta, m, M, t \rangle$ such that $\beta \in AS_D$, $m \in \mathbf{N}$, $M \in \mathbf{N} \cup \{\infty\}$, and $t \in TI$.

Informally, $\beta$ describes all substitutions in the sequences, $m$ and $M$ are lower and upper bounds on the number of substitutions in the sequences, and $t$ is an information on termination. The ordering on abstract sequences is defined as follows.

*Definition 5.3.* [Ordering on Abstract Sequences]
Let $B_1, B_2 \in ASS_D$.

$$B_1 \leq B_2 \text{ iff } \beta_1 \leq \beta_2 \text{ and } m_1 \geq m_2 \text{ and } M_1 \leq M_2 \text{ and } t_1 \leq t_2.$$

The set of program substitution sequences described by an abstract sequence $B$ is formally defined as follows.

*Definition 5.4.* [Concretization for Abstract Sequences]
Let $B{=}\langle \beta, m, M, t \rangle \in ASS_D$. We define

$Cc(B) = Sseq_1(\beta) \cap Sseq_2(m, M) \cap Sseq_3(t)$
where

$Sseq_1(\beta) = \{S : S \in PSS_D \text{ and } Subst(S) \subseteq Cc(\beta)\}$,
$Sseq_2(m, M) = \{S : S \in PSS \text{ and } m \leq Ns(S) \leq M\}$,
$Sseq_3(snt) = \{S : S \in PSS \text{ and } S \text{ is incomplete or infinite}\}$,
$Sseq_3(st) = \{S : S \in PSS \text{ and } S \text{ is finite}\}$,
$Sseq_3(pt) = PSS$.

Monotonicity of the concretization function is a simple consequence of the definition. We denote by $B_\perp$ the special abstract sequence $\langle \beta_\emptyset, 0, 0, snt \rangle$ which is such that $Cc(B_\perp) = \{< \perp >\}$ as required in Section 3.1. It is easy to prove that for all abstract sequences $B \in ASS_D$, the set $Cc(B)$ is chain-closed (see [59]).

**Abstract Sequences with Cut Information.** Abstract sequences with cut information are obtained by enhancing abstract sequences with information about execution of cuts.
Let us first define the abstract domain for cut information.

*Definition 5.5.* [Abstract Cut Information]
An abstract cut information $acf$ is an element of the set $ACF = \{cut, nocut, weakcut\}$.

*Definition 5.6.* [Abstract Sequences with Cut Information]
Let $D$ be a finite set of program variables. We denote by $ASSC_D$ the set of pairs $\langle B, acf\rangle$ where $B \in ASS_D$ and $acf \in ACF$.

Informally, *cut* indicates that a cut has been executed in all sequences, *nocut* that no cut has been executed in any sequence, and *weakcut* that a cut has been executed for all sequences producing at least one solution. More formally, the concretization of an abstract sequence with cut information is defined as follows.

*Definition 5.7.* [Concretization for Abstract Sequences with Cut Information]
Let $B \in ASS_D$. We define

$$\begin{array}{ll} Cc(\langle B, cut\rangle) & = \{\langle S, cut\rangle : S \in Cc(B)\}, \\ Cc(\langle B, nocut\rangle) & = \{\langle S, nocut\rangle : S \in Cc(B)\}, \\ Cc(\langle B, weakcut\rangle) & = \{\langle S, cut\rangle : S \in Cc(B)\} \cup \\ & \quad \{\langle S, nocut\rangle : \ S \in Cc(B) \text{ and } S \in \{<>, <\bot>\}\}. \end{array}$$

## *5.2. Abstract Operations*

Our next task is to provide definitions of all abstract operations specified in Section 3.2. For space reasons, we describe here a subset of the operations, i.e., extended widening, unification, operation treating cut, and concatenation. The other operations are described in the appendix. The reader is referred to [59] for the correctness proofs.
The operations on abstract substitutions which are used in the definition of the operations on abstract sequences will be recalled when needed.

**Extended Widening:** $\nabla : ASS_D \times ASS_D \to ASS_D$
We require that the abstract domain $AS_D$ is equipped with a widening operation $\nabla' : AS_D \times AS_D \to AS_D$. It can be an extended widening, a normal widening, or, if $AS_D$ is finite or enjoys the finite ascending chain property, any upper bound operation. The widening on sequences is obtained by taking the least upper bound of the termination components, the minimum of the lower bounds and setting the upper bound to infinity.
Assume that $B_{old} = \langle \beta_{old}, m_{old}, M_{old}, t_{old}\rangle$ and $B_{new} = \langle \beta_{new}, m_{new}, M_{new}, t_{new}\rangle$.
The operation $\nabla : ASS_D \times ASS_D \to ASS_D$ is defined as follows.

$$\begin{array}{lll} B_{new} \nabla B_{old} = \langle \beta_{new} \nabla' \beta_{old}, m_{new}, M_{new}, t_{new}\rangle & \text{if} & \beta_{new} \not\leq \beta_{old} \\ = \langle \beta_{old}, m_{new}, M_{new}, pt\rangle & \text{if} & \beta_{new} \leq \beta_{old} \text{ and } t_{new} \not\leq t_{old} \\ = \langle \beta_{old}, \min(m_{new}, m_{old}), \infty, t_{old}\rangle & \text{if} & \beta_{new} \leq \beta_{old} \text{ and } t_{new} \leq t_{old} \text{ and} \\ & & (m_{new} < m_{old} \text{ or } M_{new} > M_{old}) \\ = B_{old} & \text{if} & B_{new} \leq B_{old}. \end{array}$$

The first case makes sure that the algorithm iterates until the abstract substitution part stabilizes. When it is stable, the widening is applied on sequences.

*Example.* Consider the following program:

```
repeat.
repeat :- repeat.
```

The concrete semantics of this program maps the input $\langle \epsilon, \texttt{repeat} \rangle$, where $\epsilon$ is the empty substitution, to the infinite sequence $< \epsilon, \ldots, \epsilon, \ldots >$.
On this example, because the program has no variables, our domain of abstract substitutions only contains two values, say $\beta_\emptyset$ and $\beta_\top$, such that

$$Cc(\beta_\emptyset) = \emptyset$$
$$Cc(\beta_\top) = \{\epsilon\}.$$

Let $B_\perp = \langle \beta_\emptyset, 0, 0, snt \rangle$. Starting from $B_\perp$, the algorithm computes the abstract sequences

$$B_0 = B_\perp \qquad B'_0 = B_\perp$$
$$B_1 = \langle \beta_\top, 1, 1, snt \rangle \qquad B'_1 = B_1 \nabla B'_0 = \langle \beta_\top, 1, 1, snt \rangle$$
$$B_2 = \langle \beta_\top, 2, 2, snt \rangle \qquad B'_2 = B_2 \nabla B'_1 = \langle \beta_\top, 1, \infty, snt \rangle$$
$$B_3 = \langle \beta_\top, 2, \infty, snt \rangle$$

Notice that the widening on sequences is applied when the abstract substitution part stabilizes, i.e., after the computation of the abstract sequence $B_2$. The next iterate $B_3$ satisfies the property that $B_3 \leq B'_2$. Hence, according to the discussion in Section 4.1, the execution terminates returning the final value

$$B'_2 = \langle \beta_\top, 1, \infty, snt \rangle.$$

Observe that $B'_2$ safely approximates the concrete infinite sequence $< \epsilon, \ldots, \epsilon, \ldots >$. Moreover, it expresses the fact that the execution of `repeat` *surely* succeeds at least once and *surely* does not terminate.

**Unification of Two Variables:** `UNIF-VAR`: $AS_{\{x_1,x_2\}} \rightarrow ASS_{\{x_1,x_2\}}$
Given an abstract substitution $\beta$ with domain $\{x_1, x_2\}$, this operation returns an abstract sequence which represents a set of substitution sequences of length 0 or 1 (depending upon the success or failure of the unification). The terms bound to $x_1$ and $x_2$ are unified in all these sequences. The operation `UNIF-VAR` on abstract sequences uses an upgraded version of the operation `UNIF-VAR` on abstract substitutions defined in [53, 61]. The latter, in addition to the resulting abstract substitution, produces now two flags indicating whether the unification always succeeds, always fails, or can both succeed and fail. The additional information is expressed by the boolean values *ss* and *sf* as specified below.

*Operation* `UNIF-VAR`: $AS_{\{x_1,x_2\}} \rightarrow (AS_{\{x_1,x_2\}} \times Bool \times Bool)$
Let $\beta \in AS_{\{x_1,x_2\}}$ and $\langle \beta', ss, sf \rangle = \texttt{UNIF-VAR}(\beta)$. The following conditions hold:

1. $\forall \theta \in Cc(\beta) : \forall \sigma \in SS : (\sigma \in mgu(x_1\theta, x_2\theta) \Rightarrow [\![\theta\sigma]\!] \in Cc(\beta'))$;
2. $ss = true \Rightarrow (\forall \theta \in Cc(\beta) :$ $x_1\theta$ and $x_2\theta$ are unifiable);
3. $sf = true \Rightarrow (\forall \theta \in Cc(\beta) :$ $x_1\theta$ and $x_2\theta$ are not unifiable).

Based on the upgraded operation `UNIF-VAR` for abstract substitutions, we provide an implementation of the operation `UNIF-VAR` for abstract sequences, which is correct

with respect to the corresponding specification given in Section 3.2.
The operation UNIF-VAR: $AS_{\{x_1,x_2\}} \to ASS_{\{x_1,x_2\}}$ on abstract sequences is defined as follows. Let $\beta \in AS_{\{x_1,x_2\}}$ and $\langle \beta'', ss, sf \rangle$ = UNIF-VAR($\beta$). We have that UNIF-VAR($\beta$) = $B'$ where $B'$ is the abstract sequence $\langle \beta', m', M', t' \rangle$ such that

$$\begin{aligned} \beta' &= \beta'' \\ m' &= \text{if } ss \text{ then 1 else 0} \\ M' &= \text{if } sf \text{ then 0 else 1} \\ t' &= st. \end{aligned}$$

**Abstract Interpretation of the Cut:** AI-CUT: $ASSC_{D'} \to ASSC_{D'}$
Let $C = \langle \langle \beta, m, M, t \rangle, acf \rangle$. AI-CUT($C$) = $\langle \langle \beta', m,' M', t' \rangle, acf' \rangle$ where

$$\begin{array}{lll} \beta' & = \beta & \\ m' & = \min(1, m) & \\ M' & = \min(1, M) & \\ t' & = st & \text{if } m \geq 1 \text{ or } t = st \\ & = snt & \text{if } M = 0 \text{ and } t = snt \\ & = pt & \text{otherwise} \\ acf' & = cut & \text{if } m \geq 1 \text{ or } acf = cut \\ & = nocut & \text{if } M = 0 \text{ and } acf = nocut \\ & = weakcut & \text{otherwise.} \end{array}$$

*Example.* Consider the program

```
p(X) :- q(X), !.
q(X) :- X = a.
q(X) :- X = b.
```

For the sake of simplicity we use a simple domain of abstract substitutions which can be seen as the mode component of the Pattern domain [56, 61]. The example is intended to illustrate the abstract execution of the operation AI-CUT. Hence, we do not enter here into the details of the other operations, but the reader is referred to the appendix for their definition.

