MASTER IN COMPUTER SCIENCE

Abstract Interpretation of full Prolog

Chabot, Francois; Joucken, Patrick

Award date:
1995

Awarding institution:
University of Namur

Link to publication

Abstract Interpretation of full Prolog

François CHABOT

Patrick JOUCKEN

Promoteur: Baudouin LE CHARLIER

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique

Facultés Universitaires
Notre-Dame de la Paix
Institut d'Informatique
N a m u r

FM B16/1995/4

Abstract Interpretation of full Prolog

Abstract

Abstract Interpretation is a general methodology for building tools performing static analysis of programs.

We present in this report a new abstract interpretation framework for *full* Prolog — i.e. taking into account the system predicates `asserta`, `assertz` and `retract`. The Prolog programs can thus modify themselves dynamically, what is at the origin of new difficulties. We expose the methodological aspects of our solution which consists of a general approach widely applicable and that was shown very effective for quite large logic programs. The new framework still allows to integrate traditional analyses (e.g. mode and type), determinacy analysis, functionality analysis and cardinality analysis.

To achieve it, we propose a new approach where the abstract domain captures information about not only modes, types, number of solutions, cut and termination, but also about the '*dynamic context*' upon which a literal is executed.

Abrégé

L'interprétation abstraite constitue une méthode générale pour construire des outils qui analyse statiquement des programmes.

Dans ce rapport, nous présentons un nouveau système d'interprétation abstraite pour l'entièreté du Prolog, c'est-à-dire tenant compte des prédicats systèmes `asserta`, `assertz` et `retract`. Tout programme Prolog est alors à même de se modifier durant son exécution, ce qui est à l'origine de nouvelles difficultés. Nous présentons les aspects méthodologiques de notre solution, consistant en une approche générale largement applicable et ayant prouvé son efficacité d'analyse sur des programmes logiques relativement long. Le nouveau modèle permet toujours d'intégrer les traditionnelles analyses (telles que l'analyse du mode et du type), le déterminisme, l'analyse fonctionnelle et l'analyse de la cardinalité.

Pour y parvenir, nous proposons une nouvelle approche où le domaine abstrait synthétise non seulement des propriétés concernant les modes, les types, le nombre de solutions, le cut et la terminaison mais aussi des informations concernant le '*contexte dynamique*' dans lequel s'exécute les littéraux.

Contents

Introduction

Abstract interpretation is a general methodology to obtain, in a systematic way, tools to analyze programs statically (at compile time). The basic idea behind abstract interpretation is to approximate (usually undecidable) properties by using an abstract domain instead of the actual domain of computation. As a consequence, the program as a whole can be given an approximated meaning, hopefully capturing interesting properties while leaving out irrelevant details as much as possible.

Abstract interpretation was originally designed by P. and R. Cousot as a *general methodology* for building static analyses of programs. The original ideas have been subsequently developed by many researchers and mainly applied to declarative languages, more amenable to optimizations.

Recently, much attention has been devoted to the abstract interpretation of logic programs and abstract interpretation of Prolog is a very active field of research. This effort is motivated by the need of optimization in logic programming compilers to be competitive with procedural languages and the declarative nature of the languages which makes them more amenable to static analysis. Considerable progress has been realized in this area in terms of the frameworks, the algorithms, the abstract domains and the implementations. Recent results indicate that abstract interpretation can be competitive with specialized data flow algorithms and could be integrated in industrial compilers. However relatively little attention has been devoted to the abstract interpretation of *full* Prolog with cut, built-ins and the system predicates `asserta`, `assertz` and `retract`.

The purpose of this report is to describe a new abstract interpretation framework, and its implementation, suited for the system predicates `asserta`, `assertz` and `retract`. The key idea of the new framework is to split the set of predicates of a logic program in a *static* part and a *dynamic* part whereas the definition of dynamic predicates can be modified dynamically, i.e. during the execution of the program. Clauses for a dynamic predicate can be added or removed using the system predicates `asserta`, `assertz` and `retract` respectively.

The aim of the new framework is not to allow a new kind of analysis nor to increase accuracy, but to extend the class of programs that can be analyzed by the abstract interpretation algorithm to programs using the system predicates `asserta`, `assertz` and `retract`. To our knowledge, there is no existing framework that can deal with this class of programs. Thus, the challenge was to design a new framework not only suited for treating *full* Prolog, but also providing a good tradeoff between accuracy and performance of the analysis. We consider having fulfilled the challenge.

This report is divided in two parts. In the three chapters of the first part, we try to give to the reader the background necessary to understand the rest of the report. Chapter one makes a comprehensive recall of the computational model of Prolog and details the operational semantics of system predicates `asserta`, `assertz` and `retract` to fix the thinking background. Chapter two explains the main concepts of abstract interpretation and chapter three exposes a general methodology for the design of abstract interpretation frameworks.

The three other chapters of the second part present an evolutionary approach leading the reader through successive refinements from a quite simple abstract interpretation framework suited for mode and type analysis of *pure* Prolog to a considerably more complex framework allowing cardinality analysis for *full* Prolog, i.e. Prolog programs including any kind of system predicates. Chapter four introduces the first generic abstract interpretation algorithm capable to collect information about pure Prolog programs. In chapter five, we explain an enhanced version of the first one suited to deal with the Prolog SLD-Resolution and cut primitive. Finally, in chapter six, we detail the last version of the algorithm which can handle the system predicates `asserta`, `assertz` and `retract`. The second part thus presents, one at a time, the frameworks and the algorithms getting more and more refined. The aim is to get a framework with its algorithm capable to analyze *full Prolog* programs according to the SLD-Resolution.

Remerciements

Nous tenons tout particulièrement à remercier Baudouin Le Charlier pour nous avoir permis, grâce aux contacts qu'il a établis, d'effectuer notre stage de maîtrise à Brown University, ainsi que pour ses conseils utiles à la rédaction du mémoire. Nous remercions *également* Pascal Van Hentenryck pour nous avoir suivi durant notre visite à Brown et pour la motivation qu'il a su nous insuffler.

Part

I

Background

The chapters of this first part contain a comprehensive introduction to logic programming, and Prolog in particular. They also develop the prerequisite concepts necessary to the abstract interpretation. We mainly insist on the methodology used to build the later frameworks but also on the theoretical, mathematical aspects. We wanted this first introduction part to be as comprehensive as possible. Therefore, all the aspects detailed are done almost independently of the others, in order to be as clear and complete as possible.

Chapter

1

Prolog Computational Model

« Programming in Prolog opens the mind to a new way of looking at computing. There is a change of perspective every Prolog programmer experiences when first getting to know the language. »

D.H.D. Warren

As the abstract interpretation algorithm can process any Prolog program we believe a comprehensive recall of the computational model of Prolog could be necessary to fix the thinking background. Moreover, it is justified since few books give a comprehensive insight of the system predicates assert and retract. As a matter of fact, we will see that the operational semantics of the system predicate retract is not the intuitive one. In order to clarify this, the fundamental mechanisms underlying the computational model of Prolog need to be reconsidered.

Prolog is a logic programming language, that is a high-level declarative language. The main idea in logic programming is that deduction can be viewed as a form of computation, and that a declarative statement `P if Q and R and S` can also be interpreted procedurally as "to solve P, solve Q and R and S". Under these assumptions, a logic program is a set of axioms, or rules, defining relations between objects and a computation of a logic program is a deduction of consequences of the program.

However, Prolog is not a logic programming language in its ultimate and purest form! We will try to give a clear insight about what this involves.

We should have said first and foremost that the major part of this chapter is quoting [LLO87] and [STE94]. In fact, the first four sections have almost entirely their origins in these two books. For the sake of the account, we preferred to borrow someone else's words instead of giving descriptions surely not so accurate and complete that the borrowed ones.

Contents of this chapter

Logic Programming

Logic programming began in the early 1970's as a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. Logic programming departs radically from the mainstream of computer languages. Rather then being derived, by a series of abstraction and reorganizations, from the VON NEUMANN machine model and instruction set, it is derived from an abstract model which has no direct relation to or dependence onto one machine model or another. It's based upon the belief that instead of the human learning to think in terms of the operations of a computer, the computer should perform instructions that are easy for humans to provide. *In its ultimate and purest form, logic programming suggests that even explicit instructions for operation not be given but rather that the knowledge about the problem and assumptions sufficient to solve it be stated explicitly, as logical axioms.* Such a set of axioms constitutes an alternative to the conventional program. The program can be executed by providing it with a problem, formalized as a logical statement to be proved, called a goal statement or a query. The execution is an attempt to solve the problem, that is to prove the goal statement, given the assumptions in the logic program.

The idea was that First Order Logic could be used as a programming language. This was revolutionary, because, until 1972, logic had only ever been used as a specification or declarative language in computer science.

Definitions. A *term* is defined inductively as follows:
1. A variable is a term.
2. A constant is a term.
3. If f is a n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom*. A *literal* is an atom or the negation of an atom. A *goal* is a sequence of literals.

Definition. A *logic program* is a finite set of clauses. A clause or rule is a logical sentence of the form $A \leftarrow B_1, \ldots, B_k.$ $(k \geq 0)$ where A and the B_i are atoms.

Such a sentence is read declaratively "A is implied by the conjunction of the B_i". A is called the *head* of the clause and the conjunction of the B_i the *body* of the clause. If $k = 0$, the clause is known as a *fact* and written $A.$.

Definition. A *query* is a conjunction of the form $\leftarrow A_1, \ldots, A_n.$ $(n > 0)$ where the A_i are goals.

Moreover it was shown that logic has a procedural interpretation which makes it very effective as a programming language. Briefly, a program clause $A \leftarrow B_1, \ldots, B_n.$ is regarded as a procedure definition. If $\leftarrow C_1, \ldots, C_k.$ is a query, then each C_j is regarded as a procedure call. A logic program is run by giving it an initial goal, a query. If the current goal, called the *resolvent*, is $\leftarrow C_1, \ldots, C_k$, a step in the computation involves unifying some C_j with the head A of a program clause $A \leftarrow B_1, \ldots, B_n.$ and thus reducing the current goal to the goal $\leftarrow (C_1, \ldots, C_{j-1}, B_1, \ldots, B_n, C_{j+1}, \ldots, C_k)\theta$, where θ is the *unifying* substitution. *Unification* is at the heart of the resolution proof procedure and thus becomes a uniform mechanism for parameter passing, data selection and data construction. The computation terminates when the *empty goal* is produced.

Informally, a *unifier* of two terms is a substitution making the terms syntactically identical. Similarly, if two terms have a unifier, we say they *unify*.

Definition. A *substitution* θ is a finite set (possibly empty) of the form $\{X_1=t_1, \ldots, X_n=t_n\}$, where each X_i is a variable, each t_i is a term distinct from X_i and the variables $X_1, \ldots, X_n$ are distinct. A pair $X_i=t_i$ is called a *binding*.

For any substitution $\theta = \{X_1=t_1, \ldots, X_n=t_n\}$ and term S, the term $S\theta$ denotes the result of simultaneously replacing in S each occurrence of the variable X_i by t_i; the term $S\theta$ is called an *instance* of S.

A computation of a logic program P thus finds an instance of a given query logically deducible from P. A goal G is deducible from a program P if there is an instance A of G where $A \leftarrow B_1, \ldots, B_n.$ $(n \geq 0)$, is a ground — i.e. where variables do not occur — instance of a clause in P, and the B_i are deducible from P.

Prolog Execution Model: SLD-Derivation

One of the main ideas of logic programming is that an algorithm consists of two disjoint components: the logic and the control. Prolog is an approximate realization of the logic programming computation model on a sequential machine. This section discusses the execution model of Prolog in contrast to logic programming.

Two major decisions must be taken to convert the computational model of logic programs into a form suitable for a concrete programming language. First, the arbitrary choice of which goal in the resolvent to reduce must be specified. Second, the non deterministic choice of the clause from the program to effect reduction must be stated. Hence we now define the notion of a Prolog program as a logic program in which an *order* is defined both for the clauses in the program and for the goals in the body of a clause.

This order of the clauses and goals is in fact the order in which the Prolog interpreter selects clauses and goals to reduce. In the execution model of Prolog, the *leftmost* goal is chosen, instead of an arbitrary one, and the non deterministic choice of a clause is made by *sequential search for a unifying* clause and *backtracking*.

When attempting to reduce a goal, the first clause whose head unifies with the goal is chosen. If no unifying clause is found for the goal to be reduced, the computation is unwound — i.e. backtracks — to the last choice made, and the next unifying clause is chosen.

A computation of a goal G with respect to a Prolog program P is the generation of all solutions of G with respect to P. In terms of logic programming concepts, a Prolog computation of a goal G is a complete depth-first traversal of a particular search tree of G obtained by always choosing the leftmost literal. A search tree is also called a *derivation tree* in Prolog jargon and is obviously unique with respect to a program.

