An Abstract Interpretation Framework for (almost) Full Prolog

B. Le Charlier[1]
S.Rossi[2]
P. Van Hentenryck[3]

Technical Report No. CS-94-16

April, 1994

[1]University of Namur 21 rue Grandgagnage,B-5000 Namur(Belgium)
[2]University of Padova 7 via Belzoni I-35131 Padova (Italy)
[3]Brown University, Computer Science Department Box 1910, Providence, RI 02912

An Abstract Interpretation Framework for (almost) Full Prolog[1]

B. Le Charlier
University of Namur
21 rue Grandgagnage
B-5000 Namur (Belgium)

S. Rossi
University of Padova
7 via Belzoni
I-35131 Padova (Italy)

P. Van Hentenryck
Brown University
Box 1910, Providence
RI 02912 (USA)

Abstract

A novel abstract interpretation framework is introduced, which captures Prolog depth-first strategy and the cut operation. The framework is based on a new conceptual idea, the notion of substitution sequences, and the traditional fixpoint approach to abstract interpretation. It broadens the class of analyses that are amenable in practice to abstract interpretation and refines the precision of existing analyses. Its practicability is demonstrated in a companion paper [4]. This paper focuses on theoretical foundations.

1 Introduction

Abstract interpretation has been shown to be a valuable tool to obtain high-performance implementation of Prolog [24, 25]. Yet traditional abstract interpretation frameworks of Prolog (e.g. [5, 19, 20]) usually ignores many features of Prolog, such as the depth-first search strategy and the cut operation. Still these frameworks are very valuable, since they allow many analyses such as types, modes, and sharing to be performed with good accuracy and they were fundamental in developing practical analysis tools. However, as the technology matures, their limitations become more apparent. In particular, they lead to the following two inconvenients:

1. The precision of the analysis is inherently limited for some classes of programs. A typical example is the definition of multi-directional procedures, using cuts and meta-predicates to select among several versions. Ignoring the depth-first search strategy and the cut prevents the compiler from performing various important compiler optimizations such as dead-code elimination [4].

2. The existing frameworks are not expressive enough to capture certain analyses in their entirety. A typical example is determinacy analysis, where existing approaches either resort to special-purpose proofs (e.g. [22]) or their frameworks ignore certain aspects of the analysis, e.g. the cut and/or how to obtain the determinacy information from input/output patterns (e.g. [13, 9]).

This paper proposes a step in overcoming these limitations. A novel abstract interpretation framework is introduced, which captures the depth-first seach strategy and the cut operation (only dynamic predicates such as `assert/retract` are ignored). The key conceptual idea underlying the framework is the notion of substitution sequences which models the successive answer substitutions of a Prolog goal. This notion enables the framework to deduce and reason about information not available in most frameworks, such as sure success and failure, the number of solutions, and/or

[1]Partly supported by the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225 and the National Science Foundation under grant numbers CCR-9357704 and a NSF National Young Investigator Award.

termination, broadening the class of applications amenable to abstract interpretation and improving the accuracy of existing analyses.

The main technical contribution of this paper is to show how to apply the traditional fixpoint approach [8] to the conceptual idea. A main difficulty lies in the fact that the abstract semantics cannot simply be defined as the least fixpoint of the abstract transformation obtained from the collecting semantics, since the least fixpoint of the transformation obtained by "lifting" the concrete semantics to sets of substitution sequences is not a consistent approximation of the concrete semantics. The notion of pre-consistent postfixpoint is introduced to remedy this problem. The practical consequences of this formalization are discussed and include the need for so-called upper-closed abstract domains and a special form of widening in the abstract interpretation algorithm. The paper also specifies precisely all the abstract operations of the framework through consistency conditions.

It is important to point out that this paper is more than yet another abstract interpretation framework. The framework was motivated by computational considerations and its practicability and simplicity have been demonstatred on a cardinality analysis described in a companion paper [4]. The cardinality analysis, which is an instantiation of the framework to a specific abstract domain, approximates among other things the number of solutions to a goal. It is shown that the analysis requires only a small overhead compared to a mode/sharing analysis and outperforms existing determinacy analyses in precision.

The rest of the paper is organized as follows. Section 2 motivates the paper through two extremely simple examples and gives an overview of the framework. Section 3 is an informal presentation of the main technical difficulties on a single example as well as the adopted solutions. Section 4 sketches the concrete semantics. Section 5 and 6 contain respectively the specification of the abstract operations and the abstract semantics. Section 7 discusses the abstract interpretation algorithm while section 8 presents related work, in particular the work of File and Rossi [12] and Barbuti et al. [2]. Section 9 concludes the paper. The appendices describe the concrete semantics and sketch the proofs of the main technical results.

2 Overview of the Framework

Two Simple Examples We start by two extremely simple examples which are not handled well by existing abstract interpretation frameworks. Consider the program

p(a).	*q(b).*
p(X) :- q(X).	*q(c).*

Assume that we are interested in determinacy analysis of *p/1* called with a ground argument. Examinations of the clauses in isolation will not determine the determinacy of the goal. This was recognized in several places (e.g. [13, 9]) which proposes to use input/output patterns to remedy the problem. However, these works focus on determining the patterns and cannot integrate all aspects of the analysis in a single abstract interpretation framework. As a consequence, they need special-purpose proofs for the final part of the analysis losing the simplicity of the abstract interpretation framework. Similarly these work do not take control (depth-first search and the cut) into account which reduces the precision of the analysis. Our framework handles all aspects of the analysis in a single framework. Consider now the extension of the previous program

r(X) :- p(X), ! .
r(d).

and assume that *r/1* is called with a variable. Abstract interpretation frameworks ignoring the search rule and the cut cannot infer that *p(X)* surely succeeds and hence that the cut is executed. In fact, they simply ignore the cut and would conclude when instantiated with a type domain (e.g. [14, 7]) that r produces an element from $\{a, b, c, d\}$. Using our framework, it is possible to design an analysing concluding that *r/1* only produces the element a. We now describe informally the basic ideas on how to obtain such an analysis.

Concrete Semantics The starting point of our approach is a concrete semantics which associates with a program P a total function from the set of pairs $\langle\theta, p\rangle$ to the set of substitution sequences, where p is a predicate symbol and θ is a substitution. The novelty is the notion of substitution sequences which models the sequence of computed answer substitutions (e.g. [18]) produced by the execution of $p(x_1, \ldots, x_n)\theta$. The resulting sequence can have different shapes. If the execution terminates (producing m computed answer substitutions), S is a *finite* sequence $< \theta_1, \ldots, \theta_m >$. If the execution produces m computed answer substitutions and then enters into an infinite loop, then S is an *incomplete* sequence $< \theta_1, \ldots, \theta_m, \bot >$, where $\bot$ models non termination [3]. Finally, if the execution produces an infinite number of computed answer substitutions, then S is an *infinite* sequence $< \theta_1, \ldots, \theta_i, \ldots >$ $(i \in \mathbf{N})$. We note $SUBST(S)$ the set of substitutions in S. The concrete semantics internally manipulates slightly more complex objects, substitution sequences with cut, to take the cut into account.

Abstract Semantics The abstract semantics works with description of sequences called abstract sequences. It associates with a program a total function which, given a pair $\langle\beta, p\rangle$ (where β is an abstract substitution), returns an abstract sequence B, whose informal semantics can be described as follows:

> "The execution of $p(x_1, \ldots, x_n)\theta$ with θ satisfying the property β produces a substitution sequence S satisfying the property described by B."

It is important to realize that abstract domains for sequences need not be much more complicated than traditional abstract domains. We illustrate this with two examples.

Abstract Domain 1: An abstract sequence $B \in ASS$ is of the form $\langle\beta, m, M, t\rangle$ where β is an abstract substitution, $m \in \mathbf{N}$, $M \in \mathbf{N} \cup \{\infty\}$, $t \in \{snt, st, pt\}$. The concretization function $Cc : ASS \to CSS$ maps B on the set of substitution sequences S such that any substitution θ which is an element of S belongs to $Cc(\beta)$ (the set of substitutions described by β); the number of elements of S, excluding $\bot$, is not smaller than m and not greater than M. Additionally, the sequences are finite if $t = st$ and incomplete or infinite if $t = snt$. (*snt* means "sure non termination", *st* means "sure termination" and *pt* stands for "possible termination".) The abstract domain is used in the companion paper and, on our first program, the abstract semantics defines $\langle p(\{a, b, c\}), 0, 1, st)\rangle$ as the result of a query *q(ground)*. Note that the domain is not too complex computationally.