The abstract execution of the procedure p called with its argument being a variable is as follows. Let

$$\beta = \mathtt{X} \mapsto \mathtt{var}$$

be the initial abstract substitution. Let $c$ be the clause of the program defining p. First, the abstract sequence with cut information $C$ is computed by

$$C = \mathtt{EXTC}(c, \beta) = \langle \langle \mathtt{X} \mapsto \mathtt{var}, 1, 1, st \rangle, nocut \rangle.$$

Then, the procedure q that occurs in the body of $c$ is executed with $\beta$ = SUBST($C$) returning the abstract sequence

$$B = \langle \mathtt{X} \mapsto \mathtt{ground}, 2, 2, st \rangle.$$

Hence, the abstract sequence with cut information $C'$ is computed as follows

$$C' = \mathtt{EXTGS}(\mathtt{q(X)}, C, B) = \langle \langle \mathtt{X} \mapsto \mathtt{ground}, 2, 2, st \rangle, nocut \rangle.$$

Now, the operation AI-CUT($C'$) is applied. Following the definition above, one obtains

$$\mathtt{AI\text{-}CUT}(C') = \langle \langle \mathtt{X} \mapsto \mathtt{ground}, 1, 1, st \rangle, cut \rangle$$

expressing the fact that a cut in the body of $c$ is *surely* executed. The final result is

$$B' = \texttt{SEQ}(C') = \langle \texttt{X} \mapsto \texttt{ground}, 1, 1, st \rangle$$

stating that the execution of `p` called with its argument being a variable surely terminates and succeeds exactly once.

Consider now the abstract execution of the procedure `p` called with a ground argument. Let

$$\beta = \texttt{X} \mapsto \texttt{ground}$$

be the initial abstract substitution. In this case, the abstract sequence with cut information $C$ is first computed by

$$C = \texttt{EXTC}(c, \beta) = \langle \langle \texttt{X} \mapsto \texttt{ground}, 1, 1, st \rangle, nocut \rangle.$$

Then, the procedure `q` is executed with $\beta = \texttt{SUBST}(C)$ returning

$$B = \langle \texttt{X} \mapsto \texttt{ground}, 0, 1, st \rangle.$$

The abstract sequence with cut information $C'$ is computed as follows

$$C' = \texttt{EXTGS}(\texttt{q(X)}, C, B) = \langle \langle \texttt{X} \mapsto \texttt{ground}, 0, 1, st \rangle, nocut \rangle.$$

The operation $\texttt{AI-CUT}(C')$ returns

$$\texttt{AI-CUT}(C') = \langle \langle \texttt{X} \mapsto \texttt{ground}, 0, 1, st \rangle, weakcut \rangle$$

expressing the fact that, in this case, the computation either fails without executing the cut or succeeds once after executing the cut. The final result is

$$B' = \texttt{SEQ}(C') = \langle \texttt{X} \mapsto \texttt{ground}, 0, 1, st \rangle$$

stating that the execution of `p` called with a ground argument succeeds at most once and surely terminates.

The `Pattern` domain used in our experiments is more elaborated than the simple domain of abstract substitutions used in this example. However, it does not provide more precision in these cases. A more sophisticated domain where an abstract sequence is represented as $\langle < \beta_1, \ldots, \beta_n >, m, M, t \rangle$ with $< \beta_1, \ldots, \beta_n >$ being an explicit sequence of abstract substitutions could return in the first case a more precise result. Indeed, one could obtain $B = \langle \{\texttt{X} \mapsto \texttt{a}\}, \{\texttt{X} \mapsto \texttt{b}\}, 2, 2, st \rangle$ and then $B' = \langle \{\texttt{X} \mapsto \texttt{a}\}, 1, 1, st \rangle$. However, such a domain could not improve the result in the second case since the fact that either $\texttt{X} \mapsto \texttt{a}$ or $\texttt{X} \mapsto \texttt{b}$ would be represented by $\texttt{X} \mapsto \texttt{ground}$ as we have done above.

**Abstract Lazy Concatenation.** The implementation of the operation `CONC` is complicated here, in order to get accurate results when the domain $AS_D$ is instantiated to the domain `Pattern`. The implementation works on *enhanced* sets of abstract sequences which allow us to keep individual structural information about the results of every clause in order to detect mutual exclusion of the clauses.

Let us motivate the lifting of abstract sequences to enhanced abstract sequences. Lifting an abstract domain to its power set (see e.g., [20, 35]) is sometimes useful when the original abstract domain is not expressive enough to gain a given level

of accuracy. Replacing an abstract domain by its power set is computationally expensive however (see [90]). Sometimes, the accuracy is lost only inside a few operations; thus, a good compromise can be to lift the domain only locally, when these operations are executed, and to go back to the simple domain afterwards. This is exactly what we are going to do for the operation `CONC`. The lifted version of the abstract domain that we are about to define is useful when the abstract domain is able to express definite, but not disjunctive, structural information about terms. In such a domain, for instance, the principal functor of the term bound to a program variable can be either definitely known or not known at all; it is not possible to express that it belongs to a given finite set. The domain `Pattern` used in our experiments is an abstract domain of this kind. Disjunctive structural information is however essential to implement the operation `CONC` accurately: it allows us to detect mutually exclusive abstract sequences, i.e., abstract sequences that should not be "abstractly concatenated" since they correspond to different concrete inputs. In order to keep disjunctive structural information, our implementation of `CONC` works on a finite set of abstract sequences. This set is "normalized" in some way, in order to simplify the case analysis in the implementation. Basically, we differentiate between "surely empty" abstract sequences, approximating only sequences of the form $<>$ or $< \bot >$, and "surely non empty" abstract sequences, approximating only sequences of the form $< \theta >:: S$. This is useful because sequences such as $<>$ or $< \bot >$ are possible outputs for any input, while sequences of the form $< \theta >:: S$ are only possible for some inputs. Therefore we only have to check incompatibility of "surely non empty" abstract sequences. This discussion motivates the following definitions of semi-simple abstract sequences and simple abstract sequences.

*Definition 5.8.* [Semi-Simple Abstract Sequences]
Let $B \in ASS_D$. We say that $B$ is a *semi-simple* abstract sequence if

1. either, $\beta = \beta_\emptyset$ and $m = M = 0$
2. or, $\beta \neq \beta_\emptyset$ and $1 \leq m \leq M$.

*Definition 5.9.* [Simple Abstract Sequences]
Let $B \in ASS_D$. We say that $B$ is a *simple* abstract sequence if it is semi-simple and $t \in \{snt, st\}$.

Semi-simple abstract sequences formalize our idea of distinguishing between "surely empty" and "surely non empty" abstract sequences. Note that, assuming that $\beta_\emptyset$ is the only abstract substitution such that $Cc(\beta_\emptyset) = \emptyset$, we have that $Cc(B) \neq \emptyset$ for any semi-simple abstract sequence $B$.

*Definition 5.10.* [Enhanced Abstract Sequences]
Let $D$ be a finite set of program variables. We denote by $ASS_D^{enh}$ the set of all sets of the form $\{B_1, \ldots, B_n\}$, where $n \geq 0$ and $B_1, \ldots, B_n$ are semi-simple abstract sequences from $ASS_D$. Elements of $ASS_D^{enh}$ are called *enhanced abstract sequences*; they are denoted by $SB$ in the following. The concretization function $Cc : ASS_D^{enh} \rightarrow CSS_D$ is defined by $Cc(SB) = \bigcup_{B \in SB} Cc(B)$.

The operation `SPLIT1` transforms an arbitrary abstract sequence into an equivalent enhanced abstract sequence.

*Operation* $\texttt{SPLIT1} : ASS_D \rightarrow ASS_D^{enh}$
This operation is required to satisfy the property that for every $B \in ASS_D$, $Cc(\texttt{SPLIT1}(B)) = Cc(B)$. Let $B = \langle \beta, m, M, t \rangle$. We define $SB' = \texttt{SPLIT1}(B)$ as $SB' = SB_1 \cup SB_2$ where

$$\begin{array}{lll} SB_1 = \{\langle \beta_\emptyset, 0, 0, t \rangle\} & & \text{if } m = 0 \\ \quad = \emptyset & & \text{otherwise} \\ SB_2 = \{\langle \beta, \max(1,m), M, t \rangle\} & & \text{if } \beta \neq \beta_\emptyset \text{ and } \max(1,m) \leq M \\ \quad = \emptyset & & \text{otherwise.} \end{array}$$

The operation MERGE is the converse of SPLIT1: it transforms an enhanced abstract sequence into a plain abstract sequence. Most of the time, this operation loses part of the information expressed by the enhanced abstract substitution sequence; but it does not lose any information when the enhanced abstract sequence results from a single application of SPLIT1.

*Operation* $\texttt{MERGE} : ASS_D^{enh} \rightarrow ASS_D$
The operation MERGE satisfies the following properties:

1. For every $SB \in ASS_D^{enh}$, $Cc(SB) \subseteq Cc(\texttt{MERGE}(SB))$
2. For every $B \in ASS_D$, $Cc(\texttt{MERGE}(\texttt{SPLIT1}(B))) = Cc(B)$.