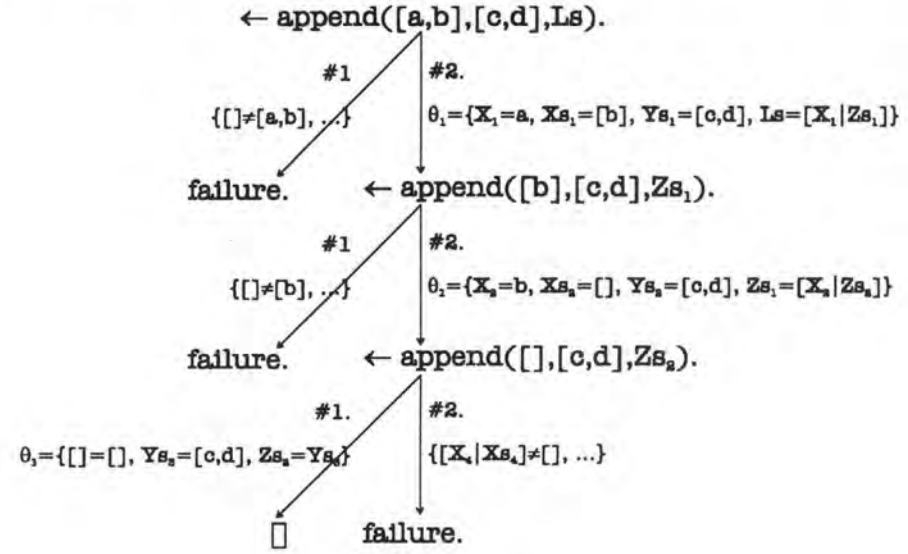
Definition. A *search tree* of a goal G with respect to a program P is a tree whose root is G. Nodes of the tree are (possibly conjunctive) goals with one selected goal. There is an edge leading from a node to each clause in the program whose head unifies with the selected goal. Each branch in the tree from the root is a computation of G by P. Leaves of the tree are *success nodes*, where the empty goal □ has been reached, or *failure nodes*, where the selected goal cannot be further reduced — that is where unification failed. Success nodes correspond to solutions of the root of the tree.

Figure 1.1 Appending two lists.

```
#1. append([],Ys,Ys).
#2. append([X|Xs],Ys,[X|Zs]) :-
        append(Xs,Ys,Zs).
```

Let us illustrate this by an example. Given the program append depicted at Figure 1.1, let us build the derivation tree of the query ← append([a,b],[c,d],Ls). The derivation tree is depicted at Figure 1.2. The edges of the tree are labeled with two components: the number of the clause being selected in the program append, and the (possibly) unifying substitution. Each time a clause is selected all the variables appearing in it are renamed, this is an inherent mechanism in the computation model of Prolog. This renaming is the logical counterpart of the classic notion of reentrant code.

Figure 1.2 The derivation tree of the query ← append([a,b], [c,d],Ls). *with respect to the program* append..


← append([a,b],[c,d],Ls).
#1
{[]≠[a,b], ...}
#2.
θ₁={X₁=a, Xs₁=[b], Ys₁=[c,d], Ls=[X₁|Zs₁]}
failure.
← append([b],[c,d],Zs₁).
#1
{[]≠[b], ...}
#2.
θ₂={X₂=b, Xs₂=[], Ys₂=[c,d], Zs₁=[X₂|Zs₂]}
failure.
← append([],[c,d],Zs₂).
#1.
θ₃={[]=[], Ys₃=[c,d], Zs₂=Ys₃}
#2.
{[X₄|Xs₄]≠[], ...}
□
failure.

The solution to the query is given by the composition of the substitutions $\theta_1\theta_2\theta_3 = \theta_4$. The resulting substitution θ_4 tells us that Ls = [a,b,c,d], which is after all what we expected.

As you can see on Figure 1.2, a Prolog program, which does not enter an infinite loop, *always finitely fails*. This will be the case when one wants to compute *all* the possible solutions to a given query, since all the derivation tree will have to be thoroughly searched. This '*finitely failing assumption*' is an important feature of Prolog and will play a central part in our original framework for full Prolog abstract interpretation.

Cut System Predicate

The cut system predicate, !, affects the procedural behavior of programs. Its main function is to reduce the search space of Prolog computations by dynamically pruning the search tree. The cut can be used to prevent Prolog from following fruitless computation paths that the programmer knows could not produce solutions. The cut can also be used to prune computation paths that do contain solutions; by doing so, a weak form of negation can be effected.

Operationally, the cut is handled as follows:

- a cut prunes all clauses below it,

- a cut prunes all alternative solutions to the conjunction of goals appearing to its left in the clause,
- a cut doesn't affect the goals to its right in the clause.

Let us restate more formally the effect of a cut in a general clause $C = A \leftarrow B_1, \ldots, B_k, !, B_{k+1}, \ldots, B_n.$ in a procedure defining A. If the current goal G unifies with the head of C, and $B_1, \ldots, B_k$ further succeed, the cut has the following effect. The program is committed to the choice of C for reducing G; any alternative clauses for A that might unify with G are ignored. Further, should B_i fail for $i>k+1$, backtracking goes back only so far as the cut. Other choices remaining in the computation of B_i $(i \leq k)$ are pruned from the search tree. If backtracking actually reaches the cut, then the cut fails and the search proceeds from the last choice made before the choice of G.

Figure 1.3 Appending two lists with cut.

```
#2. append([X|Xs],Ys,[X|Zs]) :- !,
        append(Xs,Ys,Zs).
#1. append([],Ys,Ys).
```

Let us illustrate this with an example. Suppose we have the same program **append** as depicted at Figure 1.1, but with the two clauses permuted and a cut inserted in the second clause. We obtain the program depicted at Figure 1.3. Suppose also we compute the same query as the one at the root of the derivation tree illustrated at Figure 1.2. Then the derivation tree of this goal with respect to the modified version of the program **append** is the one at Figure 1.4 — we labeled the edges with the clause number only.

Figure 1.4 The derivation tree of the query ← append([a,b], [c,d],Ls). *with respect to the modified program* append..

$$\begin{tikzcd}
\leftarrow \text{append([a,b],[c,d],Ls).} \arrow[d, "\#2"] & \\
\leftarrow \text{append([b],[c,d],}Zs_1\text{).} \arrow[d, "\#2"] & \\
\leftarrow \text{append([],[c,d],}Zs_2\text{).} \arrow[d, "\#2"] \arrow[dr, "\#1"] & \\
\text{failure.} & \Box
\end{tikzcd}$$

Program Manipulation

Prolog programs can be modified at runtime, i.e. programs can modify themselves during computation. However, we require that the user clearly distinguish of user-defined predicates[1]: that is the dynamic ones and the static ones. Built-in predicates are assumed to be static. The dynamic predicates are obviously the ones handled by the **assert** and **retract** system predicates. For the sake of the account and to make the distinction clear, we adopt the following syntactical convention: the dynamic predicates can only be called through a meta-call predicate.

[1] This is a choice we have made: we wanted to have a sound conceptual model that can ease formal reasoning about programs. If we had chosen to allow dynamic modifications of either static or dynamic user-defined predicates, there would be nothing certain upon which we could rely.

The basic predicate for adding clauses is assertz(Clause) which adds Clause as the last clause of the corresponding procedure. There is a variant of this system predicate, asserta, that adds the clause at the beginning of a procedure.

The predicate for removing clauses from a program is retract(Clause). Intuitively, retract removes from the program all the clauses unifying with Clause, and it removes them one by one. This is true to a certain extent, but there are some restrictions to this intuitive principle as we will see in the next section. At the operational level, a call to retract may only mark a clause for removal, rather than physically removing it, and the actual removal would occur only when the top-level query is solved.

« *The predicates* assert *and* retract *introduce to Prolog the possibility of programming with side effects. Code depending on side effects for its successful execution is hard to read, hard to debug and hard to reason about formally. Hence these predicates are somewhat controversial, and using them is sometimes a result of intellectual laziness or incompetence. They should be used as little as possible when programming. Many of the programs can be written using* assert *and* retract, *but the results are less clean and less efficient. Further, as Prolog compiler technology advances, the inefficiency in using* assert *and* retract *will become more apparent.*

It is possible, however, to give logical justification for some limited uses of assert *and* retract. *Asserting a clause is justified, for instance, if the clause already logically follows from the program. In such a case, adding it will not affect the meaning of the program, since no new consequences can be derived. Perhaps program efficiency will improve, as some consequences could be derived faster.*

Similarly, retracting a clause is justified if the clause is logically redundant. In this case, retracting constitutes a kind of logical garbage collection, whose purpose is to reduce the size of the program. » [STE94]

Memo-functions

Memo-functions save the results of subcomputations to be used later in a computation. Remembering partial results is impossible within Prolog, so memo-functions are implemented using side effects to the program.

The prototypical memo-function is lemma(Goal). Operationally, it attempts to prove the goal and, if successful, stores the result of the proof as a lemma. It is implemented as

```
lemma(P) ← call(P), !.
lemma(P) ← P, !, asserta(P).
```

The next time the goal P is attempted, the new solution will be used and there will be no unnecessary recomputation. The cut is present to prevent the more general program from being used; its use is justified only if P does not have multiple solutions.

Let us define a program which computes the Fibonacci function for any given number and which is supported by a memo-function. The program is depicted at Figure 1.5. No particular comment about the program is necessary beyond the fact that the second recursive call does not need to be implemented as a memo-function. It is due to the fact that the first recursive call encompasses all the number that will be computed by the second call. Therefore if both calls were implemented as memo-functions, there would be redundant asserted lemmas which would be moreover exclusive because of the presence of a cut within the asserted clauses.

Figure 1.5 Example of the lemma technique to compute the Fibonacci function.