Abstract Domain 2: The first abstract domain does not achieve maximal precision on the second example. A more precise domain consists of abstract sequences B of the form $\langle <\beta_1,\ldots,\beta_m>,\beta,m,M,t\rangle$, where m, M and t are given the same meaning as before. If $S \in Cc(B)$, it must be of the form $<\theta_1,\ldots,\theta_m>:: S'$ with $\theta_i \in Cc(\beta_i)$ $(1 \leq i \leq m)$, where :: denotes the usual concatenation operation on sequences. Moreover, each substitution in S' must belong to $Cc(\beta)$. On the second example, the abstract semantics defines $\langle <p(\{a\}),p(\{b\}),p(\{c\})>,p(\{\}),3,3,st)\rangle$ as the result of *p(var)*. The new domain is likely to be computationally reasonable, since they are few situations where a large number of abstract substitutions will be maintained.

Abstract Interpretation Algorithm The last step of the analysis is the computation of the abstract semantics with extensions of existing algorithms such as *GAIA* [17] and *PLAI* [21].

3 Technical Difficulties and Adopted Solutions

The foundation of this work is the fixpoint approach to abstract interpretation [8]. Starting from a concrete semantics, we try to define a collecting semantics, an abstract semantics approximating the collecting semantics, and an algorithm to compute part of the abstract semantics. Applying this approach to the above informal ideas leads to some novel theoretical and practical problems.[2] The main problem is that the abstract semantics can no longer be defined as the least fixpoint of the basic transformation obtained by "lifting" the concrete semantics to sets of substitution sequences. In this section, we illustrate these problems and their proposed solutions on a simple example.

Concrete Semantics Consider the following program

repeat.
repeat :- repeat.

The concrete semantics of this program maps the input $<\epsilon, \mathit{repeat}>$, where ϵ is the empty substitution, to the infinite sequence $<\epsilon,\ldots,\epsilon,\ldots>$. This comes from the fact that the result S is described as the least fixpoint of a transformation $\tau_1 : PSS \rightarrow PSS$:

$$\tau_1 S = <\epsilon>:: S.$$

where *PSS* is the set of substitution sequences. Operationally, this expresses that the first clause first succeeds once producing the result ϵ. The second clause then succeeds exactly as many times as the recursive call, producing the same sequence of results. *PSS* can be endowed with the following ordering: $S_1 \sqsubseteq S_2$ iff either $S_1 = S_2$ or there exist $S, S' \in PSS$ such that $S_1 = S ::<\perp>$ and $S_2 = S :: S'$. *PSS* is then a pointed *cpo* with minimal element $<\perp>$. τ_1 is continuous and has a least fixpoint which is computed as follows: $S_0 =<\perp>$, $S_{i+1} =<\epsilon>:: S_i =<\epsilon,\ldots,\epsilon,\perp>$ (with i occurrences of ϵ), and $lfp(\tau_1) = \sqcup_{i=0}^{\infty} S_i =<\epsilon,\ldots,\epsilon,\ldots>$ as expected.

Collecting Semantics The technical problems arise when we "lift" the semantics to sets of substitution sequences. The "collecting" semantics associates with the program the transformation $\tau_2 : \wp(PSS) \rightarrow \wp(PSS)$ defined by

$$\tau_2 \Sigma = \{<\epsilon>:: S \ : \ S \in \Sigma\}.$$

[2] Note that similar problems have been encountered in functional programming [1].

$\wp(PSS)$ is a complete lattice for set inclusion and τ_2 is monotonic. However, $lfp(\tau_2)$ is not a consistent approximation of $lfp(\tau_1)$ (i.e. $lfp(\tau_1) \notin lfp(\tau_2)$), since $lfp(\tau_2)$ is the empty set. Note however that τ_2 is consistent with respect to τ_1 in the following sense: for all $S \in PSS$ and for all $\Sigma \in \wp(PSS)$, $S \in \Sigma$ implies $\tau_1(S) \in \tau_2(\Sigma)$.

The first cause of inconsistency of $lfp(\tau_2)$ is that S_0, the first iterate in the Kleene sequence for $lfp(\tau_1)$, obviously does not belong to the first iterate of the Kleene sequence for $lfp(\tau_2)$ (which is empty). In order to get a consistent approximation of $lfp(\tau_1)$, we may attempt to build another sequence of sets of substitution sequences as follows:

$$\Sigma_0 = \{< \perp >\}, \qquad \Sigma_{i+1} = \tau_2 \Sigma_i = \{< \epsilon, \ldots, \epsilon, \perp >\} \ (i \geq 0).$$

The problem is that this sequence is not increasing with respect to inclusion.

This new problem could possibly be solved by using another ordering on (some subset of) $\wp(PSS)$. This ordering should in a way combine the ordering on PSS and inclusion in $\wp(PSS)$. The traditional solution to this problem in denotational semantics consists in using a power domain construction (e.g. [23]). Although this solution is elegant theoretically, it is somewhat heavy for an abstract interpretation framework which should lead to efficient implementations. We adopted a solution which is less natural from a denotational standpoint but leads to effective analyses as demonstrated by the companion paper [4]. The solution is best presented in three steps.

First, τ_2 is replaced by a transformation τ_3:

$$\tau_3 \Sigma = \Sigma \cup \tau_2 \Sigma.$$

τ_3 is *extensive* (i.e. $\Sigma \subseteq \tau_3 \Sigma$ for all Σ). In addition, the sequence defined by $\Sigma_0 = \{< \perp >\}$ and $\Sigma_{i+1} = \tau_3 \Sigma_i$ is increasing and its limit is the set:

$$\Sigma_\infty = \bigcup_{i=0}^{\infty} \Sigma_i = \{< \perp >, < \epsilon, \perp >, \ldots, < \epsilon, \ldots, \epsilon, \perp >, \ldots\}.$$

Σ_∞ contains the entire Kleene sequence for $lfp(\tau_1)$ but still not $lfp(\tau_1)$ itself.

The second step is thus to complete increasing chains of sets of substitution sequences (with respect to $\sqsubseteq$) with their limits. Sets of substitution sequences so completed are called *upper-closed* and we denote by CSS the set of such upper-closed sets[3]. τ_2 and τ_3 can be redefined over CSS. The upper bound operation $\bigsqcup$ in CSS is no longer $\bigcup$: it adds to the union the limit of every chain in the union. Applying the new construction to τ_3 leads to the result $\Sigma_\infty = \bigsqcup_{i=0}^{\infty} \Sigma_i$ which contains all non finite sequences of empty substitutions.

The last step of our construction consists in refining this correct but imprecise result. Instead of starting the iteration with τ_3, τ_2 is used during an arbitrary number of steps before switching to τ_3. Since each iterate for τ_2 contains the corresponding iterate for τ_1, switching to τ_3 after i steps guarantees that the set Σ_∞ contains all iterates from the i-th and also the limit, since sets are upper-complete. In the above example, we deduce that *repeat* produces at least i results.

Abstract Computation The construction can be adapted to the abstract semantics by using consistent abstractions of τ_2 and τ_3. However, if the abstract domain is not noetherian, a widening operation must be used instead of the upper bound operation to ensure the finiteness of the analysis.

[3] Upper-closed sets can actually be viewed as a (simple) form of power domain construction.

Let us consider this last case. Consider an abstract domain ASS with a concretization function $Cc : ASS \rightarrow CSS$ and with an element B_0 such that $< \perp > \in Cc(B_0)$. Consider also an abstract version on τ_4 of τ_2, i.e.

$$\forall S \in PSS \;\; \forall B \in ASS : S \in Cc(B) \Rightarrow \tau_1 S \in Cc(\tau_4 B).$$

The computation in the abstract domain iterates τ_4 for j steps:

$$B_{i+1} = \tau_4 B_i (0 \leq i \leq j).$$

Then, *unless a fixpoint has already been reached*, the computations "jumps" to a value B_w such that $B_j \leq B_w$ and $\tau_4 B_w \leq B_w$.

The process is sound for the following reason. Let S_j be the iterates to $lfp(\tau_1)$. Since $S_0 = \langle \perp \rangle \in Cc(B_0)$ and τ_4 is consistent, $S_j \in Cc(B_j)$ by induction. Since $B_j \leq B_w$ and B_w is a postfixpoint, $S_k \in Cc(B_w)$ for all $k \geq j$ because $S_k \in Cc(B_w) \Rightarrow S_{k+1} \in Cc(\tau_4 B_k)$, by consistency of τ_4. Hence, $S_{k+1} \in Cc(B_w)$ by $\tau_4 B_w \leq B_w$ and monotonicity of Cc. Finally, since $Cc(B_w)$ is upper-closed, $lfp(\tau_1) \in Cc(B_w)$.