The definition of MERGE requires choosing a particular abstract sequence $B_\emptyset$ such that $Cc(B_\emptyset) = \emptyset$. We decide that $B_\emptyset = \langle \beta_\emptyset, 1, 0, st \rangle$. This choice is arbitrary since there is no best (least) representation of the empty set of abstract sequences in this domain. Moreover, it uses the binary operation $\texttt{UNION} : (AS_D \times AS_D) \rightarrow AS_D$, which is inherited from our previous framework. The latter is extended to finite sequences of abstract substitutions as follows:

$$\begin{array}{lcl} \texttt{UNION}(<>) & = & \beta_\emptyset \\ \texttt{UNION}(<\beta>) & = & \beta, \text{ for every } \beta \in AS_D \\ \texttt{UNION}(<\beta_1, \ldots, \beta_n>) & = & \texttt{UNION}(\beta_1, \texttt{UNION}(<\beta_2, \ldots, \beta_n>)), \\ & & \text{for all } \beta_1, \ldots, \beta_n \in AS_D \ (n \geq 2). \end{array}$$

The operation MERGE can now be defined. Let $\sqcup$ denote the least upper bound on *TI*. Let $SB \in ASS_D^{enh}$ such that $SB = \{B_1, \ldots, B_n\}$ and $B_i = \langle \beta_i, m_i, M_i, t_i \rangle$ $(1 \leq i \leq n)$. The abstract sequence $B' = \texttt{MERGE}(SB)$ is such that

$$\begin{array}{lll} B' = B_\emptyset & & \text{if } n = 0 \\ \quad = B_1 & & \text{if } n = 1 \\ \quad = \langle \texttt{UNION}(<\beta_1, \ldots, \beta_n>), \min(m_1, \ldots, m_n), & & \\ \qquad \max(M_1, \ldots, M_n), t_1 \sqcup \ldots \sqcup t_n \rangle & & \text{if } n \geq 2. \end{array}$$

The notion of simple abstract sequence with cut information is also useful to simplify the case analysis in the implementation of CONC.

*Definition 5.11.* [Simple Abstract Sequences with Cut Information]
Let $B \in ASS_D$ and $acf \in ACF$. The abstract sequence with cut information $\langle B, acf \rangle$ is said to be *simple* if $B$ is simple and $acf \in CF$.

The operation SPLIT2 converts an arbitrary abstract sequence with cut information into an equivalent set of simple abstract sequences with cut information.

*Operation* $\texttt{SPLIT2} : ASSC_D \to \wp(ASSC_D)$
The operation SPLIT2 satisfies the following properties. For every $C \in ASSC_D$,

1. $\bigcup_{C' \in \texttt{SPLIT2}(C)} Cc(C') = Cc(C)$;
2. all abstract sequences with cut information in $\texttt{SPLIT2}(C)$ are simple.

Its definition is simple. We first apply the operation SPLIT1 to the abstract sequence part of $C$. Then we split the cut information. Finally we split the termination information. Formally, $\texttt{SPLIT2}(C)$ is defined as follows.

1. Let $C = \langle B, acf \rangle \in ASSC_D$. We define

   $$\texttt{SPLIT2}(C) = \bigcup_{B' \in \texttt{SPLIT1}(B)} \texttt{SPLIT2}(\langle B', acf \rangle).$$

2. Let $B = \langle \beta, m, M, t \rangle \in ASS_D$. Assume that $B$ is semi-simple. We define

   $$\begin{array}{lll} \texttt{SPLIT2}(\langle B, weakcut \rangle) & = \texttt{SPLIT2}(\langle B, nocut \rangle) \cup \texttt{SPLIT2}(\langle B, cut \rangle) & \text{if } m = 0 \\ & = \texttt{SPLIT2}(\langle B, cut \rangle) & \text{if } m \geq 1. \end{array}$$

   (Remember that, by Definition 5.8, we also have $\beta = \beta_\emptyset$ and $M = 0$, in the first case, and $\beta \neq \beta_\emptyset$ and $m \leq M$, in the second case.)

3. Let $B = \langle \beta, m, M, t \rangle \in ASS_D$ and $cf \in CF$. Assume that $B$ is semi-simple. We define

   $$\begin{array}{lll} \texttt{SPLIT2}(\langle B, cf \rangle) & = \{\langle B, cf \rangle\} & \text{if } t \in \{snt, st\}; \\ & = \{\langle \langle \beta, m, M, snt \rangle, cf \rangle, \langle \langle \beta, m, M, st \rangle, cf \rangle\} & \text{if } t = pt. \end{array}$$

Before presenting the implementation of CONC, we still need to specify the operation EXCLUSIVE, which is aimed at detecting incompatible outputs. An implementation of this operation for the domain Pattern is given in Section 5.3.

*Operation* $\texttt{EXCLUSIVE} : (AS_D \times AS_D \times AS_D) \to Bool$
The operation EXCLUSIVE satisfies the following property. For all $\beta, \beta_1, \beta_2 \in AS_D$,

$$\begin{array}{l} \texttt{EXCLUSIVE}(\beta, \beta_1, \beta_2) \Rightarrow \neg(\exists \theta \in Cc(\beta), \theta_1 \in Cc(\beta_1), \theta_2 \in Cc(\beta_2), \sigma_1, \sigma_2 \in SS : \\ \qquad \theta\sigma_1 = \theta_1 \text{ and } \theta\sigma_2 = \theta_2). \end{array}$$

We are now ready to describe the operation CONC.

*Operation* $\texttt{CONC} : (AS_D \times ASSC_D \times ASS_D^{enh}) \to ASS_D^{enh}$.
Let $\beta \in AS_D$, $C_1 \in ASSC_D$ and $SB_2 \in ASS_D^{enh}$. $SB' = \texttt{CONC}(\beta, C_1, SB_2)$ is defined as follows. We assume that $B_i = \langle \beta_i, m_i, M_i, t_i \rangle$.

1. Let us assume first that $C_1 = \langle B_1, acf_1 \rangle$ is simple and $SB_2 = \{B_2\}$.

   (a) Suppose that $acf_1 = cut$ or $t_1 = snt$. In this case, we define $SB' = \{B_1\}$.

   (b) Suppose, on the contrary, that $acf_1 = nocut$ and $t_1 = st$. We define

$$
\begin{array}{lll}
SB' = \{B_2\} & & \text{if } M_1 = 0 \\
= \{\langle \beta_1, m_1, M_1, t_2 \rangle\} & & \text{if } M_1 \geq 1 \text{ and } M_2 = 0 \\
= \{\langle \texttt{UNION}(\beta_1, \beta_2), m_1 + m_2, M_1 + M_2, t_2 \rangle\} & & \text{if } M_1 \geq 1 \text{ and } M_2 \geq 1 \\
 & & \text{and } \neg\texttt{EXCLUSIVE}(\beta, \beta_1, \beta_2) \\
= \emptyset & & \text{if } M_1 \geq 1 \text{ and } M_2 \geq 1 \\
 & & \text{and } \texttt{EXCLUSIVE}(\beta, \beta_1, \beta_2).
\end{array}
$$

2. In the general case, we define

$$SB' = \bigcup_{\substack{C \in \texttt{SPLIT2}(C_1) \\ B \in SB_2}} \texttt{CONC}(\beta, C, \{B\}).$$

### *5.3. Instantiation to* `Pattern`

The domain of abstract substitutions `Pattern` has been introduced in [61] and it has been used in many of our previous works, e.g.,[32, 62]. The reader is referred to [61] for a detailed description of the domain and of its abstract operations.

**The Abstract Domain Pattern.** The version of `Pattern` used in the experimental evaluation of Section 5.4 can be best viewed as an instantiation of the generic pattern domain `Pat(`$\mathcal{R}$`)` [17] with mode, sharing, and arithmetic components.
The key intuition behind `Pat(`$\mathcal{R}$`)` is to represent information on some subterms occurring in a substitution instead of information on terms bound to variables only. More precisely, `Pat(`$\mathcal{R}$`)` may associate the following information with each considered subterm: (1) its *pattern*, which specifies the main functor of the subterm (if any) and the subterms which are its arguments; its *properties*, which are left unspecified and are given in the domain $\mathcal{R}$. In addition to the above information, each variable in the domain of the substitution is associated with one of the subterms. It can be expressed that two arguments have the same value (and hence that two variables are bound together) by associating both arguments with the same subterm. It should be emphasized that the pattern information may be void. In theory, information on all subterms could be kept but the requirement for a finite analysis makes this impossible for almost all applications. As a consequence, the domain shares some features with the depth-k abstraction [48], although `Pat(`$\mathcal{R}$`)` does not impose a fixed depth but adjusts it dynamically through upper bound and widening operations. Note that the identification of subterms (and hence the link between the structural components and the $\mathcal{R}$-domain) is a somewhat arbitrary choice. In `Pat(`$\mathcal{R}$`)`, subterms are identified by integer indices, say $1, \ldots, n$ if $n$ subterms are considered, and we denote sets of indices by the symbol $I$.
More formally, the pattern and same-value component can be described as follows. The *pattern* component is a partial function $frm : I \not\to Pat_I$, from the set of indices $I$ to the set of patterns over $I$, i.e., elements of the form $f(i_1, \ldots, i_n)$, where $f \in \mathcal{F}$ is a functor symbol of arity $n$ and $i_1, \ldots, i_n \in I$. When the pattern is undefined for an index $i$, we write *frm(i) = undef.* The *same-value* component is a total function $sv : D \to I$, where $D = \{x_1, \ldots, x_n\}$ is the domain of the abstract substitution.
A pattern component $frm : I \not\to Pat_I$ denotes a set of families $(t_i)_{i \in I}$ of terms as defined below.

$$Cc(frm) = \{(t_i)_{i \in I} \mid frm(i) = f(i_1, \ldots, i_n) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_n}), \\ \forall i, i_1, \ldots, i_n \in I, \forall f \in \mathcal{F}\}.$$

In order to simulate unification with occur-check, we also assume that every pattern component *frm* satisfies the following condition: the relation $\succ \subseteq I \times I$ such that $i \succ j$ if and only if $frm(i)$ is of the form $f(\ldots, j, \ldots)$ must be well-founded.
A pair $\langle sv, frm \rangle$ with $sv : D \to I$ and $frm : I \not\to Pat_I$ is called *structural abstract substitution*; it denotes a set of program substitutions as follows:

$$Cc(\langle sv, frm \rangle) = \{\theta \in PS_D \mid \exists (t_i)_{i \in I} \in Cc(frm) : \; x_j\theta = t_{sv(x_j)}, \; \forall x_j \in D\}.$$

The $\mathcal{R}$-domain is the generic part which specifies subterm information by describing properties of a set of tuples $< t_1, \ldots, t_n >$ where $t_1, \ldots, t_n$ are terms. As a consequence, defining the $\mathcal{R}$-domain amounts essentially to defining a traditional domain on substitutions and its operations. We now describe the various components of the $\mathcal{R}$-domain which can be built as an open product [17].
The *mode* component is described in [61] and associates a mode from the set $Modes = \{\texttt{var}, \texttt{ground}, \texttt{novar}, \texttt{noground}, \texttt{ngv}, \texttt{gv}, \texttt{any}\}$ with each subterm. Formally, it is a total function $mo : I \to Modes$ whose concretization is defined as

$$Cc(mo) = \{(t_i)_{i \in I} \mid \; t_i \in Cc(mo(i)), \; \forall i \in I\}.$$

The *sharing* component maintains information about possible sharing between pairs of subterms and is also described in [61]. Formally, it is a symmetrical relation $ps \subseteq I \times I$ whose concretization is defined as

$$Cc(ps) = \{(t_i)_{i \in I} \mid \; var(t_i) \cap var(t_j) \Rightarrow ps(i,j), \; \forall i, j \in I\}.$$