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,R):-
      N>1,
      N1 is N-1,
      lemma(fibonacci(N1,R1)),
      N2 is N-2,
      call(fibonacci(N2,R2)),
      R is R1+R2.
```

Operational Semantics of Assert and Retract

When a literal `assert(Clause)` is executed, the system predicate always succeeds. From that moment, the asserted clause is visible to the whole program. Moreover we can consider, from a conceptual point of view, that each asserted clause is given a *unique number* that identifies it among all the other asserted clauses; the clause is also *marked* as visible or alive.

On the event of backtracking, when an `assert` is reached it fails. Therefore, the predicate `assert` succeeds always and only once.

Definition. The *logical view* is the list of all the asserted clauses that are *visible* at the time a literal `retract(Clause)` is *first* executed. The clauses taken into account unify with the argument `Clause` of the predicate `retract`. We can consider that each clause in the logical view is distinct from any other one since a unique number was considered to be assigned to at the asserting time.

When a literal `retract(Clause)` is executed the first time, it builds its ***own*** logical view. After doing so, the first clause in the logical view is retracted and marked as invisible — or dead — to the whole program. The `retract` knows exactly which clause has just been removed since a unique number identifies it. The operation also instantiates by the unification the variable arguments of the `retract` literal.

On the event of backtracking, if the `retract` is reached it removes the next clause in its logical view. Note that this clause could already have been marked as invisible by another `retract`. Anyway it is not the case that the clause is marked as invisible in this logical view since every `retract` manages its own logical view independently from any other `retract` predicate. So each time a `retract` is reached during backtracking, the next clause in its logical view, regardless the fact it could already be marked as invisible, is removed. And so on until no more clause is left in the logical view. When this latter becomes empty and a retract is reconsidered, it fails.

Therefore a `retract` succeeds always as many times as they are visible asserted clauses unifying with its argument at the time it is first executed. The retract is thus a deterministic operation as every other system predicate.

Note also that the operational semantics of the meta-call predicate, `call`, is exactly the same as the one of the `retract` except that instead of removing a clause it executes it.

Alternative View of Prolog Operational Semantics

The presentation given so far is classic in the literature, as a matter of fact it is taken from two *classic* authors in the Prolog community. Albeit classic, this view is still sometimes intricate, that is why we will try to restate things in a more intuitive way, which is moreover the way the operational semantics is understood and considered in the abstract interpretation framework. Hence, intuition does not obligatory hamper formal thinking.

Rather than being considered within its context, as was formerly presented, a literal or a goal can be seen independently from it, as if it was abstracted from its context. Recall that the *context* is the path in the derivation tree leading from the root to the current literal.

Such an abstraction is possible if we consider a goal only in terms of its input and output substitutions. Indeed, for a given input substitution a literal is susceptible to yield a series of output substitutions due to the non determinism of Prolog — later on we will speak of *sequence of substitutions* because the substitutions are ordered following the computation rules of Prolog. Moreover, each of these output substitutions can be in turn an input substitution of the next literal, that is the next to the right. This model is the *static* counterpart of the one presented before, which was described under the dynamics of Prolog. You already see the point of this approach, it follows the 'spirit' of the abstract interpretation, that is *static* considerations of a Prolog program.

Let us restate this more formally. $\theta \xrightarrow{\ell} \langle\theta_1, \ldots, \theta_n\rangle$ denotes the fact that a literal ℓ, with the given input substitution θ, can produce the sequence of output substitutions $\langle\theta_1, \ldots, \theta_n\rangle$. For instance, it is now much easier to restate the effect of the cut primitive under these assumptions. Consider that the literal ℓ is followed by a cut, then the effect of the cut after the execution of the literal will be:

$$\theta \xrightarrow{\ell,!} \langle\theta_1\rangle.$$

References

[STE94] L. Sterling and E. Shapiro; *The art of Prolog: Advanced programming techniques*; MIT Press, Cambridge Ma.; Second edition, 1994.

[LLO87] J.W. Lloyd; *Foundation of Logic Programming*; Springer-Verlag; Second edition, 1987.

Chapter

2

Abstract Interpretation

The general abstract interpretation methodology consists of associating with every program a monotonic mapping which plays a very important role in the theory. The first section introduces the requisite concepts concerning monotonic mappings and their fixpoints. Next, we intuitively show the part these notions can play in the abstract interpretation methodology. This second is greatly based on [LEC91].

Contents of this chapter

Mathematical Requirements

Partial Order and Complete Partial Order

Definition. Let S be a set. A *relation* $\mathcal{R}$ on S is a subset of S×S.

Infix notation is usually used, so we write $(x, y) \in \mathcal{R}$ as $x\mathcal{R}y$.

Definition. A relation $\mathcal{R}$ on a set S is a *partial order* if the following conditions hold:

1. $\forall x \in S, x\mathcal{R}x$ *reflexivity*
2. $\forall x, y \in S, x\mathcal{R}y \wedge y\mathcal{R}x \Rightarrow x = y$ *anti-symmetry*
3. $\forall x, y, z \in S, x\mathcal{R}y \wedge y\mathcal{R}z \Rightarrow x\mathcal{R}z$ *transitivity*

Examples. The set of natural numbers $\mathbb{N}$ under $\leq$ is partial order. Let S be a set and $\mathcal{P}(S)$ be the power set of S, that is the set of all subsets of S. Then set inclusion $\subseteq$ is a partial order on $\mathcal{P}(S)$.

Definition. Let S be a set with a partial order $\leq$. Then $x \in S$ is an *upper bound* of a subset $W \subseteq S$ if $w \leq x$ for all $w \in W$. Similarly, $y \in S$ is a *lower bound* of W if $y \leq w$ for all $w \in W$.

Definition. Let S be a set with a partial order $\leq$. Then $x \in S$ is the *least upper bound* of a subset $W \subseteq S$ if x is an upper bound of W and, for all upper bounds x' of W, $x \leq x'$ holds. Similarly, $y \in S$ is the *greatest lower bound* of a subset $W \subseteq S$ if y is a lower bound of W and for all lower bounds y' of W, $y' \leq y$ holds.

If it exists, the least upper bound of W is unique and is denoted lub(W). Similarly, if it exists, the greatest lower bound of W is unique and is denoted glb(W).

Definition. A *chain* of the partial order $(S, \leq)$ is an infinite increasing sequence $x_0 \leq x_1 \leq \ldots \leq x_i \leq \ldots$

Definition. The partial order $(S, \leq)$ is a *complete partial order* — cpo for short — if the set S owns a bottom element, denoted $\perp_S$, and if any chain of S have a least upper bound denoted $\bigcup_{i=0}^{\infty} x_i$.

Complete Lattice

Definition. A *lattice* is a partially ordered set S, in which any two elements x_1 and x_2 have a least upper bound and a greatest lower bound in S.

Definition. A lattice L is a *complete lattice* if lub(W) and glb(W) exist for every subset $W \subseteq L$.

Let $\top$ denote the *top element* lub(L) and $\bot$ denote the *bottom element* glb(L) of the complete lattice L. We can therefore deduce that any two elements of a lattice L have a common 'ancestor' and a common 'descendant'.

Examples. The Figure 4.4 is an example of a complete lattice, where $\top$ = any. In the previous example, $\mathcal{P}(S)$ under $\subseteq$ is a complete lattice. In fact the lub of a collection of subsets of S is their union and the glb is their intersection. The top element is S and the bottom one is the empty set {}.

Monotonicity and Continuity

Definition. Let L_1, L_2 be complete lattices and $T{:}L \rightarrow L$ be a mapping. T is *monotonic* iff $T(x) \leq T(y)$ whenever $x \leq y$.

Definition. Let L be a complete lattice and $T{:}L \rightarrow L$ be a mapping. We say W is *directed* if every finite subset of W has an upper bound in W.

Definition. Let L be a complete lattice and $T{:}L \rightarrow L$ be a mapping. T is *continuous* if T(lub(W)) = lub(T(W)), for every directed subset W of L. Note that the composition of two continuous mappings is still a continuous mapping.

In particular, T is continuous if for every chain $x_n \in L$ $(n \geq 0)$, $T(\bigcup_{i=0}^{\infty} x_i) = \bigcup_{i=0}^{\infty} T(x_i)$.

We can also verify that a continuous mapping is monotonic. Assume we have a set $W = \{w_1, w_2\}$, the least upper bound of W is defined as follows:

$$\text{lub}(W) = \begin{cases} w_2 \text{ if } w_1 \leq w_2 \\ w_1 \text{ if } w_2 \leq w_1 \end{cases}$$

Assume in this case that $w_1 \leq w_2$, under the continuity assumption the following relation holds: $T(w_2) = T(\text{lub}(W)) = \text{lub}(T(W))$. Hence we deduce $T(w_1) \leq \text{lub}(T(W)) = T(w_2)$. This is true in either a complete lattice or a complete partial order. The difference between a complete lattice and a complete partial order lies in the fact that the latter has only one least upper bound for all the chains while it must be true for each subset of a complete lattice.

Fixpoint

Definition. Let L be a complete lattice and $T{:}L \rightarrow L$ be a mapping. We say that $\alpha \in L$ is the *least fixpoint* of T if α is a fixpoint (that is $T(\alpha) = \alpha$) and for all fixpoints β of T, we have $\alpha \leq \beta$.

Fixpoint theorem. If D is a complete partial order and f:D $\rightarrow$ D is a continuous mapping, f has a least fixpoint denoted $\mu(f) \in$ D. Moreover, the least fixpoint $\mu(f)$ can be defined by:

$$\mu(f) = \cup_{i=0}^{\infty} f^{i}(\perp)$$

where $f^{0}(\mathbf{x}) = \mathbf{x}$, and

$$f^{n+1}(\mathbf{x}) = f(f^{n}(\mathbf{x})).$$

The continuity guarantees the existence of a fixpoint. The proof of the fixpoint theorem consists mainly of two steps.

- We prove by induction that $\{f^{n}(\perp)\}$ is a chain thanks to the monotonicity of f.
- Thanks to the continuity, we deduce $f(\cup_{i=0}^{\infty} f^{i}(\perp)) = \cup_{i=0}^{\infty} f^{i+1}(\perp) = \cup_{i=0}^{\infty} f^{i}(\perp)$. So that $\cup_{i=0}^{\infty} f^{i}(\perp)$ is a fixpoint of f.

The fixpoint theorem will allow us to interpret recursive definitions like x = f(x) as least fixpoints. It also forms the basis of the abstract interpretation algorithms described later on through this report.

Static Analysis and Abstract Interpretation

Static analysis of programs encompasses all the treatments a program can undergoes except its execution. The information collected this way serves as a basis either to transform the program in order to optimize it — for instance, during compilation — or to verify it fulfills some correction criteria.

Different techniques of static analysis have been used for a long time in compilers without requiring a theoretical basis. However the correctness of such techniques is essential, for before being optimized, a program has essentially to be correct. Due to the complexity of the programming languages, proving that a static analysis method is correct is an intricate and tedious task.

Abstract interpretation was introduced by P. and R. Cousot [COU77] as a *general mathematical framework* to express current methods and to prove more systematically their correctness. The main idea of the method is to execute the program over an *abstract domain* instead of the standard one. Nevertheless two conditions have to be satisfied:

1. elements of the abstract domain captures interesting and useful properties of elements of the standard domain,
2. computation over the abstract domain should be efficient and converge in a *finite* number of steps.

There exist well-known rules which are applications of this idea. For instance, algebraic multiplication can be abstracted by the signs rule which amounts to reduce the set of numbers to the abstract set of signs: $\{+, -\}$. The algebraic multiplication can this way be summarized by the four rules: $+ \times + = +$, $+ \times - = -$, $- \times + = -$, $- \times - = +$. With these rules, it is easy to know the sign of any product of any length without having to actually effect it.

Abstract interpretation was first introduced to systematize static analysis effected in classical programming with Algol-like languages. However we assisted to a real breakthrough of abstract interpretation with the quite recent development of declarative programming and most particularly of logic programming. Indeed, as it was said before, a declarative program is of a higher level of formalism and, in its ideal form, should just specify the results without mentioning how to compute them. Such high-level languages harbors much more opportunities of optimizations and hence static analysis.

The mathematical theory

Let D denotes the domain of values handled by a language we want to analyze by means of abstract interpretation. In general, D will have a complex structure to take into account the different basic types of the languages: scalars, structured types, files... Assume that D is unstructured, this will not change the nature of results but will simplify the account. Note that other simplifications will be made in the following without ever be mentioned. As you will see later, all this can be *extended* to the case of actual languages but not without overhead.

Suppose we want to analyze the following procedure of our language, written in a functional style. The procedure computes values of a function from D to D:

$$f(x) = [\ \mathbf{if}\ x > 100\ \mathbf{then}\ x-100\ \mathbf{else}\ f(f(x+11))\ ]$$

This is the 91-function which computes values from $\mathbb{Z}$ to $\mathbb{Z}$ (the set of integers). An interesting property that we could try to analyze about this function amounts to know which values will take the variables at different points[2] of the execution.

To capture such *global* information we have to replace individual values of D by sets of values, i.e. elements of $\mathcal{P}(D)$. This set will be denoted C, as the concrete domain of all possible properties. The function calculating over D can now be replaced by a function calculating over C. So we get the following functional 'expression':

$$f(X) = \{x-10: x > 100 \text{ and } x \in X\} \cup f(f(\{x+11: x \leq 100 \text{ and } x \in X\})) \quad (1)$$

Basic operations, operating on individual values, can generally be replaced – *lifted* – by operations handling sets of values. Functions transformed this way compute the set of possible results corresponding to the set of possible data. Actually those 'procedures' are not useful for two reasons.

Any sets of values are not workable, it is moreover theoretically impossible. The concrete domain C will be replaced by an abstract domain A conserving only some elements of C such that any element of C can be approximated by an element of A. Technically it is often demanded that A be a complete lattice and that two monotonic functions, $\alpha: C \rightarrow A$, and $\gamma: A \rightarrow C$, exist and verify these two conditions:

- $\forall c \in C: \gamma(\alpha(c)) \supseteq C;$
- $\forall a \in A: \alpha(\gamma(a)) = a.$

The abstraction function α associates *each set* of values with its best approximation. The concretization function γ associates *each element* of A with the set of values it represents. In the above example, $C = \mathcal{P}(\mathbb{Z})$. An example of abstract domain A would be the set of intervals $\mathfrak{I}$: an interval $[i..s]$ where $i,s \in \mathbb{Z} \cup \{-\infty, +\infty\}$ is the set of integers e such that $i \leq e \leq s$. The set $\mathfrak{I}$ is a complete lattice for the set inclusion $\subseteq$. {} is the least element.

[2] We call "*granularity*" the notion which consists of establishing the program points where information about all possible executions is collected.

$[-\infty, +\infty]$ is the greatest element. $\alpha(\mathtt{x})$ is the interval $[\mathtt{min(x)..max(x)}]$. γ is the inclusion of $\mathfrak{I}$ in $\mathcal{P}(\mathbb{Z})$.

It is possible to rewrite the former function such that it maps an interval of data to an interval of results.

$$f([\mathtt{i..s}]) = [\mathtt{max}(91, \mathtt{i}-10)..(\mathtt{s}-10)] \mathbin{\dot\cup} f(f([(\mathtt{i}+11)..\mathtt{min}(\mathtt{s}+11, 111)])) \quad (2)$$

where $\dot\cup$ approximates the union of two intervals by the smallest interval that includes both of them, that is to say their glb. From this definition, let us try to approximate the set of values produced by the function. We obtain the following two equations:

$$f([-\infty..+\infty]) = [91..+\infty] \mathbin{\dot\cup} f(f([-\infty..111])),$$

$$f([-\infty..111]) = [91..101] \mathbin{\dot\cup} f(f([-\infty..111])).$$

These equations demonstrate that the usual computation method is impossible, it would entail an infinite loop since $f([-\infty..111])$ recursively triggers off the computation of itself. We have just pointed out the second difficulty. Computations over the abstract domain cannot blindly simulate computations over the standard domain. We separate the difficulty in two levels. On the one hand, we will give an accurate – mathematical – sense to the procedures such that (1) and (2) thanks to the least fixpoint of a monotonic transformation. On the other hand, we will expose in the next section the problem of calculating least fixpoints.

We will reason over the abstract domain of intervals $\mathfrak{I}$, but the same process could be applied over any abstract domain. Let $\mathfrak{I} \to \mathfrak{I}$ be the set of monotonic and continuous functions from $\mathfrak{I}$ to $\mathfrak{I}$, i.e. such that:

- $\forall\, \mathrm{I}, \mathrm{I}' \in \mathfrak{I}: \mathrm{I} \subseteq \mathrm{I}' \Rightarrow f(\mathrm{I}) \subseteq f(\mathrm{I}')$,
- for all infinite series of embedded intervals $\mathrm{I}_1 \subseteq \dots \subseteq \mathrm{I}_i \subseteq \dots$

$$f(\cup_{i=1}^{\infty} \mathrm{I}_i) = \cup_{i=1}^{\infty} f(\mathrm{I}_i).$$

The conditions express that f is well an abstraction of a function from $\mathbb{Z}$ to $\mathbb{Z}$. The set $\mathfrak{I} \to \mathfrak{I}$ can be endowed with an order:

$$f \le g \text{ iff } \forall\, \mathrm{I} \in \mathfrak{I}: f(\mathrm{I}) \subseteq g(\mathrm{I}).$$

This order means that the procedure corresponding to g produces at least as much results as the procedure corresponding to f. The functional definition (2) can be replaced by a transformation of function:

$$\tau: (\mathfrak{I} \to \mathfrak{I}) \to (\mathfrak{I} \to \mathfrak{I})$$

with

$$(\tau f)\,([\mathtt{i..s}]) = [\mathtt{max}(91, \mathtt{i}-10)..(\mathtt{s}-10)] \mathbin{\dot\cup} f(f([(\mathtt{i}+11)..\mathtt{min}(\mathtt{s}+11, 111)])).$$

Hence, the equation (2) simply means that the function f is a fixpoint of the transformation τ, that is to say $(\tau f) = f$.

We can show the following results.

1. Provided that the concrete and abstract domains verify the formerly mentioned properties, *every* fixpoint of τ gives a correct approximation of the properties of the associated procedure.

2. The transformation τ has necessarily a least fixpoint (the most accurate one).

3. The least fixpoint of τ is equal to the limit of an increasing series of approximations:

$$f_0 \subseteq \dots \subseteq f_k \subseteq \dots$$

where $f_0(I) = \{\} \quad \forall\, I \in \Im$,

$f_{k+1} = \tau(f_k) \quad \forall\, k \geq 0.$

The algorithms

Fixpoints computations of some transformations associated with the programs according to an abstract semantics are close to recurrent procedures computations in a programming language. There are however two major differences:

- computation must terminate in all possible cases;
- it is generally sufficient to compute an approximation of the fixpoint itself.

Bottom-up and top-down evaluation of recursive definitions

Fixpoint algorithms are derived from the methods allowing to evaluate recurrent definitions of functions. That's why we first expose these methods before generalizing them for abstract interpretation. Consider the following recursive definition:

```
f(x) = [ if x ∈ {0, 1}
         then x
         else f(x–1) + f(x–2) ].
```

To compute a particular value $f(v)$, we could first evaluate the function bottom-up, starting from the "*small*" values 0 and 1 for which the value of f is immediate, and next, by progressively propagating these results to 2, 3, 4... until we obtain the desired value. If we wanted to compute $f(4)$, this would look like:

$f(0) = 0$
$f(1) = 1$
$f(2) = f(1) + f(0) = 1 + 0 = 1$
$f(3) = f(2) + f(1) = 1 + 1 = 2$
$f(4) = f(3) + f(2) = 2 + 1 = 3$

We could also evaluate the function top-down, starting from the expression $f(v)$ to compute and by further developing it until we obtain an expression easily computable. Again for $f(4)$, this would look like:

$$\begin{aligned}
f(4) &= f(3) + f(2)\\
&= (f(2) + f(1)) + (f(1) + f(0))\\
&= ((f(1) + f(0)) + f(1)) + (f(1) + f(0))\\
&= ((1 + 0) + 1) + (1 + 0)\\
&= (1 + 1) + 1 = 2 + 1\\
&= 3
\end{aligned}$$

The bottom-up method seems to be more efficient on the above example. However, it is difficult to systematize it for it needs to find the right series of values to compute and which moreover always rely on values previously processed. It is not always possible. The top-down method has the advantage to be systematic. Unfortunately it is very inefficient because the same value could be often reevaluated. On the example $f(2)$ is computed twice, and generally, top-down computation are exponential in time, when the bottom-up method is generally linear.

Fortunately the top-down method can be systematically improved to be generally as efficient as the bottom-up one. This improvement, known as *memo-ization*, will be enhanced for the abstract interpretation algorithms. It consists to record in a table the values already computed in order to prevent them from being evaluated twice: they will be simply picked up in the table. The former example would this time give:

$$\begin{aligned}
f(4) &= f(3) + f(2) \\
&= (f(2) + f(1)) + f(2) \\
&= ((f(1) + f(0)) + f(1)) + f(2) \\
&= ((1 + f(0)) + f(1)) + f(2) && \{f(1) = 1\} \\
&= ((1 + 0) + f(1)) + f(2) && \{f(0) = 0\} \\
&= (1 + f(1)) + f(2) && \{f(2) = 1\} \\
&= 2 + f(2) && \{f(3) = 2\} \\
&= 3 && \{f(4) = 3\}
\end{aligned}$$

Now, the algorithm is linear, since every expression $f(v)$ is only evaluated once.

Abstract interpretation algorithms

Suppose we want to compute the least fixpoint of the abstract transformation

$$\tau\colon (A \to A) \to (A \to A).$$

Recall that we want an algorithm capable of computing $f(a)$ for all abstract value a. Moreover we do not want to compute the whole set of pairs $\langle a, f(a)\rangle$, although this is what would do the simplest algorithm: to compute the series of approximations

1. $f_0(a) = \bot$ $\quad \forall\, a \in A,$
2. $f_{k+1} = \tau(f_k)$ $\quad \forall\, k \geq 0,$
3. $f = f_n$ $\quad$ such that $f_{n+1} = f_n$.

This bottom-up method, demanding the domain A be finite, is much too inefficient: at the first iterate we obtain all the values directly computable, at the second one, those values that are only depending on the previous ones and so on. This method can be easily improved by trying to dynamically determine the values that are strictly necessary to compute $f(a)$ for the particular value a in which we are interested.

On the other hand, a top-down method consists in memorizing intermediate and partial results of a computation, this is an enhancement of the *memo-ization*. We start with the recursive definition of the fixpoint and we try to recursively compute the value $f(a)$. During the computation, we keep up-to-date an array of values a_i for which a recursive call has already been initiated (whether terminated or not) together with its associated *lower* approximation $f(a_i)$. When the same recursive call is reconsidered, it is not developed but its current approximation in the array is returned. At the end of each call, the content of the array is updated with the value just computed. As this latter is generally an approximation — since the values in the array are so — we repeat the same computation until the result cannot be further improved.

Let us illustrate this top-down algorithm with the fixpoint computation of the 91-function, so we have to compute $f([-\infty..+\infty])$ for the transformation

$$(\tau f)([\mathtt{i}..\mathtt{s}]) = [\mathtt{max}(91, \mathtt{i}-10)..(\mathtt{s}-10)] \mathbin{\dot\cup} f(f([(\mathtt{i}+11)..\mathtt{min}(\mathtt{s}+11, 111)])).$$

Each time a recursive call is initiated, a value is added in the array, representing the current approximated result: $\{\}$. Hence, when we start the computation, the approximated value of $f([-\infty..+\infty])$ in the array is $\{\}$. After a first iterate we get:

$$f([-\infty..+\infty]) = [91..+\infty] \mathbin{\dot\cup} f(f([-\infty..111])).$$

Thus a call for $f([-\infty..111)$ — whose current approximation is $\{\}$ — is initiated. This call can be further developed in

$$f([-\infty..111]) = [91..101] \mathbin{\dot\cup} f(f([-\infty..111])).$$

The same recursive call should be initiated, but we rather pick up its current approximated result in the array; it would have otherwise entails an infinite loop. Recall that $f([-\infty..111])$ is associated with $\{\}$ in the array. Hence we get:

$$\begin{aligned} f([-\infty..111]) &= [91..101] \mathbin{\dot\cup} f(\{\}) \\ &= [91..101]. \end{aligned}$$

This new result is updated within the array, and next the same computation is reconsidered just in case we could improve the current result thanks to the new information just obtained.

$$\begin{aligned} f([-\infty..111]) &= [91..101] \mathbin{\dot\cup} f(f([-\infty..111])) \\ &= [91..101] \mathbin{\dot\cup} f([91..101]). \end{aligned}$$

This new iterate triggers off the computation of $f([91..101])$. Again the same method is applied to compute $f([91..101])$. As being quite long, it is not mentioned. The least fixpoint computation terminates with the best possible result

$$f([-\infty..+\infty]) = [91..+\infty].$$

This algorithm can be systematically implemented for any kind of language, provided it has been endowed with an abstract semantics allowing to associate with any program a transformation τ. This is theoretically always possible.

Approximation, termination and convergence

The abstract domain $\mathfrak{I}$ of intervals is infinite. With such a domain, the algorithm presented can loop. It will be the case if an infinity of different calls are initiated; it will not be the case if the same call is recursively repeated, this is avoided by the array. The previous example was not concerned with this eventuality, but this is not always the case. To avoid such problems we could limit ourselves to finite abstract domains, but it is not always possible to find a better finite domain approximating the infinite domain we wish to use. A much clever approach consists in dynamically choosing those approximations, according to the particular example. The idea is to replace the virtually infinite set of values to consider by a finite set which covers them. Results obtained this way will generally be *safe – consistent* – approximations of the fixpoint values. The use of these approximations moreover enables to fasten convergence. We can illustrate this again with the computation of $f([-\infty..+\infty])$. We first compute

$$f([-\infty..+\infty]) = [91..+\infty] \dot{\cup} f(f([-\infty..111])).$$

As $[-\infty..111] \subseteq [-\infty..+\infty]$, we can replace $f([-\infty..111])$ by $f([-\infty..+\infty])$ — with a risk of losing precision — that will give us

$$\begin{aligned} f([-\infty..+\infty]) &\cong [91..+\infty] \dot{\cup} f(f([-\infty..+\infty])) \\ &\cong [91..+\infty] \dot{\cup} f(\{\}) \\ &\cong [91..+\infty] \end{aligned}$$

When reconsidering the computation we *finally* get

$$\begin{aligned} f([-\infty..+\infty]) &\cong [91..+\infty] \dot{\cup} f(f([-\infty..+\infty])) \\ &\cong [91..+\infty] \dot{\cup} f([91..+\infty]) \\ &\cong [91..+\infty] \dot{\cup} f([-\infty..+\infty]) \\ &\cong [91..+\infty] \dot{\cup} [91..+\infty] \\ &\cong [91..+\infty] \end{aligned}$$

The result is obtained after two iterations with the best possible result, when the exact algorithm would have required tens iterations. Of course the example is well tailored for the account. The design of operations allowing to speed up the convergence without losing too much precision is an intricate task, strongly depending on the specific abstract domain.

References

[COU77] P. Cousot and R. Cousot; *Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fix-points*; In Conf. Record of Fourth ACM Symposium on POPL, pp. 238-252, Los Angeles, CA, 1977.

[LEC91] B. Le Charlier; *L'analyse statique des programmes par l'interprétation abstraite*; Nouvelles de la Science et des Technologies, vol. 9, N°4, 1991, pp. 19-25.

[LLO87] J.W. Lloyd; *Foundation of Logic Programming*; Springer-Verlag; Second edition, 1987.

Chapter

3

Generic Abstract Interpretation

In this chapter we discuss a number of important issues for the design of abstract interpretation frameworks. In the first section we give a methodological overview of our approach — we insist of its wide applicability and theoretical soundness. The next section present the general approach we follow to design the frameworks presented in the second part of this report.

Contents of this chapter

Design of Abstract Interpretation Frameworks

We present an approach to the design of abstract interpretation framework that is

- *theoretically sound*: it encompasses systematic methods for proving correctness of applications;
- *widely applicable*: it is not restricted to a particular class of languages or analyses;
- *effective*: it has been used to design practical systems that have shown efficient and accurate for static analysis of Prolog.

A first idea is that packages for static analysis should consist of one or several *generic abstract interpretation algorithms* parametrized on an abstract domain and of one or more implemented *abstract domains* with respect to which the algorithms can be instantiated.

Abstract fixpoint semantics

Static program analyses aim at deriving information about the actual behavior of programs. This information is in fact completely determined by the operational semantics of the programming language. It is nevertheless convenient to use another non standard semantics as a basis for performing the analyses, because it is in practice impossible to 'generate and analyze' all possible execution traces for a given program.

There are several approaches to the choice of the non standard semantics. We basically follow the original approach of P. and R. Cousot. Their so-called *collecting semantics* provides a fixpoint characterization of the relevant information.

Let us sketch the approach from a general point of view. The operational semantics of any programming language consists of

- a set of states Σ (states are denoted σ);
- an immediate transition relation between states denoted $\rightarrow$, where $\sigma \rightarrow \sigma'$ means that σ' is a possible successor state of σ;
- an unary predicate on states, denoted $\mathtt{final}(\sigma)$, meaning that execution terminates in state σ.

From this semantics one can derive several fixpoint semantics, that collect different kinds of relevant information. For example the following semantics characterizes the set $\mathtt{Output(S)}$ of final states reachable from an initial set of states $\mathtt{S}$:

$$\mathtt{Output(S)} = \{\sigma\text{: } \sigma \in \mathtt{S} \wedge \mathtt{final}(\sigma)\} \cup \mathtt{Output}(\{\sigma'\text{: } \sigma \in \mathtt{S} \wedge \sigma \rightarrow \sigma'\}).$$

As we said in the previous chapter, such a recursive definition is not computable by usual recursive evaluation for two reason at least:

- $\mathtt{S}$ can be infinite;
- sequences of transitions can be infinite.

However it can given a mathematical meaning as the least fixpoint of a transformation τ such that

$$(\tau f)(\mathtt{S}) = \{\sigma\text{: } \sigma \in \mathtt{S} \wedge \mathtt{final}(\sigma)\} \cup f(\{\sigma'\text{: } \sigma \in \mathtt{S} \wedge \sigma \rightarrow \sigma'\}).$$

It is straightforward to show from the operational semantics that this definition correctly maps each set of states onto the set of final states reachable from them. The exact and un-computable fixpoint semantics can then be approximated as follows. First we define a set A of abstract states. An abstract state α is a finite representation of a set S of 'concrete' states. We note Cc the function that maps abstract states on their meaning (therefore $S = Cc(\alpha)$). The next step is to derive an 'abstract' version τ_A of τ that maps monotonic functions from $A \to A$ to $A \to A$. This transformation correctly abstracts τ if the following *consistency*, or safeness, condition holds:

$$\left.\begin{array}{r}\forall f: P(\Sigma) \to P(\Sigma)\\ \forall f_A: A \to A\\ \forall \alpha \in A\end{array}\right\}: (\forall \alpha' \in A: f(Cc(\alpha')) \subseteq Cc(f_A(\alpha')))$$

$$\Downarrow$$

$$(\tau f)(Cc(\alpha)) \subseteq Cc((\tau_A f_A)(\alpha)).$$

Consistency simply means that $\tau_A f_A$ safely approximates τf provided that f_A safely approximates f. If τ_A is furthermore monotonic its least fixpoint exists and safely approximates the collecting semantics:

$$\forall \alpha \in A: \mu\tau(Cc(\alpha)) \subseteq Cc(\mu\tau_A(\alpha)).$$

An *abstract semantics* for the language L is a set of rules to derive transformation τ_A for any program of L. It is basically defined by mimicking the collecting semantics definition and replacing 'concrete' operations by 'abstract' ones. In the case of our simplified language, we define two global operations:

$$FS(S) = \{\sigma: \sigma \in S \wedge \mathrm{final}(\sigma)\},$$

$$DR(S) = \{\sigma: \exists \sigma' \in S: \sigma' \to \sigma\}.$$

FS stands for 'Final States' and DR for 'Directly Reachable'. We can rephrase the definition of τ:

$$(\tau f)(S) = FS(S) \cup f(DR(S)).$$

The abstract semantics is identical to the concrete one except that concrete operations are replaced by their abstract counterpart, FS_A, DR_A, $\cup_A$:

$$(\tau_A f_A)(\alpha) = FS_A(\alpha) \cup_A f_A(DR_A(\alpha)).$$

τ_A is consistent provided that the abstract operations are. That is:

$$\begin{array}{ll}\forall \alpha \in A & : FS(Cc(\alpha)) \subseteq Cc(FS_A(\alpha))\\ \forall \alpha \in A & : DR(Cc(\alpha)) \subseteq Cc(DR_A(\alpha))\\ \forall \alpha_1, \alpha_2 \in A & : Cc(\alpha_1) \cup Cc(\alpha_2) \subseteq Cc(\alpha_1 \cup_A \alpha_2)\end{array}$$

Such an abstract semantics is *generic* with respect to the abstract domain: it does not depend on the particular choice of A, FS_A, DR_A, $\cup_A$. This is an important feature because it allows to reuse the same semantics (and therefore all the algorithms derived from it) for many different applications. Note that the collecting semantics is a particular instance of the abstract semantics. Alternatively it is possible to define a generic abstract semantics without relying on an explicit collecting semantics. This requires only to modify the consistency definitions for abstract operations. FS_A, DR_A and $\cup_A$ are now consistent iff:

$$\forall \sigma \in \Sigma: \forall \alpha \in A: \left.\begin{array}{c}\sigma \in \mathtt{Cc}(\alpha)\\ \mathtt{final}(\sigma)\end{array}\right\} \Rightarrow \sigma \in \mathtt{Cc}(\mathtt{FS}_A(\alpha))$$

$$\forall \sigma, \sigma' \in \Sigma: \forall \alpha \in A: \left.\begin{array}{c}\sigma \in \mathtt{Cc}(\alpha)\\ \sigma \to \sigma'\end{array}\right\} \Rightarrow \sigma' \in \mathtt{Cc}(\mathtt{DR}_A(\alpha))$$

$$\forall \sigma \in \Sigma: \forall \alpha_1, \alpha_2 \in A: \left.\begin{array}{c}\sigma \in \mathtt{Cc}(\alpha_1)\\ \mathtt{or}\\ \sigma \in \mathtt{Cc}(\alpha_2)\end{array}\right\} \Rightarrow \sigma \in \mathtt{Cc}(\alpha_1 \cup_A \alpha_2)$$

Basic operations from the standard operational semantics are now used instead of the 'collecting' ones: FS, DR. Relying on basic operations allows for simpler definitions in practice. However the later approach can look less systematic because consistency definitions do no longer fit in the same formal scheme. It must be noticed that a collecting semantics still exists implicitly: it can be defined as the more precise (or 'concrete') instantiation of the generic semantics. Simplicity stems from the fact that it is no longer *explicitly* defined.

The above presentation looks very simple because it was assumed that a relation $\to$ and a predicate final are primitive operations of the language. In real size applications those (complex) operations are defined by means of simpler primitives and the abstract semantics uses operations abstracting those simpler primitives. Real size abstract semantics are therefore more complex but the design principle is entirely the same.

Prolog Abstract Interpretation

The approach we follow consists mainly of three steps:

- the definition of a fixpoint semantics of the programming language, called the concrete semantics;
- an abstraction of the concrete semantics to produce a particular semantics;
- the design of an algorithm to compute relevant parts of the smallest fixpoint of the abstract semantics (or of a conservative approximation of it).

The concrete semantics may be different from the standard semantics of the language. However the concrete semantics should enable to observe the properties which have to be inferred. It also needs not to be unique for all applications but can be tailored for a class of applications.

An abstract semantics is an abstraction of the concrete semantics on some abstract domain for a given application. There can be a variety of abstract semantics derived from a single concrete semantics, each of them being suited for a specific application.

The link between the concrete and the abstract semantics is given by a concretization function from the abstract domain to the concrete one. The concretization function specifies which concrete objects are represented by one abstract object.

Moreover, each of the concrete operations is associated with a consistent abstract operation. The consistency condition makes sure the abstract operation is a *safe approximation* of its concrete counterpart.

Definition. Let O_α be an abstract operation of arity $\mathtt{n}$ and O_c be the corresponding concrete one. Let also $\alpha_1, \ldots, \alpha_n$ be abstract elements. O_α is *consistent* with respect to O_c iff if satisfies the following property:

$$\left.\begin{array}{r}\gamma_1 \in Cc(\alpha_1), \ldots, \gamma_n \in Cc(\alpha_n)\\ \gamma \in O_c(\gamma_1, \ldots, \gamma_n)\end{array}\right\} \Rightarrow \gamma \in Cc(O_\alpha(\alpha_1, \ldots, \alpha_n))$$

Consistency of the abstract operations ensures that the abstract semantics only represents correct properties of the concrete program.

Any fixpoint of the abstract transformation safely approximates the least fixpoint of the concrete transformation when either the concretization function or the abstract transformation are monotonic.

We apply the above approach to the abstract interpretation of Prolog programs. The concrete semantics is a fixpoint characterization of SLD-Resolution. The concrete semantics is defined in terms of tuples $(\Theta_{in}, p, \Theta_{out})$ where Θ_{in}, Θ_{out} are sets of substitutions on $\{X_1, \ldots, X_n\}$ and p/n is a predicate symbol. The purpose of the semantics is to characterize the tuples $(\Theta_{in}, p, \Theta_{out})$ for a program P such that Θ_{out} represents the substitutions obtained by solving $p(X_1, \ldots, X_n)\theta$ with respect to P for all $\theta \in \Theta_{in}$. The concrete semantics is abstracted by replacing sets of substitutions by abstract substitutions and replacing the concrete operations by abstract ones.

Since a concrete semantics can yield various abstract semantics, we present a generic abstract semantics based on [LCMVH91], i.e. a semantics parametrized by an abstract domain and a number of abstract operations. A particular application is obtained by instantiating the abstract domain and operations. Finally a top-down algorithm to compute the least fixpoint of the abstract semantics is given. The algorithm is query directed — i.e. it computes only the part of the fixpoint relevant to the query — and uses optimizations such as early detection of termination and dependencies between procedure calls.

References

[COU77] P. Cousot and R. Cousot; *Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*; In Conf. Record of Fourth ACM Symposium on POPL, pp. 238-252, Los Angeles, CA, 1977.

[LCVH92] B. Le Charlier and P. Van Hentenryck; *On the design of generic abstract interpretation frameworks*; In M. Billaud and all, editors, Proceedings of the Workshop on Static Analysis (WSA'92), Bordeaux, France, September 1992. Bigre 81-82.

[LCMVH91] 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.

Part

II

Prolog Abstract Interpretation

In this part we will present different abstract interpretation frameworks. These frameworks keep to be enhanced through the following chapters and have been designed accordingly to the principles presented in the previous part. Moreover almost each of these frameworks has been actually implemented and tested on a large variety of Prolog programs and have proved to be very efficient.

The first framework is designed to analyze statically pure Prolog programs. The second one, besides the functionality of the previous one, pays special attention to the control mechanisms of Prolog and more particularly to the cut. The last one is capable of analyzing full Prolog programs, that is to say programs containing virtually any kind of predicates, and especially `asserta`, `assertz` and `retract`.

Chapter

4

Pure Prolog Abstract Interpretation

The generic abstract interpretation framework and algorithm presented in this chapter were first exposed in [LCVH94]. They are designed to capture information about pure Prolog programs, that is Prolog programs that could be considered as pure logic programs, without taking into account the control issue of the Prolog computational model. Furthermore, the algorithm is designed to take advantage of early detection of termination and redundant computations thanks to a dependency graph, and designed to foil the infinite nature of the abstract domain by means of a widening technique...

The approach is presented here in great details and therefore will serve as a basis for the future enhancements of the abstract interpretation algorithm. Nowhere else the approach will be such detailed, only the enhancements will be mentioned and explained.

Contents of this chapter

Concrete Semantics

Normalized programs and substitutions

Figure 4.1 The normalized version of the program append.