To illustrate the process on a concrete example, consider the first abstract domain, dropping the abstract substitution part since it is useless. τ_4 is defined by $\tau_4 \langle m, M, t \rangle = \langle m+1, M+1, t \rangle$ and $B_0 = \langle 0, 0, snt \rangle$. The first iterations give $B_j = \langle j, j, snt \rangle$. To get a postfixpoint, the second j is replaced by ∞ to obtain $B_w = \langle j, \infty, snt \rangle$, since $\tau_4 B_w = \langle j+1, \infty, snt \rangle \leq B_w$. B_w is a consistent approximation of $lfp(\tau_1)$ and expresses that at least j substitutions are generated and that the procedure surely loops. We do not know however if it loops after giving a finite number of substitutions or if it produces an infinite number of substitutions.

Theoretical Implications The above construct implies that the abstract semantics can no longer be defined as the least fixpoint of the abstract transformation obtained by abstracting the collecting semantics. The abstract semantics is defined as certain postfixpoints of the abstract transformation (see the definition of pre-consistent set of abstract tuples later on).

Practical Implications In practice, the construct imposes two requirements on the abstract domain. First, it is necessary to make sure that the concretization function only returns upper-closed sets. This requirement, which is satisfied by our two abstract domains, does not seem to be too restrictive in practice. Second, the designer needs to decide when to apply the widening operation. This is of course domain-dependent. A heuristic is to let the decision be driven by the substitution part of the domain. The widening on abstract sequences is applied when this part stabilizes. This is the choice adopted in the companion paper and it seems to give an effective tradeoff between precision and efficiency.

4 Concrete Semantics

Space restrictions forbid us to include the concrete semantics in the paper (see the appendix). The concrete semantics is a fixpoint semantics defined on normalized programs [5], i.e. clause heads are of the form $p(x_1,\ldots,x_n)$ and bodies contain atoms of the form $p(x_{i_1},\ldots,x_{i_n})$, $x_i = x_j$, $x_{i_1} = f(x_{i_2},\ldots,x_{i_n})$, and !. To simplify the traditional problems with renaming, we use two sets of variables and substitutions [17]. (Program) substitutions (denoted by θ) are of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where the t_i are terms and the x_i are (so-called) *program* variables (parameters). We assume another infinite (disjoint) set of (so-called) *standard* variables. The t_i's may only contain standard variables. By definition, $dom(\theta) = \{x_1, \ldots, x_n\}$ and $codom(\theta)$ is the set of variables in the t_i's. We also use *standard* substitutions which are substitutions in the usual sense.They are denoted by σ possibly subscripted and only use standard variables. *mgu*'s are standard substitutions. The composition $\theta\sigma$ of a program substitution with a standard substitution is defined in a non standard way by $\theta\sigma = \{x_1/t_1\sigma, \ldots, x_n/t_n\sigma\}$. We note *PS* the set of program substitutions.

The concrete semantics uses objects of the form $\langle\theta, p\rangle$, $\langle\theta, pr\rangle$, $\langle\theta, c\rangle$, and $\langle\theta, g\rangle$, where p, pr, c, g are respectively a predicate name, a procedure, a clause, and the body or a prefix of the body of a clause. It also uses substitution sequences and objects of the form $\langle S, cf\rangle$, where S is a substitution sequence and $cf \in \{cut, nocut\}$. Objects of the form $\langle\theta, p\rangle$ are mapped to substitution sequences which model the sequence of answer substitutions produced by p for θ. Objects of the form $\langle\theta, c\rangle$ and $\langle\theta, g\rangle$ are mapped to objects of the form $\langle S, cf\rangle$, where *cf* indicates whether the execution of the clause or of the prefix has been cut. Assuming an underlying program P, we note by $\langle\theta, p\rangle \longmapsto S$ the fact that the concrete semantics of P maps $\langle\theta, p\rangle$ to S.

5 Abstract Operations

Abstract Domains We assume the existence of three *cpo*s: *AS*, *ASS* and *ASSC*. Elements of *AS* are called abstract substitutions and denoted by β. Elements of *ASS* are called abstract sequences and denoted by B. Elements of *ASSC* are called abstract sequences with cut information and denoted by C. The meaning of these abstract objects is given through monotonic concretization functions: $Cc : AS \rightarrow CS$, $Cc : ASS \rightarrow CSS$ and $Cc : ASSC \rightarrow CSSC$. $CS = \wp(PS)$, *CSS* is the set of sets of substitution sequences which are upper-closed. *CSSC* is similarly defined but increasing chains only contain substitution sequences with identical cut information. *CS*, *CSS* and *CSSC* are ordered by inclusion. Each object O in *AS*, *ASS* and *ASSC* has a domain $dom(O)$ which is the common domain of all program substitutions in its concretization.

Organization Each abstract operation is motivated and specified by a consistency condition. Many of these operations are identical or simple generalizations of operations described in [16, 17], which were themselves inspired by [5]. Other are simple "conversion" operations between the three different domains. The newer operations are *CONC*, *AI-CUT*, *EXTGS* and they are explained in detail since they contain the main originality of our framework. Reference [4] proposes an implementation of these operations on a particular abstract domain.

Concatenation of Abstract Sequences: $CONC(\beta, C, B) = B'$. Let *pr* be a procedure of the form $c_1, \ldots, c_n$ $(n \geq 1)$. A *suffix* of *pr* is any sequence of clauses $c_i, \ldots, c_n$ $(1 \leq i \leq n)$. Operation *CONC* is used to "concatenate" (at the abstract level) the result C of a clause c_i with an abstract sequence B resulting from "concatenating" the results of $c_{i+1}, \ldots, c_n$ $(1 \leq i < n)$. It is assumed that all results are produced for the same abstract input substitution β. β is added as an extra parameter in order to improve the accuracy of the operation.

In order to express the consistency conditions for the operation *CONC*, "concatenation" of concrete sequences needs to be defined first. Consider two sequences S_1 and S_2 without cut information. S_1 stands for the result of c_i and S_2 stands for the (combined) result of $c_{i+1}, \ldots, c_n$.

If execution of c_i terminates, then suffix $c_{i+1}, \ldots, c_n$ is executed. Otherwise $c_{i+1}, \ldots, c_n$ is not executed. Therefore, the combined result $S_1 \Box S_2$ of $c_i, \ldots, c_n$ is defined by

$$\begin{array}{lcll} S_1 \Box S_2 & = & S_1 :: S_2 & \text{if } S_1 \text{ is finite (i.e. neither incomplete nor infinite),} \\ & & S_1 & \text{otherwise.} \end{array}$$

The definition can be extended to sequences with cut information. If no cut is executed in c_i (because c_i does not contain a cut or c_i fails or loops before reaching a cut), the previous reasoning applies. Otherwise, suffix $c_{i+1}, \ldots, c_n$ is not executed. In the first case, the result of c_i is $\langle S_1, nocut\rangle$, while, in the second case, the result is $\langle S_1, cut\rangle$. So, the combined result $\langle S_1, cf\rangle \Box S_2$ of $c_i, \ldots, c_n$ is defined by

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

Operation *CONC* performs the concatenation of abstract sequences, i.e. of descriptions of sets of sequences, and is defined as follows (recall that we note $CONC(\beta,C,B)=B'$)[4]:

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

Since β represents many different input substitutions, C and B may contain *incompatible* substitution sequences, i.e. sequences containing substitutions which are not all instances of the same input substitution. Concatenations of incompatible substitution sequences are removed by the last condition, since they do not correspond to any actual execution. ($\theta' \leq \theta$ means that θ' is more instantiated than θ.)[5]

Abstract Unification of two program variables: $AI\text{-}VARS(\beta) = B'$**.** This operation is similar to operation *AI-VAR* of [16, 17] but returns an abstract sequence instead of an abstract substitution. As the concrete unification may only fail or succeed (assuming an occur-check), $Cc(B')$ should only contain *finite* substitutions of length 0 or 1:

$$\left.\begin{array}{l} \theta \in Cc(\beta), \\ \sigma \in mgu(x_1\theta, x_2\theta) \end{array}\right\} \Rightarrow\ <\theta\sigma> \in Cc(B'); \qquad \left.\begin{array}{l} \theta \in Cc(\beta), \\ \emptyset = mgu(x_1\theta, x_2\theta) \end{array}\right\} \Rightarrow\ <> \in Cc(B').$$

Abstract Unification of a variable and a functor: $AI\text{-}FUNCS(\beta, f) = B'$**.** The operation is similar to the previous one: let $Smgu = mgu(x_1\theta, f(x_2, \ldots, x_n)\theta)$.

$$\left.\begin{array}{l} \theta \in Cc(\beta), \\ \sigma \in Smgu \end{array}\right\} \Rightarrow\ <\theta\sigma> \in Cc(B'); \qquad \left.\begin{array}{l} \theta \in Cc(\beta), \\ \emptyset = Smgu \end{array}\right\} \Rightarrow\ <> \in Cc(B').$$

[4] In order to enhance readability of the specifications, it is assumed that all free symbols are implicitely universally quantified and range over a domain which is "obvious" from the context. "," denotes conjunction.