The *arithmetic* component is novel and aims at using arithmetic predicates to detect mutual exclusion between clauses. It approximates information about arithmetic relationships by rational order constraints, i.e., binary constraints of the form $i \; \delta \; j$ and unary constraints of the form $i \; \delta \; c$, where $i, j$ are indices, $\delta \in \{>, \geq, =, \neq, \leq, <\}$ and $c$ is an integer constant. For instance, a built-in $X \geq Y + 2$ is approximated by a constraint $X > Y$. Formally, an element *arithm* is a set of rational order constraints over indices, whose concretization is defined as follows (a constraint being satisfied only if the terms are numbers).

$$Cc(arithm) = \{(t_i)_{i \in I} \mid \; \forall \; i \; \delta \; j \in arithm : \; t_i \; \delta \; t_j \text{ and } \forall \; i \; \delta \; c \in arithm : \; t_i \; \delta \; c\}.$$

**The Operation EXCLUSIVE.** We describe here the implementation of the operation `EXCLUSIVE` on our domain of abstract substitutions. This operation was not present in our previous works. It aims at detecting situations where two output abstract sequences $B_1$ and $B_2$ are incompatible, given that they both originate from the same abstract input substitution $\beta$. Only the abstract substitution components $\beta_1$ and $\beta_2$ of $B_1$ and $B_2$ are useful to detect such situations. Thus the operation `EXCLUSIVE` has three arguments $\beta$, $\beta_1$, and $\beta_2$. (See its specification in Section 5.2.)
Let us first introduce the notion of decomposition of a program substitution with respect to a structural abstract substitution. It represents the family of terms, occurring in the program substitution, that are given an index by the structural abstract substitution.

*Definition 5.12.* [Decomposition of a Program Substitution]
Let $\langle sv, frm \rangle$ be a structural abstract substitution over domain $D = \{x_1, \ldots, x_n\}$ and set of indices $I$. Let also $\theta \in Cc\langle sv, frm \rangle$. The *decomposition of $\theta$ with respect to $\langle sv, frm \rangle$* is the (unique) family of terms $(t_i)_{i \in I}$ such that

$$\theta = \{x_1/t_{sv(x_1)}, \ldots, x_n/t_{sv(x_n)}\} \text{ and } (t_i)_{i \in I} \in Cc(frm).$$

Existence and unicity of the family $(t_i)_{i \in I}$ can be proven by an induction argument that uses the fact that the relation $\succ$ over $I$ is well-founded. Unicity holds conditional to the fact that $I$ does not contain any "useless" element, i.e., for every $i \in I$, there exists a variable $x_j \in D$ and a set of indices $i_1, \ldots, i_k$ such that $i_1 = sv(x_j)$, $i_1 \succ \ldots \succ i_k$, and $i_k = i$. From now on we assume that this condition always holds.

The next definition models a property of the structural abstract substitutions obtained by performing any number of abstract unification steps on another structural abstract substitution.

*Definition 5.13.* [Instance of a Structural Abstract Substitution]
Let $\langle sv, frm \rangle$ and $\langle sv', frm' \rangle$ be two structural abstract substitutions over the same domain $D = \{x_1, \ldots, x_n\}$ and respective sets of indices $I$ and $I'$. Let also $im : I \to I'$ be a total function. We say that $\langle sv', frm' \rangle$ *is an instance of* $\langle sv, frm \rangle$ *with respect to* $im$ if the following conditions hold:

1. $sv' = im \circ sv$;
2. for all $i, i_1, \ldots, i_m \in I$,
   $frm(i) = f(i_1, \ldots, i_m) \Rightarrow frm'(im(i)) = f(im(i_1), \ldots, im(i_m))$.

Moreover, we say that $\langle sv', frm' \rangle$ *is an instance of* $\langle sv, frm \rangle$ if there exists a function $im$ such that the conditions hold.

The next property holds.

*Property 5.1.* Let $\langle sv, frm \rangle$ and $\langle sv', frm' \rangle$ be two structural abstract substitutions, and let $im : I \to I'$ be such that $\langle sv', frm' \rangle$ is an instance of $\langle sv, frm \rangle$ with respect to $im$. Let also $\theta \in Cc\langle sv, frm \rangle$, $\theta' \in Cc\langle sv', frm' \rangle$, and $\sigma \in SS$. Finally, let $(t_i)_{i \in I}$ and $(t'_i)_{i \in I'}$ be the decompositions of $\theta$ and $\theta'$ with respect to $\langle sv, frm \rangle$ and $\langle sv', frm' \rangle$, respectively. Then we have

$$\theta' = \theta\sigma \Rightarrow (t_i\sigma)_{i \in I} = (t'_{im(i)})_{i \in I}.$$

The proof is a simple induction on the well-founded relation $\succ$, induced on $I$ by $frm$.

The next definitions and properties are instrumental to the implementation and correctness proof of the operation `EXCLUSIVE`.

*Definition 5.14.* [Exclusive Pair of Indices]
Let $frm_1$ and $frm_2$ be two pattern components over sets of indices $I$ and $J$, respectively. Let also $i \in I$ and $j \in J$.

1. We say that $\langle i, j \rangle$ is *directly exclusive* with respect to $\langle frm_1, frm_2 \rangle$ iff $frm_1(i) = f(i_1, \ldots, i_p)$, $frm_2(j) = g(j_1, \ldots, j_q)$ and either $f \neq g$ or $p \neq q$.
2. We say that $\langle i, j \rangle$ is *exclusive* with respect to $\langle frm_1, frm_2 \rangle$ iff $\langle i, j \rangle$ is *directly exclusive* with respect to $\langle frm_1, frm_2 \rangle$, or $frm_1(i) = f(i_1, \ldots, i_p)$, $frm_2(j) = f(j_1, \ldots, j_p)$ and there exists $k : 1 \leq k \leq p$ such that $\langle i_k, j_k \rangle$ is exclusive with respect to $\langle frm_1, frm_2 \rangle$.

*Property 5.2.* Let $frm_1$ and $frm_2$ be two pattern components over sets of indices $I$ and $J$, respectively. Let $(t_i)_{i\in I} \in Cc(frm_1)$ and $(t_j)_{j\in J} \in Cc(frm_2)$. Let also $i \in I$ and $j \in J$.

1. If the pair $\langle i, j\rangle$ is *directly exclusive* with respect to $\langle frm_1, frm_2\rangle$, then the terms $t_i$ and $t_j$ are compound and they have distinct principal functors.
2. If the pair $\langle i, j\rangle$ is *exclusive* with respect to $\langle frm_1, frm_2\rangle$, then the terms $t_i$ and $t_j$ are distinct ($t_i \neq t_j$).

We are now in position to provide the implementation of the operation `EXCLUSIVE` for the domain `Pattern`. We just show here a partial implementation which only uses the pattern, same-value, and mode components but it gives the idea behind the complete implementation. For additional details, the reader is referred to [7].

*Operation* `EXCLUSIVE` : `Pattern` $\times$ `Pattern` $\times$ `Pattern` $\rightarrow$ *Bool*
Let $\beta, \beta_1, \beta_2$ be abstract substitutions over the same domain $D$ and sets of indices $I$, $I_1$, and $I_2$, respectively. Assume that $\langle sv_1, frm_1\rangle$ and $\langle sv_2, frm_2\rangle$ are instances of $\langle sv, frm\rangle$ with respect to $im_1$ and $im_2$, respectively. The value of `EXCLUSIVE`$(\beta, \beta_1, \beta_2)$ is *true* if and only if there exists $i \in I$ such that

1. $mo(i) \in \{$`ngv`, `novar`$\}$ and the pair $\langle im_1(i), im_2(i)\rangle$ is directly exclusive with respect to $\langle frm_1, frm_2\rangle$, or
2. $mo(i) =$ `ground` and the pair $\langle im_1(i), im_2(i)\rangle$ is exclusive with respect to $\langle frm_1, frm_2\rangle$.

Correctness of the implementation follows from Properties 5.1 and 5.2 (see [59]).

**Prolog's Built-in Predicates.** Prolog's built-in predicates such as test predicates (`var`, `ground`, and the like) or arithmetic predicates (`is`, `<`, ...) can be handled in essentially the same way as abstract unification. Our implementation actually includes abstract operations that deal with test and arithmetic predicates (see [7]). Other built-in predicates can be accommodated as well, including the predicates `assert` and `retract`. However, the treatment of the latter predicates assumes that dynamic predicates are disjoint from static predicates, i.e., it assumes that the underlying program $P$ is not modified. A more satisfactory treatment of dynamic predicates requires to introduce a new abstract object representing the dynamic program; this improvement is a topic for further work.

## *5.4. Experimental Evaluation*

The experimental results presented in this section provide evidence of the fact that the approach presented in this paper allows one to integrate predicate level analysis to existing variable level analysis at a reasonable implementation cost. Comparisons with other cardinality and determinacy analyses can be found in Section 6.