```
append(X1,X2,X3):-
        X1=[],
        X3=X2.
append(X1,X2,X3):-
        X1=[X4|X5],
        X3=[X4|X6],
        append(X5,X2,X6).
```

The generic abstract interpretation semantics and algorithm are defined on normalized logic programs. The use of normalized programs, first suggested in [BJCD87], greatly simplifies the semantics, the algorithm and its implementation. The normalized version of the classical list appending program generated by our implementation is depicted at Figure 4.1.

Assume the existence of the sets $\mathbb{F}_i$ and $\mathbb{P}_i$ ($i \geq 0$) denoting sets of function and predicate symbols of arity i and an infinite set $\mathbb{PV}$ of *program variables*. Variables in $\mathbb{PV}$ are ordered and denoted by $X_1, X_2, \ldots, X_i, \ldots$ Normalized programs are built from $\mathbb{PV}$. A normalized program is a set of clauses of the form:

$$p(X_1, \ldots, X_i) \leftarrow \ell_1, \ldots, \ell_n.$$

where $p(X_1, \ldots, X_i)$ is the head and $\ell_1, \ldots, \ell_n$ the body. If a clause contains n variables, they are necessary $X_1, \ldots, X_n$. The literals in the body of a clause can be either a procedure call (the first form below) or built-ins enabling to achieve unification (the last two forms below):

- $q(X_{i_1}, \ldots, X_{i_n})$ where $X_{i_1}, \ldots, X_{i_n}$ are distinct variables and $q \in \mathbb{P}_n$;
- $X_{i_1} = X_{i_2}$ with $X_{i_1} \neq X_{i_2}$;
- $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$ where $f \in \mathbb{F}_{m-1}$ and $X_{i_1}, \ldots, X_{i_m}$ are distinct variables.

The rationale of normalized programs comes from the fact that an input/output substitution for a procedure p/n is always expressed in terms of variables $X_1, \ldots, X_n$. This greatly simplifies all the traditional problems encountered with renaming.

We also assume the existence of another infinite set $\mathbb{RV}$ of *renaming variables*. Terms and substitutions are constructed using program and renaming variables. We distinguish two kinds of substitutions: *program variable substitutions* (*ps* for short) whose domain and codomain are subsets of $\mathbb{PV}$ and $\mathbb{RV}$ respectively, and *renaming variable substitutions* – or *standard substitutions* – (*rs* for short) whose domain and codomain are subsets of $\mathbb{RV}$. In the following, $\mathbb{PS}$ denotes the set of *ps* and $\mathbb{RS}$ the set of *rs*.

Let θ be a substitution and $D \subseteq dom(\theta)$. The restriction of θ to D, denoted $\theta_{|D}$, is the substitution σ such that $dom(\sigma)=D$ and $X\theta=X\sigma \; \forall X \in D$.

The definition of substitution composition is slightly modified to take into account the special role held by program variables. The modification occurs when $\theta \in \mathbb{PS}$ and $\sigma \in \mathbb{RS}$ for which $\theta\sigma \in \mathbb{PS}$ is defined by $dom(\theta\sigma) = dom(\theta)$ and $X(\theta\sigma) = (X\theta)\sigma$ for all X

$\in \mathrm{dom}(\theta)$. 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 renaming variable that appears nowhere else. Unifiers are defined as usual but only belongs to IRS. We use $\mathrm{mgu}(t_1, t_2)$ to denote the set of most general unifiers of t_1 and t_2. Let Θ be a subset of IPS. Θ is *complete* if and only if for all $\theta \in \Theta$, θ and θ' being variant implies that $\theta' \in \Theta$. Being complete formalizes the fact that variant program substitutions are somehow the "same".

Let D be a subset of IPV. $\mathrm{CS}_D = \{\Theta\colon \forall\theta \in \Theta, \mathrm{dom}(\theta)=D \text{ and } \Theta \text{ is complete}\}$. CS_D is a complete lattice with respect to set inclusion $\subseteq$.

Concrete operations

Union of Sets of Substitutions. Let $\Theta_1, \ldots, \Theta_n \in \mathrm{CS}_D$ and $D \subseteq$ IPV.

$$\mathrm{UNION}(\Theta_1, \ldots, \Theta_n) = \Theta_1 \cup \ldots \cup \Theta_n$$

Unification of Two Variables. Let $D = \{X_1, X_2\}$ and $\Theta \in \mathrm{CS}_D$.

$$\mathrm{AI\_VAR}(\Theta) = \{\theta\sigma\colon \theta \in \Theta, \sigma \in \mathrm{IRS} \text{ and } \sigma \in \mathrm{mgu}(X_1\theta, X_2\theta)\}$$

Unification of a Variable and a Function. This operation unifies X_1 with $f(X_2, \ldots, X_n)$. Let $D = \{X_1, \ldots, X_n\}$, $\Theta \in \mathrm{CS}_D$ and $f \in \mathrm{IF}_{n-1}$.

$$\mathrm{AI\_FUNC}(\Theta, f) = \{\theta\sigma\colon \theta \in \Theta, \sigma \in \mathrm{IRS} \text{ and } \sigma \in \mathrm{mgu}(X_1\theta, f(X_2, \ldots, X_n)\theta)\}$$

Restriction and Extension of a Set of Substitutions for a Clause. The RESTRC operation restricts a set of substitutions on all the variables occurring in a clause to the variables occurring only in the head. The EXTC operation extends a set of substitutions on variables occurring in the head of a clause with all the variables occurring in the body of the clause. Let c be a clause, D_H be the set of variables in the head and D_C be the set of all the variables occurring in c.

$$\mathrm{RESTRC}(c, \Theta) = \{\theta_{|D_H}\colon \theta \in \Theta\}$$

$$\mathrm{EXTC}(c, \Theta) = \{\theta\colon \mathrm{dom}(\theta) = D_C, \theta_{|D_H} \in \Theta \text{ and } \forall X \in D_C \backslash D_H, X \text{ is free in } \theta\}$$

Restriction and Extension of a Set of Substitutions for a Call. The RESTRG operation expresses a set of substitution Θ in terms of the formal parameters $X_1, \ldots, X_n$ of a call ℓ. The EXTG operation extends a set of substitutions Θ with a set of substitutions Θ_ℓ representing the result of executing a call ℓ on Θ. Let D_Θ be the domain of Θ, $D_\ell = \{X_{i_1}, \ldots, X_{i_n}\}$ the set of variables appearing in ℓ exactly in that order, and $D = \{X_1, \ldots, X_n\}$.

$$\mathrm{RESTRG}(\ell, \Theta) = \{\theta\colon \mathrm{dom}(\theta) = D, \exists\theta' \in \Theta\colon X_j\theta = X_{i_j}\theta' \ (1 \le j \le n)\}$$

$$\begin{aligned}\mathrm{EXTG}(\ell, \Theta, \Theta_\ell) = \{\theta\sigma\colon\ & \theta \in \Theta, \sigma \in \mathrm{IRS}, \theta'\sigma \in \Theta_\ell, \mathrm{dom}(\sigma) \subseteq \mathrm{codom}(\theta'),\\ & (\mathrm{codom}(\theta)\backslash\mathrm{codom}(\theta')) \cap \mathrm{codom}(\sigma) = \varnothing,\\ & \mathrm{dom}(\theta') = D, X_j\theta' = X_{i_j}\theta \ (1 \le j \le n)\}\end{aligned}$$

Sets of concrete tuples

We assume in the following an underlying program P. The semantics of P is captured by a set of concrete tuples of the form $(\Theta_{in}, p, \Theta_{out})$ where Θ_{out} is intended to represent the set of output substitutions obtained be executing $p(X_1, ..., X_n)$ on the set of input substitutions Θ_{in}; $\Theta_{in}, \Theta_{out} \in CS_D$ with $D = \{X_1, ..., X_n\}$. We only consider *functional sets of concrete tuples*, sct, implying that for all (Θ_{in}, p) there exist at most one set Θ_{out} such that $(\Theta_{in}, p, \Theta_{out}) \in$ sct. This tuple is denoted sct(Θ_{in}, p). dom(sct) is the set of pairs (Θ_{in}, p) for which there exists a Θ_{out} such that $(\Theta_{in}, p, \Theta_{out}) \in$ sct. We call *underlying domain* UD the set of pairs (Θ_{in}, p) where p is a predicate symbol of arity n in P and $\Theta_{in} \in CS_D$ where $D = \{X_1, ..., X_n\}$. We also denote SCT the set of all *monotonic* sct, i.e. those satisfying $\Theta_1 \subseteq \Theta_2 \Rightarrow sct(\Theta_1, p) \subseteq sct(\Theta_2, p)$, each time $sct(\Theta_1, p)$ and $sct(\Theta_2, p)$ are defined.

We denote SCTT the set of all total and continuous sct, i.e. those for which any non decreasing chain $\Theta_1 \subseteq \Theta_2 \subseteq ... \subseteq \Theta_n \subseteq ...$ satisfies $sct(\cup_{i=1}^{\infty}\Theta_i, p) = \cup_{i=1}^{\infty}\{sct(\Theta_i ,p)\}$. SCTT is endowed with a structure of complete partial order by defining:

- $\bot = \{(\Theta, p, \emptyset): (\Theta, p) \in UD\}$,
- $sct \le sct' \Leftrightarrow \forall(\Theta, p) \in UD: sct(\Theta, p) \subseteq sct'(\Theta, p)$.

Fixpoint Concrete Semantics

The concrete semantics is defined in terms of three functions and one transformation given at Figure 4.2. We assume an underlying program P and let b and c be respectively a sequence of atoms and a clause using only predicate symbols from P.

The transformation and functions are monotonic and continuous with respect to SCCT and the canonical ordering on the Cartesian product $CS_D \times$ SCCT respectively. Since SCCT is a complete partial order, the semantics of logic programs is the least fixpoint of the transformation TSCT, denoted μ(TSCT). This fixpoint can be shown to be equivalent to SLD-Resolution.

Figure 4.2
A concrete fixpoint semantics.