[5] In the implementation B is only computed on demand, since it is not always needed.

Abstract Treatment of the Cut: $AI\text{-}CUT(C) = C'$. Let g be the sequence of literals before a cut (!) in a clause. Execution of g for a given input substitution θ either fails or loops without producing any result, or produce one or more results before failing, looping or producing results for ever. Execution of the goal g , ! also fails or loops without producing results in the first case but, in the second case, it produces exactly one result (the first result of g) and then stops. At the abstract level, C represents a set of substitution sequences produced by g, while C' represents the corresponding set of substitution sequences produced by g , ! . Clearly the sequences in $Cc(C')$ should be obtained by "cutting" the sequences in $Cc(C)$ after their first element if it is a substitution. Hence, the following specification:

$$\begin{array}{lcl} \langle <>, cf\rangle \in Cc(C) & \Rightarrow & \langle <>, cf\rangle \in Cc(C'); \\ \langle <\perp>, cf\rangle \in Cc(C) & \Rightarrow & \langle <\perp>, cf\rangle \in Cc(C'); \\ \langle <\theta>:: S, cf\rangle \in Cc(C) & \Rightarrow & \langle <\theta>, cut\rangle \in Cc(C'). \end{array}$$

We now turn to the projection and extension operations. The first and the third are the same as in our previous papers [16, 17] and we provide their specifications without additional explanations. The second one is a simple generalization of an existing one to sequences. The fourth one is a more complex generalization and we explain it in detail.

Extension at Clause entry: $EXTC(c, \beta) = \beta'$. Assume that β is an abstract substitution on $\{x_1, \ldots, x_n\}$ and c is a clause containing variables $\{x_1, \ldots, x_m\}$ $(m \geq n)$.

$$\left.\begin{array}{l} \theta \in Cc(\beta), \\ y_1, \ldots, y_{m-n} \text{ are distinct} \\ \text{standard variables,} \\ y_1, \ldots, y_{m-n} \notin codom(\theta) \end{array}\right\} \Rightarrow \{x_1/x_1\theta, \ldots, x_n/x_n\theta, x_{n+1}/y_1, \ldots, x_m/y_{m-n}\} \in Cc(\beta').$$

Restriction at Clause Exit: $RESTRC(c, C) = C'$. With the same notations as above, the execution of the body of c, for the input β', produces the abstract sequence with cut information C. Operation $RESTRC$ simply restricts C to the variables in $D = \{x_1, \ldots, x_n\}$: [6]

$$\langle <\theta_1, \ldots, \theta_i, \ldots>, cf\rangle \in Cc(C) \Rightarrow \langle <\theta_{1|D}, \ldots, \theta_{i|D}, \ldots>, cf\rangle \in Cc(C').$$

Restriction before a call: $RESTRG(l, \beta) = \beta'$. Assume that β is an abstract substitution on $D = \{x_1, \ldots, x_m\}$, and l is a literal $p(x_{i_1}, \ldots, x_{i_n})$ (or $x_{i_1} = x_{i_2}$ $(n = 2)$ or $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$) (or any other built-in using variables $x_{i_1}, \ldots, x_{i_n}$).

$$\theta \in Cc(\beta) \Rightarrow \{x_1/x_{i_1}\theta, \ldots, x_n/x_{i_n}\theta\} \in Cc(\beta').$$

Extension of the Result of a Call: $EXTGS(l, C, B) = C'$ This operation is rather complex and we first motivate it through the correspondence between the concrete and abstract executions. We assume the same notations as for $RESTRG$, that l occurs in the body of a clause c and that g is the sequence of literals before l in the body.

[6] The notation $\theta_{|D}$ should be obvious (otherwise see the appendix).

In the concrete semantics, execution of g for an input substitution θ produces a sequence (with cut information) $\langle S, cf\rangle$. Then l is executed for each substitution θ_i of S, producing a new sequence S_i for each θ_i. The "result" of g, l is the sequence $S_1 \square \ldots \square S_i \ldots$. [7]

At the abstract level, C stands for a set of possible S's while B stands for (a superset of) all corresponding S_i's. Because of the abstraction, the mapping between each S and its corresponding S_i's is lost as well as the ordering of the S_i's. Operation *EXTGS* has to reestablish this mapping as best as possible by a kind of backward unification.

Note that, in the above (concrete) concatenation, there can be infinitely many S_i's and the definition of $\square$ must be extended as follows:

$$\square_{k=1}^{0} S_k = <>; \quad \square_{k=1}^{i+1} S_k = (\square_{k=1}^{i} S_k) \square S_{i+1} \quad (i \geq 0); \quad \square_{k=1}^{\infty} S_k = \sqcup_{i=0}^{\infty}((\square_{k=1}^{i} S_k) \square < \perp >).$$

It can be verified that this definition fits the intuition in all cases. For instance, if one of the S_i is incomplete or infinite, subsequent sequences are ignored. $\square_{k=1}^{\infty} <> = < \perp >$ which expresses that the computation of an infinite number of sequences (albeit all empty) never terminates.

More technically, B is obtained by 1) extracting the substitution part β of C (the sequence structure is forgotten), 2) applying *RESTRG* to β, 3) executing procedure p with input β giving B. Therefore, B is an abstract sequence on $\{x_1, \ldots, x_n\}$ and we have to reexpress it on $\{x_{i_1}, \ldots, x_{i_n}\}$ while combining it with C. The precise specification is as follows. $NELEM(S)$ stands for the number of elements in S. $NELEM(S) = NSUBST(S) + 1$ if S is incomplete; in this case, we define $S_{NELEM(S)} = < \perp >$, by convention. Otherwise, $NELEM(S) = NSUBST(S)$.

$$\left.\begin{array}{l} \langle S, cf\rangle \in Cc(C), \\ (\forall k : 1 \leq k \leq NSUBST(S): \\ \theta_k \text{ is the } k\text{-th substitution of } S, \\ \theta'_k = \{x_1/x_{i_1}\theta_k, \ldots, x_n/x_{i_n}\theta_k\}, \\ S'_k \in Cc(B), \\ S'_k = < \theta'_k\sigma_{k,1}, \ldots, \theta'_k\sigma_{k,j}, \ldots >, \\ S_k = < \theta_k\sigma_{k,1}, \ldots, \theta_k\sigma_{k,j}, \ldots >) \end{array}\right\} \Rightarrow \langle \square_{k=1}^{NELEM(S)} S_k, cf\rangle \in Cc(C').$$

In order to prevent introduction of undesired variable sharing in the result, we can also specify that no substitution $\sigma_{k,j}$ introduces "new" variables already in $codom(\theta_k)$ but not in $codom(\theta'_k)$. Formally: $dom(\sigma_{k,j}) \subseteq codom(\theta'_k)$ and $(codom(\theta_k) \setminus codom(\theta'_k)) \cap codom(\sigma_{k,j}) = \emptyset \; \forall k, j$.
Finally, we need three less important conversion operations.

$SEQ(C) = B'$. This operation forgets the cut information in C. It is applied to the result of the last clause of a procedure before combining this with the result of the other clauses.

$$\langle S, cf\rangle \in Cc(C) \quad \Rightarrow \quad S \in Cc(B').$$

$SUBST(C) = \beta'$. 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*.

$$\left.\begin{array}{l} \langle S, cf\rangle \in Cc(C), \\ \theta \text{ is an element of } S \end{array}\right\} \Rightarrow \quad \theta \in Cc(\beta').$$

[7] Note that the execution of g is in fact interleaved with the executions of l in Prolog. However, in abstract interpretation, it is natural to assume that g is completed before starting executions of l because abstract executions always terminates.