**Benchmarks.** Our experiments use our traditional benchmarks (available by anonymous ftp from ftp://ftp.info.fundp.ac.be/pub/users/ble/bench.p) except that cuts have been reinserted as in the original versions. In addition, some new programs have been added. `Boyer` is a theorem-prover from the DEC-10 benchmarks,

| | OR | | PC | | | | PCA | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Programs | I | T | I | T | IR | TR | I | T | IR | TR |
| Qsort | 13 | 0.08 | 17 | 0.12 | 1.31 | 1.50 | 13 | 0.08 | 1.00 | 1.00 |
| Qsort2 | 15 | 0.08 | 19 | 0.12 | 1.27 | 1.50 | 15 | 0.09 | 1.00 | 1.13 |
| Queens | 15 | 0.07 | 18 | 0.08 | 1.20 | 1.14 | 18 | 0.10 | 1.20 | 1.43 |
| Press1 | 532 | 11.77 | 581 | 13.11 | 1.09 | 1.11 | 581 | 13.45 | 1.09 | 1.14 |
| Press2 | 197 | 3.27 | 200 | 3.56 | 1.02 | 1.09 | 200 | 3.56 | 1.02 | 1.09 |
| Gabriel | 78 | 0.90 | 84 | 1.00 | 1.08 | 1.11 | 84 | 0.98 | 1.08 | 1.09 |
| Peep | 132 | 3.21 | 131 | 18.85 | 0.99 | 5.87 | 131 | 19.08 | 0.99 | 5.94 |
| Read | 432 | 23.91 | 458 | 25.32 | 1.06 | 1.06 | 458 | 25.37 | 1.06 | 1.06 |
| Kalah | 115 | 1.90 | 121 | 2.09 | 1.05 | 1.10 | 120 | 2.11 | 1.04 | 1.11 |
| Cs | 79 | 2.19 | 91 | 3.05 | 1.15 | 1.39 | 90 | 3.02 | 1.14 | 1.38 |
| Plan | 36 | 0.21 | 38 | 0.30 | 1.06 | 1.43 | 38 | 0.27 | 1.06 | 1.29 |
| Disj | 64 | 1.95 | 68 | 2.14 | 1.06 | 1.10 | 68 | 2.12 | 1.06 | 1.09 |
| Pg | 38 | 0.32 | 40 | 0.36 | 1.05 | 1.13 | 39 | 0.35 | 1.03 | 1.09 |
| Boyer | 56 | 0.76 | 56 | 1.15 | 1.00 | 1.51 | 56 | 1.17 | 1.00 | 1.54 |
| Credit | 63 | 0.57 | 64 | 0.81 | 1.02 | 1.42 | 64 | 0.80 | 1.02 | 1.40 |
| Mean | | | | | 1.09 | 1.56 | | | 1.05 | 1.52 |

**TABLE 5.1.** Efficiency of the Cardinality Analysis

`Credit` is an expert system from [85]. There are two versions of `Qsort` which differ in procedure `Partition` which uses or does not use auxiliary predicates for the arithmetic built-ins. The benchmarks have been run on a SUN SS-10/20.

**Efficiency.** The efficiency results are reported in Table 5.1. Several algorithms are compared: `OR` is the original `GAIA` algorithm on `Pattern` [61], `PC` is the cardinality analysis with `Pattern` and `PCA` is `PC` with the abstraction for arithmetic predicates. I, T, IR and TR are the number of iterations, the execution time (in seconds), the iteration's ratio and the time's ratio respectively. The first interesting point to notice is the slight increase (about 5% on `PCA`) in iterations when moving from abstract substitutions to abstract sequences, showing the effectiveness of our widening operator. Even more important perhaps is the fact that the time overhead of the cardinality analysis is small with respect to the traditional analysis: `PCA` is 1.52 slower than `OR`. Note that in fact most programs enjoys an even smaller overhead but `Peep` is about 6 times slower than `OR` in `PCA`. This comes from many procedures with many clauses, most of which being not surely cut; much time is spent in the concatenation operation. Finally, note that adding more functionality in the domain did not slow down the analysis by much.

**Accuracy.** The accuracy results are reported in Table 5.2. For each program we specify the initial query to which the abstract interpretation algorithm is applied (we denote by `a`, `g` and `v` the modes `any`, `ground` and `var`, respectively). Several versions of the algorithm are compared with respect to their ability to detect determinacy of procedures, which was our primary motivation. `P` is using only the domain `Pattern` (i.e., cuts are ignored), `C` is only using the cut (i.e., `EXCLUSIVE` always returns false), and `PC`, `PCA` are defined as previously. In the table, NP stands for the number of procedures and D and %D denote, respectively, the number of deterministic procedures and the percentage of deterministic procedures detected by the algorithms. There are several interesting points to notice. First, `PCA` detects

| Programs | Query | NP | P | | C | | PC | | PCA | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | D | %D | D | %D | D | %D | D | %D |
| Qsort | `qsort(g,v)` | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 100 |
| Qsort2 | `qsort(g,v)` | 5 | 2 | 40 | 2 | 40 | 2 | 40 | 5 | 100 |
| Queens | `queens(g,v)` | 5 | 2 | 40 | 0 | 0 | 2 | 40 | 2 | 40 |
| Press1 | `test_press(v,v)` | 47 | 8 | 17 | 19 | 40 | 19 | 40 | 19 | 40 |
| Press2 | `test_press(v,v)` | 47 | 12 | 26 | 19 | 40 | 28 | 60 | 28 | 60 |
| Gabriel | `main(v,v)` | 17 | 0 | 0 | 4 | 24 | 4 | 24 | 4 | 24 |
| Peep | `comppeeppopt(g,v,g)` | 24 | 4 | 17 | 7 | 29 | 16 | 67 | 16 | 67 |
| Read | `read(v,v)` | 46 | 11 | 24 | 27 | 59 | 31 | 67 | 31 | 67 |
| Kalah | `play(v,v)` | 46 | 16 | 35 | 20 | 43 | 33 | 72 | 40 | 87 |
| Cs | `pgenconfig(v)` | 32 | 11 | 34 | 7 | 22 | 11 | 34 | 13 | 41 |
| Plan | `transform(g,g,v)` | 13 | 1 | 8 | 0 | 0 | 1 | 8 | 1 | 8 |
| Disj | `top(v)` | 28 | 13 | 46 | 11 | 39 | 13 | 46 | 13 | 46 |
| Pg | `pdsbm(g,v)` | 10 | 2 | 20 | 3 | 30 | 5 | 50 | 6 | 50 |
| Boyer | `boyer(g)` | 24 | 0 | 0 | 20 | 83 | 20 | 83 | 20 | 83 |
| Credit | `credit(a,a)` | 26 | 14 | 58 | 11 | 42 | 14 | 54 | 16 | 62 |
| Mean | | | | 24 | | 33 | | 46 | | 58 |

**TABLE 5.2.** Accuracy of the Cardinality Analysis

that 58% of the procedures are deterministic, although many of these programs in fact use heavily the nondeterminism of Prolog. Most of the results are optimal and a nice example is the program `Kalah`. Second, the cut and input/output patterns are really complementary to improve the analysis. Input/output patterns alone give 42% of the deterministic procedures, while the cut detects 56% of the deterministic procedures. The abstraction of arithmetic predicates adds 22% of deterministic procedures. The main lesson here is that all components are of primary importance to obtain precise results.

## 6. RELATED WORKS ON DETERMINACY ANALYSIS

Determinacy of logic programs in general and of Prolog programs in particular is an important research topic because determinate programs can be implemented more efficiently than non-determinate programs (often, much more efficiently). Several forms of determinacy have been identified, which lead to different kinds of optimizations. In this section, we review a few interesting papers on determinacy analysis at the light of our novel framework for the abstract interpretation of Prolog. The benefit of this study is twofold: first, it sheds new light on these analyses in the context of abstract interpretation; second, it supports the claim that our proposal is appropriate to integrate most existing analyses into a single framework.

### *6.1. Sahlin's Determinacy Analysis for Full Prolog*

The analysis proposed by D. Sahlin in [81] aims at detecting procedures of a (full) Prolog program that are determinate (i.e., they succeed at most once) or fully-determinate (i.e., they succeed exactly once). The analysis is developed in the context of the partial evaluator Mixtus [80] in order to detect situations where cuts can be "executed" or removed. Sahlin's analysis is not based on abstract

interpretation; hence he provides a specific correctness proof for it.
In this section, we show that the determinacy analysis proposed by Sahlin in [81] is indeed an instance of our framework over his abstract domain.

**Abstract Domains.** Sahlin's analysis completely ignores information on program variables. The abstract domains are concerned with the sequence structure only: substitutions are completely ignored. Note that no abstract interpretation framework available at the time of his writing was adequate to his needs.

**Abstract Substitutions.** Since program variables are ignored, we can assume a domain $AS$ consisting of an arbitrary single element.

**Abstract Sequences.** Sahlin's analysis can be formalized in our framework by defining $ASS = \wp(AASS)$, where $AASS = \{\mathcal{L}, 0, 1, 1', 2, 2'\}$[2]. We call elements of $AASS$, *atomic abstract sequences*. Their concretization is defined as follows:

$$
\begin{aligned}
Cc(\mathcal{L}) &= \{< \bot >\} \\
Cc(0) &= \{ <> \} \\
Cc(1) &= \{S \in PSS \mid Ns(S) = 1 \text{ and } S \text{ is finite}\} \\
Cc(1') &= \{S \in PSS \mid Ns(S) = 1 \text{ and } S \text{ is incomplete}\} \\
Cc(2) &= \{S \in PSS \mid Ns(S) > 1 \text{ and } S \text{ is finite}\} \\
Cc(2') &= \{S \in PSS \mid Ns(S) > 1 \text{ and } S \text{ is incomplete or infinite}\}
\end{aligned}
$$

The concretization function $Cc : ASS \to \wp(PSS)$ is defined by:

$$Cc(B) = \bigcup_{b \in B} Cc(b).$$

The relation $\leq$ on $ASS$ is naturally defined as being set inclusion. The concretization function is thus clearly monotonic.

**Abstract Sequences with Cut Information.** We define the set $ASSC$ as being equal to $\wp(AASS \times CF)$. The elements of $ASSC$ are denoted by $\mathcal{L}_n$, $0_n$, $1_n$, $1'_n$, $2_n$, $2'_n$, $\mathcal{L}_c$, $0_c$, $1_c$, $1'_c$, $2_c$, $2'_c$, in [81], where the index $n$ stands for *nocut*, while the index $c$ stands for *cut*. The concretization function is defined in the obvious way.