```
TSCT(sct) = {(Θ, p, Θ'): (Θ, p) ∈ UD and Θ' = Tp(Θ, p, sct)}

Tp(Θ, p, sct) = UNION(Θ₁, ..., Θₙ)
      where      Θᵢ = Tc(Θ, cᵢ, sct)
                 c₁, ..., cₙ are the clauses of p

Tc(Θ, c, sct) = RESTRC(c, Θ')
      where      Θ' = Tb(EXTC(c, Θ), b, sct)
                 b is the body of c

Tb(Θ, ⟨⟩, sct) = Θ
Tb(Θ, ℓ.gs, sct) = Tb(Θ₃, gs, sct)
      where      Θ₃ = EXTG(ℓ, Θ, Θ₂)
                 Θ₂ =  sct(Θ₁, p)        if ℓ is p(...)
                       AI_VAR(Θ₁)        if ℓ is Xᵢ = Xⱼ
                       AI_FUNC(Θ₁, f)    if ℓ is Xᵢ = f(...)
                 Θ₁ = RESTRG(ℓ, Θ)
```

Informally speaking, the function Tp(Θ, p, sct) executes all clauses defining p on substitution β and takes the union of the results. The function Tc executes one clause by extending the substitution, executing the body and restricting the substitution. The function Tb executes the body of a procedure by considering each literal in turn. When the literal is a procedure call, this result is simply picked up in the sat, otherwise the operation AI_VAR or AI_FUNC is executed. Operation RESTRG is used before calling any literal and operation EXTG is performed after each call.

Theorem 1. Let P be a program, $\ell = \mathrm{p}(X_1, \ldots, X_n)$ be a predicate, θ_{in} be a program substitution with $\mathrm{dom}(\theta_{in}) = \{X_1, \ldots, X_n\}$, sct be $\mu(\mathrm{TSCT})$ and $\Theta_{in} = \{\theta \in \mathrm{IPS}: \theta$ and θ_{in} are variant$\}$. The following two statements are true (we assume that SLD-Resolution uses renaming variables belonging to IRS):

- If σ is an answer substitution of SLD-Resolution applied to $P \cup \{ \leftarrow \ell\theta_{in}\}$, then there exists a substitution $\theta_{out} \in \mathrm{sct}(\Theta_{in}, \mathrm{p})$ such that $\theta_{out} = \theta_{in}\sigma$.

- If $\theta_{out} \in \mathrm{sct}(\Theta_{in}, \mathrm{p})$, then there exists an answer substitution σ of SLD-Resolution applied to $P \cup \{ \leftarrow \ell\theta_{in}\}$ such that $\theta_{in}\sigma$ is more general than θ_{out}, i.e. $\theta_{out} = \theta_{in}\sigma\sigma'$ for some $\sigma' \in \mathrm{IRS}$.

Abstract Semantics

Abstract tuples

The abstract fixpoint semantics is defined in the same way as the concrete semantics, with abstract tuples and abstract operations replacing concrete ones. Abstract tuples are tuples of the form $(\beta_{in}, \mathrm{p}, \beta_{out})$ where p is a predicate symbol of arity n and β_{in}, β_{out} are abstract substitutions on variables $\{X_1, \ldots, X_n\}$. It has the informal interpretation that "*the execution of* $\mathrm{p}(X_1, \ldots, X_n)\theta$, θ *being a substitution satisfying the property expressed by* β_{in}, *will produce substitutions* $\theta_1, \ldots, \theta_n$ *which all satisfy the property expressed by* β_{out}".

The purpose of the abstract semantics is to define precisely the relationships between β_{in} and β_{out} for each predicate symbol p in the program. To define the semantics, we denote by UD the underlying domain of the program, that is the set of pairs (β_{in}, p) where p is a predicate symbol of arity n and β_{in} is an abstract substitution on variables $\{X_1, \ldots, X_n\}$. We also denote AS as the set of abstract substitutions. The abstract substitutions on variable $D = \{X_1, \ldots, X_n\}$ are elements of a complete partial order $(AS_D, \leq)$, which is *unique* for each set of variables.

To each of the concrete operations, we also associate an abstract operation with the same name and signature except that abstract substitutions replace complete sets of substitutions. Moreover, all abstract operations should be consistent versions of the concrete operations to ensure correctness of the abstract semantics. If all abstract operations are consistent, then any fixpoint of the abstract transformation is consistent. Finally all abstract operations are also required to be continuous.

Abstract operations

EXTC(c, β) where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and c is a clause containing variables $\{X_1, \ldots, X_m\}$ $(m \geq n)$. This operation returns the abstract substitution obtained by extending β to accommodate the new free variables of the clause. It is used at the beginning of a clause to include the variables in the body not present in the head. The specification is given hereafter.

Let $D = \{X_1, ..., X_n\}$ be the set of variables occurring in the head of c, $dom(\beta) = D$ and $D' = \{X_1, ..., X_{n+k}\}$ be the set of all variables in c. $EXTC(\beta, c)$ produces a substitution $\beta' = (sv', mo', frm', ps')$ such that

$$\begin{aligned} Cc(\beta') = \{\theta:\ & dom(\theta) = D' \\ & \wedge \theta_{|D} \in Cc(\beta) \\ & \wedge X_{n+1}\theta, ..., X_{n+k}\theta \text{ are distinct renaming variables} \\ & \wedge X_{n+j} \notin codom(\theta_{|D})\ (1 \leq j \leq k)\}. \end{aligned}$$

RESTRC(c, β) where β is an abstract substitution on $\{X_1, ..., X_m\}$ and $\{X_1, ..., X_n\}$ ($n \leq m$) are the head variables of clause c. This operation returns the abstract substitution obtained by projecting β on variables $\{X_1, ..., X_n\}$. It is used after the execution of a clause to restrict the substitution to the head variables only. The specification is given hereafter.

Let D be the domain of β and D' be the set of variables in the head of c ($D \subseteq D'$). $RESTRC(\beta, c)$ produces a substitution $\beta' = (sv', mo', frm', ps')$ such that

$$Cc(\beta') = \{\theta_{|D}: \theta \in Cc(\beta)\}.$$

UNION$\{\beta_1,...,\beta_n\}$ where $\beta_1, ...,\beta_n$ are abstract substitutions from the same cpo. This operation returns an abstract substitution representing all the substitutions satisfying at least one β_i. It is used to compute the output of a procedure given the outputs of its clauses. The abstract semantics makes use of this n-ary operation which is derived from a binary abstract operation as follows:

$$UNION(\beta_1,...,\beta_n) = UNION(...UNION(UNION(\bot,\beta_1),\beta_2)..., \beta_n)$$

where the operation $UNION(\beta_1,\beta_2)$ is implemented the same way as the operation $LUB(\beta_1,\beta_2)$. This implementation is also likely to be the most accurate one . These two operations are specified hereafter.

Let β_1,β_2 be two abstract substitutions such that $dom(\beta_1) = dom(\beta_2) = D$. $UNION(\beta_1,\beta_2)$ produces an abstract substitution β such that

$$dom(\beta) = D \wedge Cc(\beta_1) \cup Cc(\beta_2) \subseteq Cc(\beta).$$

Under the same hypotheses, $LUB(\beta_1,\beta_2)$ produces an abstract substitution β such that

$$\beta_1, \beta_2 \leq \beta \wedge \forall \beta' \in AS_{|D}: (\beta_1, \beta_2 \leq \beta') \Rightarrow \beta \leq \beta'.$$

RESTRG(ℓ, β) where β is an abstract substitution on $D = \{X_1, ..., X_n\}$ and ℓ is a literal $p(X_{i_1}, ..., X_{i_m})$ (or $X_{i_1} = X_{i_2}$ or $X_{i_1} = f(X_{i_2}, ..., X_{i_m})$). This operation returns the abstract substitution obtained by

1. projecting β on $\{X_{i_1}, ..., X_{i_m}\}$ obtaining β';
2. expressing β' in terms of $\{X_1, ..., X_m\}$ by mapping X_{i_k} to X_k.

It is used before the execution of a literal in the body of a clause. The resulting substitution is expressed in terms of $\{X_1, ..., X_m\}$, i.e. as an input abstract substitutions for p/m. The specification of this operation can be deduced from the specification of the RESTRC operation.

EXTG(ℓ, β, β') where β is an abstract substitution on $D = \{X_1, ..., X_n\}$, the variables of the clause where ℓ appears, ℓ is a literal $p(X_{i_1}, ..., X_{i_m})$ (or $X_{i_1} = X_{i_2}$ or $X_{i_1} =$

$f(X_{i_2}, \ldots, X_{i_m})$) with $\{X_{i_1}, \ldots, X_{i_m}\} \subseteq D$ and β' is an abstract substitution on $\{X_1, \ldots, X_m\}$ representing the result of $p(X_1, \ldots, X_m)\beta''$ where $\beta'' = \mathtt{RESTRG}(\ell, \beta)$. This operation returns the abstract substitution obtained by extending β to take into account the result β' of the literal ℓ. It is used after the execution of a literal to propagate the results of the literal to all the variables of the clause. The specification is given hereafter.

Let $X_{i_1}, \ldots, X_{i_k}$ be the sequence of variables occurring in the literal ℓ. Let us define $D = \{X_{i_1}, \ldots, X_{i_k}\}$. Let β_1 be an abstract substitution such that $\{X_{i_1}, \ldots, X_{i_k}\} \subseteq \{X_1, \ldots, X_m\} = \mathtt{dom}(\beta_1)$. Let β_2 be an abstract substitution such that $\{X_1, \ldots, X_k\} = \mathtt{dom}(\beta_2)$. Let θ_1 and θ_2 such that $\mathtt{dom}(\theta_i) = \mathtt{dom}(\beta_i)$ $(i = 1, 2)$. Let σ be a renaming substitution. The followings holds:

$$\left.\begin{array}{l} \theta_1 \in \mathtt{Cc}(\beta_1) \wedge \theta_2 \in \mathtt{Cc}(\beta_2) \\ (\forall j\colon 1 \leq j \leq k,\ X_j\theta_2 = X_{i_j}\theta_1\sigma) \\ \mathtt{dom}(\sigma) \subseteq \mathtt{codom}(\theta_{1|D}) \\ (\mathtt{codom}(\theta_1) \setminus \mathtt{codom}(\theta_{1|D})) \cap \mathtt{codom}(\sigma) = \emptyset \end{array}\right\} \Rightarrow \theta_1\sigma \in \mathtt{Cc}(\beta').$$

AI_VAR(β)=β' where β is an abstract substitution on $\{X_1, X_2\}$. This operation returns the abstract substitution obtained from β by unifying X_1 and X_2. It is used for literals of the form $X_i = X_j$ in normalized programs. The specification is given hereafter.

Let β such that $\mathtt{dom}(\beta) = \{X_1, X_2\}$. The following holds for any program substitution θ and any renaming substitution σ:

$$\theta \in \mathtt{Cc}(\beta) \wedge \sigma \in \mathtt{mgu}(X_1\theta, X_2\theta) \Rightarrow \theta\sigma \in \mathtt{Cc}(\beta').$$

AI_FUNC(β, *f*)=β' where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and *f* is a function symbol of arity $n-1$. This operation returns the abstract substitution obtained from β by unifying X_1 and $f(X_2, \ldots, X_n)$. It is used for literals of the form $x_{j_1} = f(x_{j_2}, \ldots, x_{j_n})$ in normalized programs. The specification is given hereafter.

Let β such that $\mathtt{dom}(\beta) = \{X_1, \ldots, X_n\}$. The following holds for any program substitution θ and any renaming substitution σ:

$$\theta \in \mathtt{Cc}(\beta) \wedge \sigma \in \mathtt{mgu}(X_1\theta, f(X_2, \ldots, X_n)\theta) \Rightarrow \theta\sigma \in \mathtt{Cc}(\beta').$$

Abstract semantics

The abstract semantics is defined as the least fixpoint of the transformation TSAT; it is denoted μ(TSAT). It is simply the same transformation as TSCT but working with abstract substitutions instead of complete sets of concrete substitutions. The transformation is recalled at Figure 4.3.

Figure 4.3
An abstract fixpoint semantics.