EXT-NOCUT$(\beta) = C'$. The empty prefix of the body of a clause produces a one element sequence and contains no cut. This is expressed by the following specification:[8]

$$\theta \in Cc(\beta) \quad \Rightarrow \quad \langle <\theta>, nocut\rangle \in Cc(C').$$

Other built-ins. Built-ins such as *var*, *ground*, *functor*, arithmetic predicates, ... can be handled in our framework. In fact, any meta predicate can be dealt with except *assert* and *retract* because they modify the underlying program. We do not describe the abstract treatment of these built-ins for space reasons but they are taken into account by our implementation (see [4]).

6 Abstract Semantics

Sets of Abstract Tuples The abstract semantics of a program P is defined as a set of abstract tuples (β, p, B) where p is a predicate symbol of arity n occurring in P, $\beta \in AS$, $B \in ASS$ and $dom(\beta) = dom(B) = \{x_1, \ldots, x_n\}$. The *underlying domain UD* of the program is the set of all (β, p) such that $\beta \in AS$, $dom(\beta) = \{x_1, \ldots, x_n\}$ and p occurs in P. In fact, we only consider sets of abstract tuples which are functions from *UD* into *ASS* and we use both $B = sat(\beta, p)$ or $(\beta, p, B) \in sat$. We denote $SATT$ the set of all those sets.

Abstract Transformation This transformation is in the same spirit as the transformation proposed in [16]. The main difference is that (output) abstract substitutions are replaced by abstract sequences. Abstract operations are modified accordingly. For example, the semantics in [16] uses an operation *UNION* to collect clause results. This operation is now replaced by operation *CONC*. Two major simplifications with respect to the concrete semantics have been however introduced to handle literals more simply. Let g be a goal of the form g', l and C be the abstract sequence resulting from the execution of g'. First, the input abstract sequence C for l is "abstracted" to a single abstract substitution β'' approximating all substitutions in the concretization of C, i.e. the sequence structure of C is "lost". Second, the input and ouput sequences for l are combined in all possible way through a unique operation *EXTGS*. This simplification was shown to provide a good trade-off between accuracy and efficiency for the abstract domains considered in [4]. This trade-off could be reconsidered for more elaborate domains.

The abstract transformation is defined in terms of one function and one transformation given in figure 1. T is an auxiliary function for the definition of the transformation *TSAT*. T has arguments of the form $(\beta, cons, sat)$ where *cons* may be a predicate name p, a procedure or a suffix of a procedure (both denoted pr), a clause c or a goal g (i.e. the body or a prefix of the body of a clause). β is a substitution whose domain agrees with the particular *cons*. *sat* is a set of abstract tuples. The result of T is either an abstract sequence B (for p and pr) or an abstract sequence with cut information C (for c and g).

$T(\beta, p, sat)$ executes $p(x_1, \ldots, x_n)$ with input abstract substitution β by calling the function $T(\beta, pr, sat)$ that executes all clauses defining p on β. $T(\beta, c.pr', sat)$ concatenates the results produced by the first clause and by the rest of the procedure. $T(\beta, c, sat)$ executes a clause by extending the abstract substitution β to all variables in c, executing the body and restricting the result to the variables in the head. $T(\beta, g, sat)$ executes the body of a clause by considering each

[8] In [4], operations *EXTC* and *EXT-NOCUT* are combined into a single operation *EXTCS*.

$$TSAT(sat) = \{(\beta, p, B) : (\beta, p) \in UD \text{ and } B = T(\beta, p, sat)\}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{T(\beta, p, sat) = T(\beta, pr, sat)} \\
\textbf{where} & pr \text{ is the procedure defining } p &
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{T(\beta, pr, sat) = SEQ(C)} \\
\textbf{where} & C = T(\beta, c, sat) & \text{if } pr \text{ is } c
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{T(\beta, pr, sat) = CONC(\beta, C, B)} \\
\textbf{where} & B = T(\beta, pr', sat) & \\
 & C = T(\beta, c, sat) & \text{if } pr \text{ is } c.pr'
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{T(\beta, c, sat) = RESTRC(c, C)} \\
\textbf{where} & C = T(EXTC(c, \beta), g, sat) \\
 & g \text{ is the body of } c
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{T(\beta, <>, sat) = C} \\
\textbf{where} & C = EXT\text{-}NOCUT(\beta)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{T(\beta, (g, !), sat) = AI\text{-}CUT(C)} \\
\textbf{where} & C = T(\beta, g, sat)
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{T(\beta, (g, l), sat) = EXTGS(l, C, B)} \\
\textbf{where} & B = & AI\text{-}VARS(\beta') & \text{if } l \text{ is } x_i = x_j \\
 & & AI_FUNCS(\beta', f) & \text{if } l \text{ is } x_i = f(\ldots) \\
 & & sat(\beta', p) & \text{if } l \text{ is } p(\ldots) \\
 & \multicolumn{3}{l}{\beta' = RESTRG(l, \beta'')} \\
 & \multicolumn{3}{l}{\beta'' = SUBST(C)} \\
 & \multicolumn{3}{l}{C = T(\beta, g, sat).}
\end{array}$$

Figure 1: The Abstract Transformation

literal in turn. The empty prefix of the body produces a one element abstract sequence with the information that no cut has been executed so far. When the next literal to execute is a cut, operation *AI-CUT* is executed. Otherwise the next literal l is executed with input β'' that approximates all substitutions in the concretization of C. Operation *RESTRG* expresses β'' in terms of the formal parameters $x_1, \ldots, x_n$ of l. If l is a procedure call then only a lookup in *sat* is performed, otherwise either operation *AI-VARS* or *AI-FUNCS* is executed. Operation *EXTGS* is performed after each call in order to obtain the result of the full goal. *TSAT* is a transformation from *SATT* to *SATT*.

Abstract Semantics Transformation *TSAT* can be shown monotonic if the abstract operations are. However monotonicity is not an essential requirement for our framework because we do not define the abstract semantics as the least fixpoint of *TSAT* which is not consistent in general as explained in section 2. In order to get a consistent *sat*, the transformation is applied to *sat*'s which are *pre-consistent.*

Definition 1 [Pre-Consistency] A set of abstract tuples *sat* is pre-consistent iff, for each abstract tuple $(\beta, p, B) \in sat$, $\langle\theta, p\rangle \longmapsto S$ with $\theta \in Cc(\beta)$ implies that there exists $S' \sqsubseteq S$ such that $S' \in Cc(B)$.

When there exists an abstract sequence $B_{<\perp>}$ such that $<\perp> \in Cc(B_{<\perp>})$, it is easy to define a first pre-consistent set of abstract tuples, since $<\perp> \sqsubseteq S$ for all S. Moreover, applying transformation *TSAT* to pre-consistent *sats* gives other pre-consistent *sats* which are better lower approximations of the concrete outputs by consistency of the abstract operations. Finally, a post-fixpoint is reached to obtain consistency. The abstract semantics can thus be formalized as any pre-consistent postfixpoint of the abstract transformation. Formally, the results whose proofs are summarized in appendix and follows the informal reasoning of Section 2 can be stated as follows.

Lemma 2 Let *sat* be a pre-consistent set of abstract tuples. Then *TSAT(sat)* is pre-consistent.

Theorem 3 [Consistency of the Abstract Semantics] Let *sat* be a pre-consistent set of abstract tuples such that *TSAT(sat)* $\leq$ *sat*. Then *sat* is consistent. That is: let p be a predicate symbol, θ be a substitution, β be an abstract substitution, S be a substitution sequence. We have

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

7 The Generic Abstract Interpretation Algorithm

We now discuss how postfixpoints of the abstract transformation can be computed. The key idea is that a postfixpoint can be computed by a generalization of existing generic abstract interpretation algorithms [5, 21, 16, 17]. We focus on the generalizations and their justifications here. See [4] for a description of the algorithm. The key generalization in the algorithm is the use of a more general form of widening, called E-widening, when updating the set of abstract tuples with a new result.

Definition 4 [E-widening] Let A be an abstract domain and B_i, B'_i be elements of A. A *E-widening* is an operation $\nabla : A \times A \to A$ which, given the sequences $B_1, \ldots, B_i, \ldots$ and $B'_0, \ldots, B'_i, \ldots$ such that $B'_{i+1} = B_{i+1} \nabla B'_i$ $(i \geq 0)$, satisfies

1. $B'_i \geq B_i \quad (i \geq 1)$;
2. There is a $j \geq 0$ such that all B'_i with $j \leq i$ are equal.