**Extended Widening.** In order to instantiate our generic abstract interpretation algorithm to the above domains, it remains to provide an implementation of the various abstract operations. This can be done systematically from the specifications of the operations and the domain definitions; we leave it as an exercise to the reader, except for the extended widening, whose implementation is not obvious. The basic intuition behind the extended widening is that it should "observe" how the abstract sequences evolve between the consecutive iterations in order to ensure convergence when enough accuracy seems to be attained. In this abstract domain, the abstract sequence $B_i$ produced at step $i$ may intuitively differ from $B_{i-1}$ by the fact that some "incomplete" elements (i.e., $\mathcal{L}$, $1'$, $2'$) can be removed and replaced by more "complete" ones. Of course the computation starts with $B_0 = \{\mathcal{L}\}$. Thus the algorithm waits until "enough incomplete elements have been removed" and then accumulates the next iteration results to enforce termination. This can be formalized by defining a pre-order $\sqsubseteq$ over $ASS$ such that $B_1 \sqsubseteq B_2$ holds when $B_2$

[2] We choose to denote the elements of $AASS$ by the same symbols as in [81].

only contains elements that are "more complete" than some elements of $B_1$ and when, conversely, $B_1$ only contains elements that are "less complete" than some elements of $B_2$. We first define the relation *is strictly less complete than* between atomic abstract sequences by the table:

$$\mathcal{L} \sqsubset 0 \;\; \mathcal{L} \sqsubset 1 \;\; \mathcal{L} \sqsubset 1' \;\; \mathcal{L} \sqsubset 2 \;\; \mathcal{L} \sqsubset 2' \;\; 1' \sqsubset 1 \;\; 1' \sqsubset 2 \;\; 1' \sqsubset 2' \;\; 2' \sqsubset 2.$$

Then, for all atomic abstract sequences $b_1$ and $b_2$, we say that $b_1$ *is less complete than* $b_2$, denoted by $b_1 \sqsubseteq b_2$, if $b_1 = b_2$ or $b_1 \sqsubset b_2$. This relation is lifted to general abstract sequences as follows:

*Definition 6.1.* [Computational Pre-Ordering]
Let $B_1, B_2 \in ASS$. By definition,

$$\begin{aligned} B_1 \sqsubseteq B_2 \;\text{ iff }\; & (\forall b_1 \in B_1,\, \exists b_2 \in B_2 \text{ such that } b_1 \sqsubseteq b_2) \text{ and} \\ & (\forall b_2 \in B_2,\, \exists b_1 \in B_1 \text{ such that } b_1 \sqsubseteq b_2). \end{aligned}$$

We write $B_1 \sqsubset B_2$ to denote the condition ($B_1 \sqsubseteq B_2$ and $B_2 \not\sqsubseteq B_1$).

We are now in position to define the extended widening.

*Definition 6.2.* [Extended Widening for Sahlin's Domain: $B' = B_{new} \nabla B_{old}$]

$$\begin{aligned} B' &= B_{new} && \text{if } B_{old} \sqsubset B_{new}, \\ &= B_{new} \cup B_{old} && \text{otherwise.} \end{aligned}$$

In fact, the above operation does not fulfill, strictly speaking, the requirements for being an extended widening. It works however if we have $B_{old} \sqsubseteq B_{new}$ each time it is applied. This is normally the case if the other abstract operations are carefully implemented, since each iteration of the abstract interpretation algorithm should replace every element in $B_{old}$ by one or several more complete elements. Before stating what is it actually achieved by the operation $\nabla$, we need two definitions.

*Definition 6.3.* [Equivalent Abstract Sequences]
Let $B_1, B_2 \in ASS$. By definition,

$$B_1 \approx B_2 \;\text{ iff }\; B_1 \sqsubseteq B_2 \text{ and } B_2 \sqsubseteq B_1.$$

The relation $\approx$ is an equivalence because $\sqsubseteq$ is a pre-order. It can be shown that $\approx$ determines 42 equivalence classes, of which 28 are a singleton (e.g., $\{\{\mathcal{L}, 0, 1'\}\}$), 10 have 2 elements (e.g., $\{\{\mathcal{L}, 0, 2'\}, \{\mathcal{L}, 0, 1', 2'\}\}$), and 4 have 4 elements (e.g., $\{\{\mathcal{L}, 0, 2\}, \{\mathcal{L}, 0, 2, 2'\}, \{\mathcal{L}, 0, 1', 2\}, \{\mathcal{L}, 0, 1', 2, 2'\}\}$). It is also important to note that distinct equivalent abstract sequences always have different concretizations.

*Definition 6.4.* [Strengthened Computational Ordering]
Let $B_1, B_2 \in ASS$. By definition,

$$B_1 \trianglelefteq B_2 \;\text{ iff }\; B_1 \sqsubset B_2 \text{ or } (B_1 \approx B_2 \text{ and } B_1 \subseteq B_2).$$

The relation $\trianglelefteq$ is an order; every ascending sequence $B_1 \trianglelefteq B_2 \trianglelefteq \ldots \trianglelefteq B_i \ldots$ is stationary since $ASS$ is finite.

*Property 6.1.* [Conditional Convergence of the Extended Widening]
Let $\{B_i\}_{i\in\mathbf{N}}$ and $\{B'_i\}_{i\in\mathbf{N}}$ be two sequences of elements of *ASS* such that

1. $B'_i \sqsubseteq B_{i+1}$, for all $i \in \mathbf{N}$;
2. $B'_{i+1} = B_{i+1} \nabla B'_i$, for all $i \in \mathbf{N}$.

Then we have $B_i \leq B'_i$, for all $i \in \mathbf{N}^*$, and the sequence $\{B'_i\}_{i\in\mathbf{N}}$ is stationary.

PROOF. The fact that $B_i \leq B'_i$, for all $i \in \mathbf{N}^*$, is a direct consequence of the definition of the operation $\nabla$. Moreover, the hypotheses on the sequences ensure that $B'_1 \trianglelefteq B'_2 \trianglelefteq \ldots \trianglelefteq B'_i \ldots$; thus the sequence $\{B'_i\}_{i\in\mathbf{N}}$ is stationary. ∎

If all abstract operations are congruent with respect to $\sqsubseteq$ [3], each iteration of the abstract interpretation algorithm ensures that $B_{old} \sqsubseteq B_{new}$, where $B_{old}$ is the current value in *sat* and $B_{new}$ is the newly computed abstract sequence. Thus, Property 6.1 guarantees termination of the abstract interpretation algorithm. Congruence of the abstract operations with respect to $\sqsubseteq$ is ensured if they are "as accurate as possible" (which is achieved in [81]); however, proving this property entails a lot of work. A simpler solution consists of testing whether $B_{old} \sqsubseteq B_{new}$ actually holds before each application of the extending widening. If the condition does not hold, we switch to a cruder form of widening, which simply merges all successive results.

**Comparison with our Cardinality Analysis.** The determinacy information inferred by means of Sahlin's domain is in general less accurate than our cardinality analysis (except maybe in some partial evaluation contexts). For instance, with the former domain, it is not possible to detect mutually exclusive clauses except when cuts occur in the clauses. As illustrated in Section 5.3, the information provided by the abstract substitution component of our domain is instrumental to detect sure failure, sure success, and mutual exclusion, which all contribute to improve the accuracy of the determinacy (or cardinality) analysis. Nevertheless, the specific information about the sequence structure is finer grained in Sahlin's domain than in ours. Consider the abstract sequence $\{\mathcal{L}, 1\}$; it is approximated, in our domain, by $\langle 0, 1, pt\rangle$, which is actually equivalent to $\{\mathcal{L}, 0, 1, 1'\}$. Thus, it could be interesting to design a domain for abstract sequences similar to our cardinality domain, where the sequence component coincides with Sahlin's domain.

## *6.2. Giacobazzi and Ricci's Analysis of Determinate Computations*

The work of R. Giacobazzi and L. Ricci, described in [39], is also worth being reviewed in our context. They propose an analysis of functional dependencies [72] between procedure arguments of the success set of pure logic programs. Their analysis is a bottom-up abstract interpretation, based on [3, 33]. The analysis also infers groundness information and is intended to be used for parallel logic program optimization. In our comparison, we focus on the functional dependencies and we simplify the presentation in order to concentrate on the salient points. First, we provide a definition of functional dependency tailored to our framework. The definitions use some notions from Section 5.3.

[3] We would have written *monotonic* if the relation $\sqsubseteq$ was an order, not a pre-order only.

*Definition 6.5.* [Functional Dependency]
Let $\langle sv, frm\rangle$ be a structural abstract substitution over domain $D$ and set of indices $I$. A *functional dependency for* $\langle sv, frm\rangle$, denoted by $J \to j$, is a pair consisting of a subset $J$ of $I$ and an index $j \in I$.
Let $S \in PSS_D$ be a program substitution sequence such that $Subst(S) \subseteq Cc\langle sv, frm\rangle$. We say that the functional dependency $J \to j$ *holds in* $S$ *for* $\langle sv, frm\rangle$, if for all families of terms $(t_i)_{i\in I}$, $(t'_i)_{i\in I}$ that are decompositions of some program substitutions of $Subst(S)$, the following implication is true:

$$(t_i)_{i\in J} = (t'_i)_{i\in J} \Rightarrow t_j = t'_j.$$

Then we define an abstract domain to express functional dependencies.

*Definition 6.6.* [Abstract Sequences with Functional Dependencies]
*An abstract sequence with functional dependencies* is a triple $\langle sv, frm, fd\rangle$ where $\langle sv, frm\rangle$ is a structural abstract substitution over domain $D$ and set of indices $I$, and $fd$ is a set of functional dependencies for $\langle sv, frm\rangle$. The concretization function for abstract sequences with functional dependencies is defined by

$$Cc\langle sv, frm, fd\rangle = \left\{ S \in PSS_D \;\middle|\; \begin{array}{l} Subst(S) \subseteq Cc(\langle sv, frm\rangle) \text{ and} \\ J \to j \text{ holds in } S \text{ for } \langle sv, frm\rangle, \\ \text{for every } J \to j \in fd. \end{array} \right\}.$$

In fact, the functional dependency component $fd$ is best viewed as an additional component to the cardinality domain defined in Section 5, since its usefulness for determinacy analysis depends on the availability of mode information. Let $S \in CPSS_D$ be a canonical program substitution sequence. We say that $S$ is *functional* if the set $Subst(S)$ is empty or is a singleton. Such sequences model the behavior of procedures that cannot produce two or more distinct solutions. Assume that $S$ is the output sequence corresponding to the input substitution $\theta$, for some procedure $p$. Assume that $\theta \in Cc\langle sv, frm\rangle$ and $S \in Cc\langle sv', frm', fd'\rangle$ where $\langle sv', frm'\rangle$ is more instantiated than $\langle sv, frm\rangle$. We can infer that $S$ is functional if there exists $J \subseteq I'$ such that $fd'$ contains a functional dependency of the form $J \to i$, for every $i \in sv'(D)$, and if every term $t_j$ corresponding to an index $j \in J$ in a program substitution of $S$ is not more instantiated than the corresponding term in $\theta$. The latter information is easily deduced if we know, for instance, that $t_j$ is ground or is a variable. Thus adding a functional dependency component to our cardinality domain allows us to infer that output program substitution sequences are functional.