The E-widening is used as follows in the algorithm. Given an input pair (β, p), the output abstract sequence is computed by generating two sequences $B_1, \ldots, B_i, \ldots$ and $B'_0, \ldots, B'_i, \ldots$ as follows:

1. $B'_0 = B_{<\perp>}$ is stored in the initial *sat* as the output for (β, p);
2. B_i results from the *i-th* abstract execution of procedure p for abstract input β;
3. $B'_i = B_i \nabla B'_{i-1}$ is stored in the current *sat* after the *i-th* abstract execution of procedure p;
4. reexecution stops when $B_{i+1} \leq B'_i$.

Termination of the algorithm is guaranteed because all B'_i must be equal for all i greater than some j. Hence, since $B'_j = B'_{j+1}$ and $B'_{j+1} \geq B_{j+1}$, we have $B_{j+1} \leq B'_j$. Consistency of the result is guaranteed because each B'_i is pre-consistent and the algorithm terminates with a postfixpoint. Pre-consistency of the B'_i follows from $B'_i \geq B_i$ and the pre-consistency of B_i due to Lemma 2. An example of E-widening is defined in [4].

8 Related Work

Perhaps the closest related work is the work of File and Rossi [12], who describe an extension of the framework in [6], where an *OLDT* abstract tree is adorned with information about sure success or failure of the goals. The information is then used in the cut operation to prune the *OLDT*-tree whenever the cut is reached in all corresponding executions. Sure success is modelled in our framework by abstract sequences having only non-empty sequences in their concretizations. Sure failure is modelled by the empty sequence. There are several differences between the two frameworks. At the theoretical level, their framework can be characterized as operational and non-compositional while ours is compositional and based on the fixpoint approach. At the algorithmic level, there are two main differences. The first is best described on a goal *p(X), !*. Whenever *p(X)* surely succeeds, their framework stops after generating the first "sure" solution, while ours computes the entire abstract sequence for *p(X)* and then cuts it to maintain at most one solution. Our algorithm may thus imply some redundant work. However, if *p(X)* is used in several contexts, their algorithm should recognize this situation and expand the *OLDT*-tree further. The second difference comes from the fact that our framework may deduce sure success even though the success branch may be unknown, while it is not clear how to obtain this result in their approach. Finally, our approach has been shown computationally tractable in [4]. At the time of writing, no experimental result have been reported on their approach.

The work of Barbuti et al. [2] also aims at modelling Prolog control. The main difference between their work and ours is that their framework is intended to *use* control information deduced from outside, while our framework both *deduces* and *uses* control information inside the framework.

Our framework is usually not able to compute precise termination information (except for non recursive procedures) since this is inherently outside the scope of computational induction, the basis of the abstract interpretation approach followed here. However some applications such as cardinality analysis [4] could be improved by allowing the framework to use termination information from the outside as in [2]. Furthermore, our framework is able to provide precise information about non-termination. This is an important consequence of the fact that the limit of an infinite chain of incomplete substitution sequences is either an incomplete substitution sequence or an infinite sequence. Precise information about non termination may improve other analyses significantly in the case of incorrect programs, making it useful for static debugging.

9 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 postfixpoint of the abstract transformation. Abstract interpretation algorithms need upper-closed domains and a special widening operator to compute the semantics. This approach overcomes some of the limitations of existing frameworks. In particular, it broadens the applicability of the abstract interpretation approach to new analyses and improves the precision of existing analyses. Its practicability has been demonstrated in the companion paper [4].

References

[1] S. Abramski and C. Hankin. An introduction to abstract interpretation. In S. Abramski and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1, pages 9–31. Ellis Horwood Limited, 1987.

[2] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog control. In *Proceedings of POPL'92*, pages 95–104. ACM Press, 1992.

[3] M. Baudinet. Proving Termination Properties of Prolog Programs: A Semantic Approach. In *Proc. Third IEEE Symp. on Logic In Computer Science*, pages 336–347. IEEE, 1988.

[4] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality Analysis of Prolog. Technical report, Department of Computer Science, Brown University, March 1994. (Submitted to ILPS'94).

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

[6] P. Codognet and G. Filé. Computations, abstractions and constraints in logic programs. In *Proceedings of (ICCL'92)*, Oakland, U.S.A., April 1992.

[7] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type analysis of prolog using type graphs. In *Proceedings of (PLDI'94)*, Orlando, Florida, June 1994.

[8] 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 (POPL'77)*.

[9] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In ICLP-93, Budapest (Hungary), June 1993.

[10] A. de Bruin and E. de Vink. Continuation Semantics for Prolog with cut. In *Proc. TAPSOFT'89*, Lecture Notes in Computer Science, pages 178–192, Berlin, 1989. Springer-Verlag.

[11] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. (5(1)):61–91, 1988.

[12] G. Filé and S. Rossi. Static analysis of Prolog with cut. In *Proc. of LPAR'93*.

[13] R. Giacobazzi. Detecting Determinate Computations by Bottom-up Abstract Interpretation. In *ESOP'92*, pages 167–181, 1992.

[14] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(4), 1992.

[15] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In Sten-Åke Tarnlund, editor, *Proceedings of ICLP'84*, pages 281–288.

[16] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In *Proceedings of (ICLP'91)*.

[17] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *(TOPLAS)*, January 1994.

[18] J.W. Lloyd. *Foundations of Logic Programming.* Springer Series: Symbolic Computation–Artificial Intelligence. Springer-Velag, second, extended edition, 1987.

[19] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. Ritter, editor, *Information Processing'89*, pages 601–606, San Fransisco, California, 1989.

[20] C.S. Mellish. Abstract interpretation of Prolog programs. In *Abstract Interpretation of Declarative Languages*, chapter 8, pages 181–198. Ellis Horwood Limited, 1987.

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

[22] D. Sahlin. Determinacy Analysis for Full Prolog. In *PEPM'91*, 1991.

[23] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Inc., 1988.

[24] A. Taylor. LIPS on MIPS: Results from a Prolog Compiler for a RISC. In *Proceedings of (ICLP'90)*, Jerusalem, Israel, June 1990. MIT Press.

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

∅

A The Concrete Semantics

A.1 Introduction

This appendix presents a fixpoint semantics for (normalized) definite logic programs with cut which can be proved equivalent to Prolog "standard" semantics (i.e. SLD-resolution with the left-most selection rule and the depht-first search strategy). The "raison d'être" of this instrumental semantics is the proof of consistency of the abstract semantics. The semantics is not aimed at replacing existing proposals such as [10, 11, 15] but these proposals are too far from our abstract semantics to provide a convenient basis for a consistency proof.

Technically, the concrete semantics is a denotational semantics, since it is compositional and uses a fixpoint construction over a complete partial order $SCTT$ built on the relation $\sqsubseteq$ on substitution sequences. However, our definition uses (pseudo) transition rules to highlight the similarities and differences with operational semantics for logic programs which do not take into account the search rule nor the cut. As a result, the concrete semantics "looks" very much like a simple SOS semantics but it is not operational because its fixpoint construction uses the ordering on $SCTT$ instead of set inclusion. Defining an equivalent operational semantics in SOS style would require either to complicate the semantic objects (to take into account the choice points, backtracking and so on) or to introduce an infinite hierarchy of transition rules corresponding to the different iterates of the Kleene's sequence leading to the fixpoint.

A.2 Basic definitions

We complement here the set of definitions of the main paper. Let θ be a substitution and $D \subseteq dom(\theta)$. The *restriction* of θ to D, denoted $\theta_{|D}$, is the substitution θ' such that $dom(\theta') = D$ and $x\theta = x\theta'$ for all $x \in D$. The notion of *free variable* is non-standard to avoid clashes between variables during renaming. A free variable is represented by a binding to a standard variable that appears nowhere else. We use $mgu(t_1, t_2)$ to denote the set of most general unifiers of t_1 and t_2.

Let $S_{i \in \mathbf{N}}$ be an increasing chain of substitution sequences, i.e. $S_0 \sqsubseteq \ldots \sqsubseteq S_i \sqsubseteq \ldots$. The least upper bound sequence $\sqcup_{i=0}^{\infty} S_i$ can be constructed as follows:

$$\sqcup_{i=0}^{\infty} S_i = \begin{cases} S_k & \text{if } \exists k \in \mathbf{N} \text{ such that } S_i = S_k \ (\forall i \geq k) \\ <\theta_1, \ldots \theta_i, \ldots> & \text{if } \forall i \in \mathbf{N} \exists k, S'_k : S_k = <\theta_1, \ldots, \theta_i >:: S'_k. \end{cases}$$

A.3 Concrete Operations

We define here some operations which are used by the concrete semantics. It can be observed that all operations return a set of program substitutions (resp. substitution sequences) although the result is conceptually a single substitution (resp. substitution sequence) in general. When a set contains many elements they are equivalent up to renaming. For brevity we omit some obvious preconditions in the definitions.

Extension and Restriction for a Clause The $EXTC$ operation extends a substitution on variables on the head of a clause to all variables in the clause. Let c be a clause, $var(c) = D$ and $var(head(c)) = D'$. The $RESTRC$ operation restricts a substitutions sequence on all variables in a clause to the variables in the head.

$$
\begin{array}{lcl}
EXTC(c,\theta) & = & \{\ \theta' :\ dom(\theta') = D,\ \theta'_{D'} = \theta \text{ and } \forall x \in D \setminus D',\ x \text{ is free in } \theta'\ \} \\
RESTRC(c,S) & = & \{\ S_{|D'}\ \}
\end{array}
$$

Restriction and Extension for a Call The *RESTRG* operation expresses a substitution θ, on the parameters $x_{i_1},\ldots,x_{i_n}$ of a call l, in terms of its formal parameters $x_1,\ldots,x_n$. The *EXTG* operation extends a substitution θ with a substitutions sequence S representing the result of executing a call l on θ.

$$
\begin{array}{lcl}
RESTRG(l,\theta) & = & \{\ \theta' : dom(\theta') = \{x_1,\ldots,x_n\},\ and\ x_k\theta' = x_{i_k}\theta\ (1 \leq k \leq n)\ \} \\
EXTG(l,\theta,S) & = & \{< \theta\sigma_1,\ldots,\theta\sigma_i,\ldots >: \exists\theta' \in RESTRG(l,\theta) \text{ such that } S =< \theta'\sigma_1,\ldots,\theta'\sigma_i,\ldots > \\
 & & \text{and } dom(\sigma_i) \subseteq codom(\theta') \text{ and } (codom(\theta)\backslash codom(\theta')) \cap codom(\sigma_i) = \emptyset\ \forall i\ \}
\end{array}
$$

Unification Operations Operation *AI-VAR* unifies $x_1\theta$ with $x_2\theta$. Operation *AI-FUNC* unifies $x_1\theta$ with $f(x_2,\ldots,x_n)\theta$. Observe that the operations do not specify the parameters x_1, x_2 (resp. $x_1, x_2, \ldots, x_n$) as arguments. This is because operation *RESTRG* is applied before.

$$
\begin{array}{lcll}
AI\text{-}VAR(\theta) & = & \{\ <>\ \} & \text{if } \emptyset = mgu(x_1\theta, x_2\theta) \\
 & & \{< \theta\sigma >: \sigma \in mgu(x_1\theta, x_2\theta)\} & \text{otherwise} \\
AI\text{-}FUNC(\theta, f) & = & \{\ <>\ \} & \text{if } \emptyset = mgu(x_1\theta, f(x_2,\ldots,x_n)\theta) \\
 & & \{< \theta\sigma >: \sigma \in mgu(x_1\theta, f(x_2,\ldots,x_n)\theta)\} & \text{otherwise}
\end{array}
$$

A.4 The Concrete Semantic Domain *SCTT*

We define the concrete semantic domain *SCTT* with respect to a fixed underlying program P. The *concrete underlying domain CUD* is the set of pairs $\langle\theta, p\rangle$ such that θ is a program substitution, p is a n-ary predicate of P and $dom(\theta) = \{x_1,\ldots,x_n\}$. A *set of concrete tuples* is a total function $\longmapsto: CUD \rightarrow PSS$ such that each pair $\langle\theta, p\rangle$ is mapped to a substitution sequence S where $dom(\theta) = dom(S)$. This is denoted by $\langle\theta, p\rangle \longmapsto S$. (In fact, S should be defined up to a renaming of the standard variables occurring in it but we ignore this technicality, for the sake of simplicity.)

By definition, *SCTT* is the set of all sets of concrete tuples. Its minimal element is $\longmapsto_\perp$ such that $\langle\theta, p\rangle \longmapsto_\perp < \perp >$ fot all $\langle\theta, p\rangle$. The ordering on *SCTT* is defined by

$$
\longmapsto \sqsubseteq \longmapsto' \qquad \text{iff} \qquad \forall\langle\theta, p\rangle \in CUD : \langle\theta, p\rangle \longmapsto S \text{ and } \langle\theta, p\rangle \longmapsto' S' \ \Rightarrow\ S \sqsubseteq S'.
$$

It is easy to show that *SCTT* is a *cpo* for this ordering.

A.5 Auxiliary Semantic Rules

In order to define the concrete semantics as the least fixpoint of a continuous transformation of *SCTT*, we first introduce a set of semantic rules which extend a set of concrete tuples $\longmapsto$ to a function from *ECUD* to $PSS \cup (PSS \times \{cut, nocut\})$. The *extended concrete underlying domain ECUD* consists of the pairs of the form $\langle\theta, p\rangle$, $\langle\theta, c\rangle$, $\langle\theta, g\rangle$, $\langle\theta, pr\rangle$, $\langle\theta, l\rangle$, where p, c, g, pr, l are respectively a predicate symbol, a clause, the body or a prefix of the body of a clause, a procedure or a suffix of a procedure, and (an occurrence of) a literal in the body of a clause (for the program P). The semantic rules specify the result of executing c, g, pr, l for the input p, *assuming that* $\longmapsto$: *CUD* $\rightarrow$ *PSS is used as an oracle* to solve the procedure calls.

The rules are given and explained below. The derived function is also denoted $\longmapsto$.

Execution of a Body The empty prefix produces a one element sequence (the cut is not executed). A cut is executed in a clause iff it follows a sequence of literals producing at least one result. Then it reduces this sequence to its first element. Otherwise failure or non termination occurs. Execution of other literals is more complicated. The result of g, l can be "computed" as follows. First, the result S of g is computed. Then literal l is executed with each substitution θ_k in S (restriction and renaming are handled by $RESTRG$). Each resulting sequence S'_k is extended wrt θ_k. Finally, all extended sequences are concatenated.

$$\frac{g \ ::= \ <>}{\langle \theta, g \rangle \longmapsto \langle <\theta>, nocut \rangle} \qquad \frac{\begin{array}{l} g \ ::= \ g', ! \\ \langle \theta, g' \rangle \longmapsto \langle S, cf \rangle \\ S \in \{<>, <\perp>\} \end{array}}{\langle \theta, g \rangle \longmapsto \langle S, cf \rangle} \qquad \frac{\begin{array}{l} g \ ::= \ g', ! \\ \langle \theta, g' \rangle \longmapsto \langle S, cf \rangle \\ S = <\theta> :: S' \end{array}}{\langle \theta, g \rangle \longmapsto \langle <\theta>, cut \rangle}$$

$$\frac{\begin{array}{l} g \ ::= \ g', l \\ l \ ::= \ x_{i_1} = x_{i_2} \\ \langle \theta, g' \rangle \longmapsto \langle S, cf \rangle \\ S = <\theta_1, \ldots, \theta_k, \ldots> \\ \theta'_k \in RESTRG(l, \theta_k) \\ S'_k \in AI\text{-}VAR(\theta'_k) \\ S_k \in EXTG(l, \theta_k, S'_k) \end{array}}{\langle \theta, g \rangle \longmapsto \langle \Box_{k=1}^{\text{NELEM}(S)} S_k, cf \rangle} \qquad \frac{\begin{array}{l} g \ ::= \ g', l \\ l \ ::= \ x_{i_1} = f(x_{i_2}, \ldots, x_{i_n}) \\ \langle \theta, g' \rangle \longmapsto \langle S, cf \rangle \\ S = <\theta_1, \ldots, \theta_k, \ldots> \\ \theta'_k \in RESTRG(l, \theta_k) \\ S'_k \in AI\text{-}FUNC(\theta'_k, f) \\ S_k \in EXTG(l, \theta_k, S'_k) \end{array}}{\langle \theta, g \rangle \longmapsto \langle \Box_{k=1}^{\text{NELEM}(S)} S_k, cf \rangle} \qquad \frac{\begin{array}{l} g \ ::= \ g', l \\ l \ ::= \ p(x_{i_1}, \ldots, x_{i_n}) \\ \langle \theta, g' \rangle \longmapsto \langle S, cf \rangle \\ S = <\theta_1, \ldots, \theta_k, \ldots> \\ \theta'_k \in RESTRG(l, \theta_k) \\ \langle \theta'_k, p \rangle \longmapsto S'_k \\ S_k \in EXTG(l, \theta_k, S'_k) \end{array}}{\langle \theta, g \rangle \longmapsto \langle \Box_{k=1}^{\text{NELEM}(S)} S_k, cf \rangle}$$

Procedure and Clause Execution If a cut is executed in the first clause of a procedure, other clauses are not executed. Otherwise the result of the procedure (for input θ) is the concatenation of the sequences produced by the first clause and by the rest of the procedure. Non termination of the first clause is correctly handled due to the definition of $\Box$.