It is important to point out that the new component $fd$ expresses a property of program substitution sequences, not a property of (single) program substitutions. It is meaningless to use functional dependencies in a domain of abstract substitutions, because a set of functional dependencies determines a (two valued) condition on a set of program substitution. Either the set verifies the condition, then no constraint is added, or it does not and the set is rejected as a whole. Thus, a component $fd$ defines a *set of sets of* program substitutions. As a consequence, functional dependencies cannot be handled by previous top-down abstract interpretation frameworks such as [8, 61, 68, 71, 76, 93, 95]. However the abstract interpretation framework used by [39] is bottom-up and abstracts the success set of the program. The result of an analysis represents a set of possible success sets,

i.e., *a set of sets of output patterns*, which is similar to a set of sets of program substitutions. As far as we know, it is the first time that this difference of expressivity between bottom-up and (previous) top-down abstract interpretation frameworks is pointed out in the literature. The comparison usually concentrates on the fact that bottom-up frameworks are *goal independent*, i.e., they provide information on the program as a whole, while top-down frameworks are *goal dependent*, i.e., they provide information about the program *and* a given initial goal. We believe that a more fundamental difference lies in the fact that top-down frameworks are *functional*, i.e., they abstract the behavior of a program by a function between sets of sets, while bottom-up frameworks are *relational*, i.e., they abstract the behavior of a program by a set of relations. The difference between the two approaches has been previously put forward by Cousot and Cousot [22], but not in the context of logic programs. The functional approach can easily focus on small parts of the program behavior but looses the dependencies between inputs and outputs; the converse holds for the relational approach. Our novel framework is basically functional, but the domain of abstract sequences is in some sense relational; thus the framework allows us to combine the advantages of both approaches.

### *6.3. Debray and Warren's Analysis of Functional Computations*

In the previous section, we have shown that functional dependencies are useful to infer that an output program substitution sequence is functional, i.e., does not contain two or more distinct program substitutions. Such a sequence may contain several occurrences of the same program substitution, however. The importance of functional computations for logic program optimization was advocated early by S. Debray and D. Warren in [30]. In this paper, these authors propose a sophisticated algorithm to infer functional computations of a logic program. The analysis exploits functional dependencies and mode information, as well as a set of sufficient conditions to detect mutually exclusive clauses. Their algorithm is not based on abstract interpretation and assumes that functional dependencies and mode information are given from outside. Thus the algorithm considers an annotated program; it uses a set $\{\bot, \mathbf{true}, \mathbf{false}\}$ where $\bot$ is an initializing value, **true** means that a procedure is functional and **false** means that it is not known whether the procedure is functional. Hence, the set can be viewed as a domain of abstract sequences, with concretization function $Cc : \{\bot, \mathbf{true}, \mathbf{false}\} \to \wp(CPSS)$ defined by

$$\begin{aligned} Cc(\bot) &= \{< \bot >\}; \\ Cc(\mathbf{true}) &= \{S \in CPSS \mid \mathit{Subst}(S) \text{ is empty or is a singleton.}\}; \\ Cc(\mathbf{false}) &= CPSS. \end{aligned}$$

All aspects of their analysis can be accommodated in our approach by providing suitable abstract domains. An abstract domain consisting of our cardinality domain augmented with a functional dependency component would probably be fairly accurate. Moreover, in our approach, all analyses can be performed at the same time and interact with each other, making it possible to get a better accuracy.

## 7. CONCLUSION

This paper has introduced a novel abstract interpretation framework, capturing the depth-first search strategy and the cut operation of Prolog. The framework is

based on the notion of substitution sequences and the abstract semantics is defined as a pre-consistent post-fixpoint of the abstract transformation. Abstract interpretation algorithms need chain-closed domains and a special widening operator to compute the semantics. This approach overcomes some of the limitations of previous frameworks. In particular, it broadens the applicability of the abstract interpretation approach to new analyses and can potentially improve the precision of existing analyses. On the practical side, in this paper, we have only shown that our approach allows one to integrate - efficiently and at a low conceptual cost - a predicate level analysis (i.e., determinacy analysis) to variable level analyses classically handled by abstract interpretation. However, the improvement on classical analyses is marginal because, due to our design choices for the abstract sequence domain (i.e., a simple extension of `Pattern`), the new system behaves almost as the previous version of GAIA for variable level analyses. Nevertheless, the new framework opens a door for defining and exploiting more sophisticated domains for abstract sequences.

## REFERENCES


1. K. Apt. *From Logic Programming to Prolog.* International Series in Computer Science, Prentice Hall, 1997.
2. R. Barbuti and R. Giacobazzi. A Bottom-Up Polymorphic Type Inference in Logic Programming. *Science of Computer Programming*, 19(3):281–313, 1992.
3. R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-Based Bottom-Up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 15(1):133-181, 1993.
4. R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog Control. *Journal of Logic and Computation*, 3(6):579–603, 1993.
5. M. Baudinet. Proving Termination Properties of Prolog Programs: a Semantic Approach. *Journal of Logic Programming*, 14(1&2):1–29, 1992.
6. C. Braem, B. Le Charlier, S. Modard and P. Van Hentenryck. Cardinality Analysis of Prolog. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.
7. C. Braem and S. Modard. Abstract Interpretation for Prolog with Cut: Cardinality Analysis. Master's thesis, Institut d'Informatique, University of Namur, Belgium, September 1994.
8. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, 1991.
9. F. Bueno and M. Hermenegildo. Results on Automatic Translation from Prolog to the Andorra Kernel Language. Technical Report, Universidad Politecnica de Madrid, Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1991.
10. D. Cabeza Gras and M. Hermenegildo. Extracting Non-Strict Independent And-Parallelism Using Sharing and Freeness Information. In [55], pages ? 1994.
11. J.H. Chang, A.M. Despain, and D. DeGroot. And-Parallelism of Logic Programs based on a Static Data Dependency Analysis. In *Proceedings of the 30th IEEE Compcon Spring (COMPCON'85)* , Los Alamitos, California, 1985. IEEE Computer Sciety Press.

12. M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.

13. P. Codognet and G. Filé. Computations, Abstractions and Constraints in Logic Programs. In *Proceedings of the Fourth International Conference on Computer Languages (ICCL'92)*, Oakland, April 1992.

14. M.M. Corsini. (Yet) an Abstract Domain and Unification for Accurate Groundness and Sharing Analysis based on Graphs Traversing. In *ICLP'91 Pre-Conference Workshop on Semantics-Based Analysis of Logic Programs*, INRIA Rocquencourt, France, June 1991.

15. A. Cortesi and G. Filé. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, 1991. ACM Press.

16. A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, July 1991. IEEE Computer Society Press.

17. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *Proceedings of the 21th ACM Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.

18. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

19. 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 (POPL'77)*, Los Angeles, California, January 1977. ACM Press.

20. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages (POPL'79)*, Los Angeles, California, January 1979. ACM Press.

21. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

22. P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

23. P. Cousot and R. Cousot. Comparing of the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation (invited paper). In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, LNCS, Leuven, Belgium, August 1992. Springer-Verlag.

24. P. Cousot and R. Cousot. Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages). (Invited paper). In *Proceedings of the Sixth International Conference on Computer Languages (ICCL'94)*, Toulouse, France, May 1994. IEEE Computer Society Press, Los Alamitos, California. (Invited paper).

25. S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In *Proceedings of the Tenth International Conference on Logic Programming (ICLP'93)*, Budapest, Hungary, June 1993. MIT Press.

26. A. de Bruin and E. de Vink. Continuation Semantics for Prolog with Cut. In *Proceedings of TAPSOFT'89*, LNCS, 1989. Springer-Verlag.

27. S.K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):418–450, 1989.

28. S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.

29. S.K. Debray and D.S. Warren. Automatic Mode Inference for Logic Programs. *Journal of Logic Programming*, 5(3):207–229, 1988.

30. S.K. Debray and D.S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):451–481, 1989.

31. B. Demoen, P. Van Roy, and Y.D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of TAPSOFT'87*, LCS vol. 250, 1987.

32. V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and their Experimental Evaluation. *Software Practice and Experience*, 23(4):419–459, 1993.

33. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behaviour of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

34. G. Filé and S. Rossi. Static Analysis of Prolog with Cut. *Proceedings of Fourth International Conference on Logic Programming and Automated Reasoning (LPAR'93)*, LNCS vol. 698, July 1993. Springer–Verlag.

35. G. Filé and F. Ranzato. Improving Abstract Interpretations by Systematic Lifting to the Powerset. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.

36. Y. Gang and X. Zhiliang. An Efficient Type System for Prolog. In *Proceedings of IFIP Congress 86*, 1986.

37. T. W. Getzinger. The Costs and Benefits of Abstract Interpretation-Driven Prolog Optimization. In [55], pages 1–25, 1994.

38. R. Giacobazzi and L. Ricci. Pipeline Optimizations in AND-Parallelism by Abstract Interpretation. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of Seventh International Conference on Logic Programming (ICLP'90)*, pages 291–305, Jerusalem, Israel, 1990. The MIT Press.

39. R. Giacobazzi and L. Ricci. Detecting Determinate Computations by Bottom-Up Abstract Interpretation. In *Proceedings of ESOP'92*, pages 167–181. Springer-Verlag, 1992.

40. M.V. Hermenegildo. An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In *Proceedings of Third International Conference on Logic Programming (ICLP'86)*, LNCS vol. 225, July 1986. Springer-Verlag.

41. M.V. Hermenegildo, R. Warren, and S.K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, 1992.

42. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP'89)*, Cleveland, Ohio, October 1989. MIT Press.

43. D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2&3):291–314, 1992.

44. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):205–258, 1992.

45. T.P. Jensen and T.Æ . Mogensen. A Backwards Analysis for Compile-Time Garbage Collection. In N. Jones, editor, *Proceedings Third European Symposium on Programming (ESOP'90)*, LNCS vol. 432, 1990. Springer-Verlag.

46. N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In Sten-Åke Tarnlund, editor, *Proceedings of the Second International Conference on Logic Programming, (ICLP'92)*, 1984.

47. N.D. Jones and H. Søndergaard. A Semantic-Based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, 1987. Ellis Horwood.

48. T. Kanamori and T. Kawamura. Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation. Technical report, ICOT, 1987.

49. T. Kanomori and K. Horiuchi. Type Inference in Prolog and its Application. In *Proceedings of 9th IJCAI*, pages 704–709, 1985.

50. R.B. Kieburtz. Precise Typing of Abstract Data Type Specification. In *Proceedings of Tenth ACM Symposium Principles of Programming Languages (POPL'83)*, 1983. ACM Press.

51. F. Kluźniak. Type Synthesis for Ground Prolog. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP'87)*, pages 788–816, Melbourne, Australia, May 1987. MIT Press.

52. F. Kluźniak. Compile-Time Garbage Collection for Ground Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, Seattle, Washington, August 1988. MIT Press.

53. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.