Executing a clause c with input θ amounts to extending θ to all variables in c, executing the body of c, and restricting the sequence of results to the variables in the head. The cut is executed in the clause iff it is executed in its body.

$$\frac{\begin{array}{l} pr \ ::= \ c \\ \langle \theta, c \rangle \longmapsto \langle S, cf \rangle \end{array}}{\langle \theta, pr \rangle \longmapsto S} \qquad \frac{\begin{array}{l} pr \ ::= \ c.pr' \\ \langle \theta, c \rangle \longmapsto \langle S, cut \rangle \end{array}}{\langle \theta, pr \rangle \longmapsto S} \qquad \frac{\begin{array}{l} pr \ ::= \ c.pr' \\ \langle \theta, c \rangle \longmapsto \langle S, nocut \rangle \\ \langle \theta, pr' \rangle \longmapsto S' \end{array}}{\langle \theta, pr \rangle \longmapsto S \Box S'} \qquad \frac{\begin{array}{l} c \ ::= \ p(x_1, \ldots, x_n) \ \leftarrow \ g \\ \theta_1 \in EXTC(c, \theta) \\ \langle \theta_1, g \rangle \longmapsto \langle S', cf \rangle \\ S \in RESTRC(c, S') \end{array}}{\langle \theta, c \rangle \longmapsto \langle S, cf \rangle}$$

A.6 The Concrete Semantics

We first define a semantic transformation: $TSCT : SCTT \rightarrow SCTT$, by the following rule. Note that $\langle \theta, pr \rangle \longmapsto S$ is defined by means of the previous rules which use $\longmapsto$ as an oracle (a base case).

$$\frac{\begin{array}{c} pr \text{ defines } p \text{ in } P \\ \langle \theta, pr \rangle \longmapsto S \end{array}}{\langle \theta, p \rangle \overset{TSCT}{\longmapsto} S}$$

Theorem 5 Transformation $TSCT : SCTT \rightarrow SCTT$ is monotonic and continuous.

Proof It can be shown that all basic operations used by the semantic rules (e.g. $\Box$) are monotonic and continuous. The theorem follows. $\Box$

Definition 6 [Concrete Semantics] By definition, the concrete semantics of P (the underlying program) is $lfp(TSCT)$. We note it $\longmapsto$ in the sequel.

Theorem 7 [Correctness of the Concrete Semantics] Let $p(t_1, \ldots, t_n)$ be an initial goal where p is the name of a n-ary procedure of the underlying program P. Consider the sequence $\sigma_1, \ldots, \sigma_i, \ldots$ of computed answer substitution produced for $p(t_1, \ldots, t_n)$ wrt P using *SLD*-resolution, the leftmost selection rule, the depht-first search strategy and the usual meaning of the cut. Let $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$. Define $S =< \theta\sigma_1, \ldots, \theta\sigma_m >$, if the execution terminates after producing m results; $S =< \theta\sigma_1, \ldots, \theta\sigma_m, \perp >$, if the execution produces m computed answer substitutions and then enters into an infinite loop; $S =< \theta\sigma_1, \ldots, \theta\sigma_i, \ldots > (i \in \mathbf{N})$, if the execution produces an infinite number of computed answer substitutions. We have:

$$\langle \theta, p \rangle \longmapsto S.$$

Proof [Sketch] Let us note $\overset{i}{\longmapsto}$ the i-th iterate of the Kleene's sequence leading to $\longmapsto$. We can define a sequence T_i of *SLD*-trees obtained by limiting the depth of procedure unfolding up to i and pasting an (artificial) infinite branch on each unfolded goal. Then the sequence S_i corresponding to the sequence of computed answer substitutions for T_i is such that: $\langle \theta, p \rangle \overset{i}{\longmapsto} S_i$. The result follows by continuity. $\Box$

B Correctness of the Abstract Semantics

We use the notations and definitions of the main paper.

Lemma 8 Let sat be a set of abstract tuples such that sat is pre-consistent and is a postfixpoint of $TSAT$, i.e. $TSAT(sat) \leq sat$. Let $(\beta, p) \in UD$ and $\theta \in Cc(\beta)$. Let S such that $\langle \theta, p \rangle \longmapsto S$ and S_i such that $\langle \theta, p \rangle \overset{i}{\longmapsto} S_i$ $(0 \leq i)$. Then for all $i \in \mathbf{N}$, there exists $S'_i \in PSS$ such that $S_i \sqsubseteq S'_i \sqsubseteq S$ and $S'_i \in Cc(sat(\beta, p))$.

Proof [By induction on i.] The result is straightforward for $i = 0$ since sat is pre-consistent. Suppose $i > 0$. By induction hypothesis, for all $(\beta, p) \in UD$ and $\theta \in Cc(\beta)$, there exists $S'_{i-1} \in PSS$ such that

$$S_{i-1} \sqsubseteq S'_{i-1} \sqsubseteq S \quad \text{and} \quad S'_{i-1} \in Cc(sat(\beta, p)).$$

This defines a set of concrete tuples $\overset{i-1'}{\longmapsto}$ such that

$$\overset{i-1}{\longmapsto} \sqsubseteq \overset{i-1'}{\longmapsto} \sqsubseteq \longmapsto .$$

By monotonicity of the concrete transformation,

$$\overset{i}{\longmapsto} \sqsubseteq \overset{\overset{TSCT}{i-1}\,'}{\longmapsto} \sqsubseteq \longmapsto .$$

By consistency of the abstract operations, for all $(\beta, p) \in UD$, $\theta \in Cc(\beta)$ and S'_i such that

$$\langle \theta, p \rangle \overset{\overset{TSCT}{i-1}\,'}{\longmapsto} S'_i,$$

we have $S'_i \in Cc(sat'(\beta, p))$ where $sat' = TSAT(sat)$. But, since sat is a post-fixpoint, $S'_i \in Cc(sat'(\beta, p))$ implies $S'_i \in Cc(sat(\beta, p))$. $\square$

Theorem 9 [Correctness] Let sat be a set of abstract tuples. If sat is a postfixpoint of $TSAT$, i.e. $TSAT(sat) \leq sat$, and is pre-consistent, then it is consistent, i.e., for all $(\beta, p) \in UD$ and for all $\theta \in Cc(\beta)$, $\langle \theta, p \rangle \longmapsto S$ implies that $S \in Cc(sat(\beta, p))$.

Proof Assume fixed $(\beta, p) \in UD$ and $\theta \in Cc(\beta)$. Consider the corresponding sequences S'_i in the lemma. From these S'_i, we construct an increasing chain: $S''_0 \sqsubseteq \ldots S''_i \sqsubseteq$ by defining $S''_0 =< \perp >$ and $S''_i = S'_i \sqcup S''_{i-1}$ $(i > 0)$. The least upper bound $S'_i \sqcup S''_{i-1}$ is defined because $S'_i, S''_{i-1} \sqsubseteq S$. Moreover, $S'_i \sqcup S''_{i-1}$ is either equal to S'_i or to S''_{i-1} which implies that it belongs to $Cc(sat(\beta, p))$. Clearly, $S_i \sqsubseteq S''_i \sqsubseteq S$ for all i so that $S = \bigsqcup_{i=0}^{\infty} S_i \sqsubseteq \bigsqcup_{i=1}^{\infty} S''_i \sqsubseteq S$. Therefore, since $Cc(sat(\beta, p))$ is upper-complete, $S \in Cc(sat(\beta, p))$. $\square$