54. B. Le Charlier, O. Degimbe, L. Michel, and P. Van Hentenryck. Optimization Techniques for General Purpose Fixpoint algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In Cousot P., editor, *Proceedings of the Third International Workshop on Static Analysis (WSA'93)*, LNCS vol. 724, Padova, September 1993. Springer-Verlag.

55. B. Le Charlier (Ed.). *Proceedings of the First International Static Analysis Symposium (SAS'94)*, LNCS vol. 864, Namur, Belgium, September 1994. Springer-Verlag.

56. B. Le Charlier, C. Leclère, S. Rossi and A. Cortesi. Automated Verification of Prolog Programs. *Journal of Logic Programming*, 39(1&3):3–42, 1999.

57. B. Le Charlier and S. Rossi. Sequence-Based Abstract Semantics of Prolog. Technical Report RR-96-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur, Belgium, February 1996.

58. B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework which Accurately Handles Prolog Search-Rule and the Cut. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.

59. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-Based Abstract Interpretation of Prolog. Technical Report RR-97-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur, Belgium, January 1997.

60. B. Le Charlier and P. Van Hentenryck. A General Top-Down Fixpoint Algorithm (revised version). Technical Report RR-93-022, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur, Belgium, June 1993.

61. B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, 1994.

62. B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. *Acta Informatica*, 32(3):209–270, 1995.

63. D. Leivant. Polymorphic Type Inference. In *Proceedings of Tenth ACM Symposium Principles of Programming Languages (POPL'83)*, 1983. ACM Press.

64. J.W. Lloyd. *Foundations of Logic Programming.* Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Verlag, second edition, 1987.

65. A. Marien and B. Demoen. On the Management of Choicepoint and Environment Frames in the WAM. In E. L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP'89)*, pages 1030–1047, Cleveland, Ohio, October 1989. MIT Press.

66. A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Proceedings of the Sixth International Conference on Logic Programming (ICLP'89)*, pages 33–47, Lisbon, Portugal, June 1989. MIT Press.

67. K. Marriott. Frameworks for Abstract Interpretation. *Acta Informatica*, 30(2):103–129, 1993.

68. K. Marriott and H. Søndergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. In *North American Conference on Logic Programming (NACLP'89)*, Cleveland, Ohio, 1989.

69. K. Marriott and H. Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. In G. Ritter, editor, *Proceedings of IFIP'89*, San Fransisco, California, 1989.

70. M. Meier. Recursion versus Iteration in Prolog. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.

71. C. Mellish. Abstract Interpretation of Prolog Programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 8, Ellis Horwood Limited.

72. A.O. Mendelzon. Functional Dependencies in Logic Programs. In *Proceedings of the Eleventh International Conference on Very Large Data Bases*, 1985.

73. A. Mulkers. *Deriving Live Data Structures in Logic Programs by Means of Abstract Interpretation.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 1991.

74. A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In D.S. Warren and P. Szeridi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, Jerusalem, Israel, June 1990. MIT Press.

75. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In K. Furukawa, editor, *Proceedings of the Eight International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.

76. K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2&3):315–347, 1992.

77. A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

78. U. Nilsson. Systematic Semantic Approximations of Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming (PLILP'90)*, LNCS vol. 456, Linköping, Sweden, August 1990. Springer-Verlag.

79. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus, 1981.

80. D. Sahlin. Mixtus: An Automatic Partial Evaluator for Full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

81. D. Sahlin. Determinacy Analysis for Full Prolog. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-B ased Program Manipulation (PEPM'91)*, 1991. ACM Press.

82. D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., 1988.

83. Z. Somogyi. A System of Precise Modes for Logic Programs. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming (ICLP'86)*, LNCS vol. 225, London, England, July 1986. Springer-Verlag.

84. F. Spoto and G. Levi. A Denotational Semantics for Prolog. In M. Falaschi, M. Navarro, and A. Policriti, editors, *Proceeding of APPIA-GULP-PRODE'97*, Grado, Italy, June 1997.

85. L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge Mass., 1986.

86. J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge Mass., 1977.

87. H. Tamaki and T. Sato. OLD-Resolution with Tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming (ICLP'86)*, LNCS vol. 225, London, England, July 1986. Springer-Verlag.

88. A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Proceedings of the Sixth International Conference on Logic Programming (ICLP'89)*, Lisbon, Portugal. Cambridge Mass., 1989.

89. K. Ueda. Making Exhaustive Search Programs Deterministic, part II. In J.L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP'87)*, volume 2, Melbourne, Australia. Cambridge Mass., 1987.

90. P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The Impact of Granularity in Abstract Interpretation of Prolog. In Cousot P., editors, *Proceedings of the Third International Workshop on Static Analysis (WSA'93)*, LNCS vol. 724, Padova, September 1993. Springer-Verlag.

91. P. Van Roy, B. Demoen, and Y.D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, (TAPSOFT'87)*, volume 2, LNCS vol. 250, Pisa, Italy, 1987.

92. P. Van Roy and A. Despain. High-Performance Computing with the Aquarius Compiler. *IEEE Computer*, 25(1):54-68, January 1992.

93. D.S. Warren. Memoization for Logic Programs. *Communications of the ACM*, 35(3), March 1992.

94. R. Warren, M.V. Hermenegildo, and S.K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, Seattle, Washington, August 1988. MIT Press.

95. W. Winsborough. Multiple Specialization Using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(4), 1992.

96. J. Xu and D.S. Warren. A Type Inference System for Prolog. In R.A. Kowalsky and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming (ICLP'88)*, Seattle, Washington, August 1988. MIT Press.

97. E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Journal of Logic Programming*, 10(1/2/3&4):125–153, 1991.

## APPENDIX

We complete here the description of the abstract operations started in Section 5.2. The correctness proofs of all the abstract operations can be found in [59]. The definitions below have been added in order to allow the reader to check the details of the examples in Section 5.2.

**Extension at Clause Entry**: $\texttt{EXTC}(c,\cdot) : AS_D \to ASSC_{D'}$
The implementation reuses the homonymous operation from the previous framework, which is specified as follows.

*Operation* $\texttt{EXTC}(c,\cdot) : AS_D \to AS_{D'}$
Let $\beta \in AS_D$, $\theta \in CPS_D$, and $\theta' \in PS_{D'}$ such that $x_i\theta' = x_i\theta$ $(\forall i : 1 \leq i \leq n)$ and $x_{n+1}\theta'$, ..., $x_m\theta'$ are distinct standard variables not belonging to $codom(\theta)$. Then

$$\theta \in Cc(\beta) \Rightarrow [\![\theta']\!] \in Cc(\texttt{EXTC}(c,\beta)).$$

Hence, the EXTC operation on sequences is defined by

$$\texttt{EXTC}(c,\beta) = \langle\langle \texttt{EXTC}(c,\beta), 1, 1, st\rangle, nocut\rangle.$$

**Restriction at Clause Exit**: $\texttt{RESTRC}(c,\cdot) : ASSC_{D'} \to ASSC_D$
The treatment of this operation is similar to the previous one. We first specify the abstract substitution version of the operation.

*Operation* $\texttt{RESTRC}(c,\cdot) : AS_{D'} \to AS_D$
Let $\beta \in AS_{D'}$ and $\theta \in CPS_{D'}$. We have

$$\theta \in Cc(\beta) \Rightarrow [\![\theta_{|D}]\!] \in Cc(\texttt{RESTRC}(c,\beta)).$$

Hence, the RESTRC operation on sequences is defined by

$$\texttt{RESTRC}(c,C) = \langle \texttt{RESTRC}(c,\beta), m, M, acf\rangle.$$

**Restriction before a Call**: $\texttt{RESTRG}(l,\cdot) : AS_{D'} \to AS_{D'''}$
This operation is simply inherited from the previous framework.

**Unification of a Variable and a Functor**: $\texttt{UNIF-FUNC}(f,\cdot) : AS_D \to ASS_D$
The treatment of this operation is identical to the treatment of the UNIF-VAR operation and then is omitted.

**Extension of the Result of a Call**: $\texttt{EXTGS}(l,\cdot,\cdot) : ASSC_{D'} \times ASS_{D'''} \to ASSC_{D'}$
This operation reuses the operation EXTG from the previous framework. The reused operation has to fulfill the specification just below.

*Operation* $\texttt{EXTG}(l,\cdot,\cdot) : AS_{D'} \times AS_{D'''} \to AS_{D'}$
Let $\beta_1 \in AS_{D'}$ and $\beta_2 \in AS_{D'''}$. Let $\theta_1 \in CPS_{D'}$ and $\theta_2 \in PS_{D'''}$ be such that $x_{i_j}\theta_1 = x_j\theta_2$ $(\forall j : 1 \leq j \leq n')$. Let $\sigma \in SS$ such that $dom(\sigma) \subseteq codom(\theta_2)$. Let $\{z_1,\ldots,z_r\} = codom(\theta_1) \setminus codom(\theta_2)$. Let $y_1,\ldots,y_r$ be distinct standard variables not belonging to $codom(\theta_1) \cup codom(\sigma)$. Let $\rho = \{z_1/y_1,\ldots,z_r/y_r, y_1/z_1,\ldots,y_r/z_r\}$. Under these assumptions,

$$\left.\begin{array}{r}\theta_1 \in Cc(\beta_1),\\ \theta_2\sigma \in Cc(\beta_2)\end{array}\right\} \Rightarrow [\![\theta_1\rho\sigma]\!] \in Cc(\texttt{EXTG}(l,\beta_1,\beta_2)).$$

The implementation of EXTGS is as follows.

$$\begin{array}{lll}
\beta' & = \texttt{EXTG}(l,\beta_1,\beta_2); & \\
m' & = m_1 m_2 & \text{if } t_2 = st,\\
 & = \min(1,m_1)m_2 & \text{otherwise;}\\
M' & = \min(1,M_1)M_2 & \text{if } t_2 = snt,\\
 & = M_1 M_2 & \text{otherwise;}\\
t' & = snt & \text{if } t_1 = snt \text{ or } (t_2 = snt \text{ and } m_1 \geq 1),\\
 & = st & \text{if } t_1 = st \text{ and } (t_2 = st \text{ or } M_1 = 0),\\
 & = pt & \text{otherwise;}\\
acf' & = acf. &
\end{array}$$

**Operation** $\texttt{SEQ} : ASSC_D \to ASS_D$
We define

$$\texttt{SEQ}(\langle B, acf\rangle) \;=\; B.$$

**Operation** $\texttt{SUBST} : ASSC_{D'} \to AS_{D'}$
We define

$$\texttt{SUBST}(\langle\langle \beta, m, M, t\rangle, acf\rangle) \;=\; \beta.$$