```
TSAT(sat) = {(β, p, β'): (β, p) ∈ UD and β' = Tp(β, p, sat)}

Tp(β, p, sat) = UNION(β_1, ..., β_n)
      where      β_i = Tc(β, c_i, sat)
                 c_1, ..., c_n are the clauses of p

Tc(β, c, sat) = RESTRC(c, β')
      where      β' = Tb(EXTC(c, β), b, sat)
                 b is the body of c

Tb(β, ⟨⟩, sat) = β.
Tb(β, ℓ.gs, sat) = Tb(β_3, gs, sat)
      where      β_3 = EXTG(ℓ, β, β_2)
                 β_2 =  sat(β_1, p)         if ℓ is p(...)
                        AI_VAR(β_1)         if ℓ is X_i = X_j
                        AI_FUNC(β_1, f)     if ℓ is X_i = f(...)
                 β_1 = RESTRG(ℓ, β)
```

sat is a set of abstract tuples; it is functional — i.e. there exists at most one β_{out} for each pair (β_{in}, p) such that (β_{in}, p, β_{out}) $\in$ sat, β_{out} is denoted sat(β_{in}, p) — total and monotonic. The set of functional, total and monotonic sets of abstract tuples, SAT, is endowed with the structure of a complete partial order by defining:

- $\perp = \{(\beta, p, \perp_D) \mid (\beta, p) \in UD$ and $\perp_D$ is the smallest element in $(AS_D, \leq)\}$;
- $sat \leq sat' \Leftrightarrow \forall(\beta, p) \in UD: sat(\beta, p) \leq sat'(\beta,p)$.

Abstract domains

An *abstract domain* is the association of a complete partial order (AS_D, $\leq$) with each finite set of variables D. Elements of AS_D are called *abstract substitutions*. The correspondence between abstract and concrete domains is established by a *concretization function* $Cc: AS_D \rightarrow CS_D$.

We use sat to denote the set of abstract tuples; we also denote SAT the set of all functional and monotonic sat, SATT the subset of SAT whose elements are total and continuous. As the generic algorithm works on partial sat, we define an ordering on SAT. Let $sat_1, sat_2 \in SAT$, $sat_1 \leq sat_2 \Leftrightarrow \forall(\beta, p) \in dom(sat_1), (\beta, p) \in dom(sat_2) \wedge sat_1(\beta, p) \leq sat_2(\beta, p)$.

Theorem 2. Let P be a program, TSAT and TSCT the associated transformations, sat = μ(TSAT) and sct = μ(TSCT). Let p be a predicate and D = $\{X_1, \ldots, X_n\}$ where n is the arity of p. For each $\beta \in AS_D$: $sct(Cc(\beta),p) \subseteq Cc(sat(\beta,p))$.

We give here an informal description of the abstract domain, that is we will try to show its expressiveness; a comprehensive discussion of the abstract domain is given in [LCVH94].

Abstract substitutions

The key concept in the representation of the substitutions in this domain is the notion of subterm. Given a substitution on a set of variables, an abstract substitution associates the following information *with each subterm* appearing in the substitution:

- its *mode* taken from {$\perp$, ground, var, ngv, novar, gv, noground, any}[3];
- its *pattern* which specifies the main functor as well as the subterms which are its arguments;
- its possible *sharing* with other subterms.

The pattern is optional; if it is omitted then the pattern is said to be *undefined*. Besides the above information, each variable in the domain of the substitution is associated with one of the subterms. This information even enables to express that two arguments have the *same value* — and hence that two variables are bound together — by associating two arguments with the same subterm. To identify the subterms unambiguously, an index is associated with each of them. If there are n subterms, indices from 1 through n are used; let I_n be the set of indices {1,...,n}. For instance, the substitution

$$\{X_1 \leftarrow [a,b], X_2 \leftarrow [c,d], X_3 \leftarrow Ls\}$$

will have 7 subterms. The association of indices to them could be for instance

$$\{(1,[a,b]), (2,a), (3,b), (4,[c,d]), (5,c), (6,d), (7,Ls)\}.$$

As mentioned previously, each index is associated with a mode taken from {$\perp$, ground, var, ngv, novar, gv, noground, any}. In the above example, we have the following mode associations:

$$\{(1,ground), (2,ground), (3,ground), (4,ground), (5,ground), (6,ground), (7,var)\}.$$

Formally, the mode component is a total function mo: $I_n \rightarrow$ MODES from the set of indices to the set of modes which associates to each subterm of a substitution a mode from the set MODES. The set MODES satisfy the ordering depicted in the HASSE diagram at Figure 4.4. Each node in the diagram is a mode. An oriented vertex between the nodes n_1 and n_2 denotes the ordering relation between modes, that is $mo(n_1) > mo(n_2)$. The ordering relation between modes is simply the transitive closure of the ordering described in the diagram. Moreover, the semantics of mo is given by the concretization function:

$$Cc(mo) = \{(t_1, \ldots, t_n) \mid \forall i: 1 \leq i \leq n, t_i \in Cc(mo(i))\}.$$

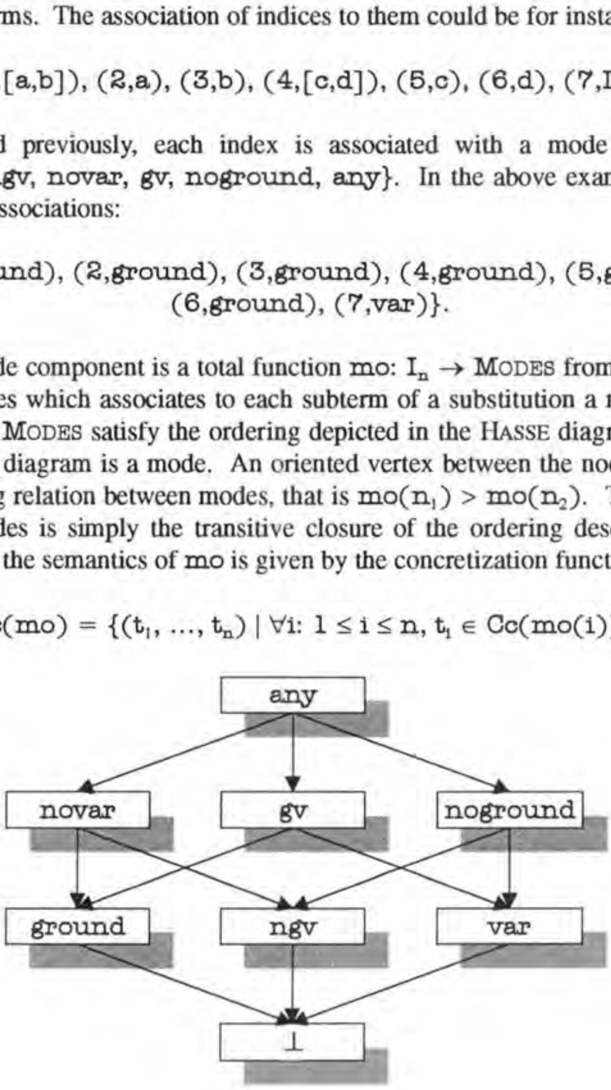


Figure 4.4 The ordering of modes as an HASSE diagram.

[3] ngv stands for "nor variable nor ground", gv stands for "ground or variable" and any is the top element.

The pattern component possibly assigns to an index an expression $f(i_1, \ldots, i_n)$ where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. In our example, the pattern component will make the following associations:

{(1,.(2,3)), (2,a), (3,b), (4,.(5,6)), (5,c), (6,d)}

where .(head_of_list,tail_of_list) is the list constructor.

Formally, the pattern component is a partial function frm: $I_n \not\rightarrow \mathbb{F}_m$ from the set of indices to the set of all patterns on I_n, i.e. elements of the form $f(i_1, \ldots, i_m)$ where f is a functor symbol and $i_1, \ldots, i_m \in I_n$. An undefined pattern is denoted frm(i) = undef. The semantics of frm is given by:

$$Cc(frm) = \{(t_1, \ldots, t_n) \mid \forall i, i_1, \ldots, i_m \in I_n: frm(i)=f(i_1, \ldots, i_m) \Rightarrow t_i=f(t_{i_1}, \ldots, t_{i_m})\}$$

The same value component is a total function sv: $D \rightarrow I_n$ from the substitution domain to the set of indices, $D = \{X_1, \ldots, X_n\}$. The concretization function of sv is:

$$Cc(sv) = \{\theta \mid dom(\theta) = D \wedge \forall X_i, X_j \in D: sv(X_i) = sv(X_j) \Rightarrow X_i\theta = X_j\theta\}.$$

Finally the sharing component specifies which indices, not associated with a pattern, may possibly share variables. We only restrict our attention to indices without pattern since the patterns already express some sharing information and we do not want to introduce inconsistencies between the components. The actual sharing relation can de computed from the two previous components. In the example, the only sharing is the couple (7,7) expressing that variable Ls shares a variable with itself.

Formally, the sharing component specifies the possible sharing of variables between undefined pattern subterms. The sharing component is a binary and symmetrical relation ps: $I_n \times I_n$. Therefore, ps satisfy the property:

$$\forall i, j \in I_n: ps(i, j) \Rightarrow frm(i) = undef = frm(j).$$

And the semantics of ps for a given pattern component is given by the following concretization function — var(t) denotes the set of variables in the syntactical object t:

$$Cc(ps, frm) = \{(t_1, \ldots, t_n): \forall i, j \in I_n: frm(i) = frm(j) = undef$$
$$\wedge\ var(t_i) \cap var(t_j) \neq \emptyset \Rightarrow ps(i,j)\}.$$

In order to clarify the concepts, a more appealing representation is given for the predicate append instantiated with the above substitution:

```
append(ground(1): .(ground(2):a, ground(3):b),
       ground(4): .(ground(5):c, ground(6):d),
       var(7))
```

together with the sharing information {(7,7)}. In the above representation, each argument is associated with a mode, with an index (between parenthesis) and with an abstract subterm after the colon.

We are now allowed to rewrite the definition of an abstract substitution as a 4-tuple (sv, mo, frm, ps). In brief, the abstract domain contains the four components, mode, pattern, same value and sharing. Formally, the meaning of an abstract substitution β ::= (sv, mo, frm, ps) is given by the following concretization function:

$$\mathtt{Cc}(\beta) = \{\theta\colon \mathtt{dom}(\theta) = \mathtt{D} \wedge \exists(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{p}) \in \mathtt{Cc(mo)} \cap \mathtt{Cc(frm)} \cap \mathtt{Cc(ps, frm)}\colon \forall \mathtt{x} \in \mathtt{D},\ \mathtt{x}\theta = \mathtt{t}_{\mathtt{sv(x)}}\}$$

where D is a finite set of program variables.

Ordering on abstract substitutions

It remains to define the ordering relation on substitutions. Consider two substitutions β_1, $\beta_2 \in \mathtt{AS}_{|\mathtt{D}}$ and assume in the following that $\mathtt{sv}_1$, $\mathtt{mo}_1$, $\mathtt{frm}_1$, $\mathtt{ps}_1$ are the components of a substitution β_1, $\mathtt{p}_1$ is the number of indices in the domains of $\mathtt{mo}_1$, $\mathtt{frm}_1$, $\mathtt{ps}_1$ and $\mathtt{m}_1$ is the number of indices in the codomain of $\mathtt{sv}_1$ — whereas the domain of the substitution is implicitly defined by the substitution. Conceptually, $\beta_1 \leq \beta_2$ should hold iff $\mathtt{Cc}(\beta_1) \subseteq \mathtt{Cc}(\beta_2)$. But for implementation purpose, we need a syntactic definition. To provide such a definition, it is necessary to use a function that relates the indices of β_1 and β_2 since the same indices do not necessarily refer to corresponding terms. This leads to the following definition where the function t is responsible for mapping the indices of β_2 into those of β_1.

$\beta_1 \leq \beta_2$ iff there exists a function $\mathtt{t}\colon \mathtt{I}_{\mathtt{p}_2} \to \mathtt{I}_{\mathtt{p}_1}$ satisfying:

- $\forall \mathtt{x} \in \mathtt{D}\colon \mathtt{sv}_1(\mathtt{x}) = \mathtt{t}(\mathtt{sv}_2(\mathtt{x}))$;
- $\forall \mathtt{i} \in \mathtt{I}_{\mathtt{p}_2}\colon \mathtt{mo}_1(\mathtt{t}(\mathtt{i})) \leq \mathtt{mo}_2(\mathtt{i})$;
- $\forall \mathtt{i}, \mathtt{i}_1, \ldots, \mathtt{i}_\mathtt{q} \in \mathtt{I}_{\mathtt{p}_2}\colon \mathtt{frm}_2(\mathtt{i}) = f(\mathtt{i}_1, \ldots, \mathtt{i}_\mathtt{q}) \Rightarrow \mathtt{frm}_1(\mathtt{t}(\mathtt{i})) = f(\mathtt{t}(\mathtt{i}_1), \ldots, \mathtt{t}(\mathtt{i}_\mathtt{q}))$;
- $\forall \mathtt{i}, \mathtt{j} \in \mathtt{I}_{\mathtt{p}_2}\colon \mathtt{frm}_2(\mathtt{i}) = \mathtt{frm}_2(\mathtt{j}) = \mathtt{undef}\colon \mathtt{ps}_1(\mathtt{t}(\mathtt{i}), \mathtt{t}(\mathtt{j})) \Rightarrow \mathtt{ps}_2(\mathtt{i}, \mathtt{j})$.

Generic Abstract Interpretation Algorithm

The overall approach is first presented. The next section highlights two high-level operations to manipulate the set of abstract tuples. The third section discusses the dependency graph structure and its specific operators. The last section exposes the algorithm.

Definitions

The following definitions aim at characterizing the set of abstract tuples needed to compute $\mathtt{Tp}(\beta, \mathtt{p}, \mathtt{sat})$. A more formal approach is brought to light in [LCMVH90].

Let $\mathtt{D} \subseteq \mathtt{UD}$ and sat be a set of abstract tuples. The restriction of sat to D, denoted $\mathtt{sat}_{|\mathtt{D}}$, is the set $\{(\beta, \mathtt{p}, \mathtt{sat}(\beta, \mathtt{p}))\colon (\beta, \mathtt{p}) \in \mathtt{D} \cap \mathtt{dom(sat)}\}$. Let β be an abstract substitution and p be a predicate symbol. $(\beta, \mathtt{p})$ is *based* in sat iff, informally, for any total set of abstract tuples sat' such that $\mathtt{sat} \subseteq \mathtt{sat'}$, the computation of $\mathtt{Tp}(\beta, \mathtt{p}, \mathtt{sat'})$ does not require values of sat' not belonging to sat. Hence the notation $\mathtt{Tp}(\mathtt{b}, \mathtt{p}, \mathtt{sat})$ can be extended to such partial set of abstract tuples. We denote by $\mathtt{base}(\mathtt{b}, \mathtt{p}, \mathtt{sat})$ the smallest set D such that $(\beta, \mathtt{p})$ is based in $\mathtt{sat}_{|\mathtt{D}}$. This set is defined iff $(\beta, \mathtt{p})$ is based in sat.

Let β be an abstract substitution, p be a predicate symbol and sat a partial set of abstract tuples. $(\beta, \mathtt{p})$ is *founded* in sat iff $\exists \mathtt{D}\colon (\beta, \mathtt{p}) \in \mathtt{D} \wedge \forall(\alpha, \mathtt{q}) \in \mathtt{D}\colon (\alpha, \mathtt{q})$ is based in $\mathtt{sat}_{|\mathtt{D}}$. We denote by $\mathtt{foundation}(\beta, \mathtt{p}, \mathtt{sat})$ the smallest set D such that $(\beta, \mathtt{p})$ is founded in $\mathtt{sat}_{|\mathtt{D}}$, when it exists.

Overview of the approach

A straightforward approach to compute the output substitution β_{out} for a pair (β_{in}, p) consists in computing the least fixpoint of TSAT — μ(TSAT) for short — entirely and picking up the value sat(β_{in}, p) in the fixpoint. This is possible using a bottom-up approach provided that some restrictions are imposed on the abstract domain to guarantee termination. The main drawback of this approach is that many elements in μ(TSAT) are not relevant to the computation of β_{out} and hence most of the computation is unnecessary.

The approach described through our algorithm does its best to focus on the relevant elements in computing a postfixpoint of μ(TSAT). Of course, it's impossible to know a priori which subset of the postfixpoint will be strictly necessary. The algorithm was therefore tailored to avoid as much as possible the irrelevant computations; its purpose is to converge towards a partial set of abstract tuples sat including $(\beta_{in}, p, \beta_{out}) \in \mu(\text{TSAT})$.

To achieve its goal, the algorithm computes a series of lower approximations $\text{sat}_0, \ldots, \text{sat}_n$ such that $\text{sat}_i < \text{sat}_{i+1}$ and such that sat_n contains (β_{in}, p, B_{out}). The first approximation is the empty set. The algorithm then moves from one partial set to another by selecting:

- an element (α_{in}, q) which is not present in the sat but needs to be computed, or
- an element (α_{in}, q) whose value sat(α_{in}, q) can be improved because the values of some elements upon which it is depending have been updated.

It is important to notice at this point that within this approach we never fully apply TSAT to a complete sat but we are rather constantly improving the current sat and working with the most accurate one.

Within this framework, there are still many decisions to consider, as detection of termination and the choice of the element to work on. Here is an informal description of the way the algorithm works. Given an input pair (β_{in}, p) it executes the function Tp of the abstract semantics. At some point, the computation may need the value of (α_{in}, q) which may not be defined or is just (lower) approximated at this stage of the computation. In this situation, the algorithm starts a new subcomputation to obtain the value of (α_{in}, q) or a lower approximation of it. This computation is carried out in the same way as the primary one except in the case where a value for (β_{in}, p) is needed. In that case, instead of starting a new computation (that may generate an infinite loop), the algorithm simply looks up the current value of (β_{in}, p) in the sat. The execution of the initial pair (β_{in}, p) is only resumed once the computation of (α_{in}, q) is completed. Note that if the computation of (α_{in}, q) has required the value of (β_{in}, p) then its resulting substitution may only be lower approximated (with respect to its fixpoint value) and hence (α_{in}, q) will have to be reconsidered if the value of (β_{in}, p) is updated. In the algorithm, a dependency graph is used to detect when an element needs to be reconsidered.

More formally, the series of approximation is such that:

- $\text{sat}_0 = \{(\beta_{in}, p, \perp)\}$;
- sat_{i+1} is obtained from sat_i by computing a new tuple $(\alpha_{in}, q, \alpha_{out})$ such that either (α_{in}, q) is not based in sat_i or $\text{Tp}(\alpha_{in}, q, \text{sat}_i) \leq \text{sat}_i(\alpha_{in}, q)$ does not hold;
- sat_n such that $\forall(\alpha_{in}, q) \in \text{sat}_n$, $\text{Tp}(\alpha_{in}, q, \text{sat}_n) \leq \text{sat}_n(\alpha_{in}, q)$ and (α_{in}, q) is based in sat_n.

Sets of abstract tuples manipulations

The *abstract semantics* does not need operations on sets of abstract tuples besides the ability to pick up a value in one of them. The algorithm however needs to construct a subset of the fixpoint containing the solution of the query. Hence it needs a number of operations on sets of abstract tuples.

There are mainly two operations that need to be defined: EXTEND and ADJUST. The purpose of the first operation is to extend a set of abstract tuples with a brand new element, while ADJUST is intended to update the result of a particular pair (β_{in}, p). Let us specify these two operations:

- **EXTEND**$(\beta_{in}, p, sat) = sat'$. Given a set of abstract tuples sat and a kind of access key to this sat constituted by a predicate symbol p and an abstract substitution β_{in}, it returns a new set of abstract tuples sat' containing (β_{in}, p) in its domain. sat was supposed not to already contain the pair (β_{in}, p). Actually, this operation inserts in sat a new pair with its default approximation, that is (β_{in}, p, β) where $\beta = lub\{sat(\beta',p): \beta' \leq \beta_{in} \wedge (\beta',p) \in dom(sat)\}$
- **ADJUST**$(\beta_{in}, p, \beta_{out}, sat) = sat'$. Given the substitution β_{out} which represents a new result computed for the pair (β_{in}, p), it returns a new set of abstract tuples sat' which is the sat updated with this new result. Actually, this operation returns: $sat' \cup \{(\beta, p, lub\{\beta_{out}, sat(\beta, p)\}): \beta \geq \beta_{in} \wedge (\beta, p) \in dom(sat)\}$ where $sat' = sat \setminus \{(\beta, p, sat(\beta, p)): \beta \geq \beta_{in} \wedge (\beta, p) \in dom(sat)\}$.

Note that a slightly more general version of ADJUST is used in the algorithm, it returns in addition to the new set of abstract tuples the set of pairs (β_{in}, p) whose values have been updated.

Procedure call dependencies

The purpose of a dependency graph is mainly the detection of redundant computations. Such a computation may occur in a variety of situations. For instance, the value of a pair (α_{in}, p) may have reached its definitive value — i.e. the value of $(\alpha_{in}, p) \in \mu(TSAT)$ — and hence subsequent considerations of (α_{in}, p) should only look up its value instead of starting a new subcomputation. Another efficiency concern is the mutually recursive programs. For those programs, it would be interesting that the algorithm reconsiders a pair (α_{in}, p) only when some elements upon which it depends have been updated.

Therefore, keeping track of the procedure call dependencies may substantially improve the efficiency on some program classes. The algorithm includes obviously specific data structure to handle procedure call dependencies. However, we only introduce the basic notions.

Definition. A *dependency graph* is a partial function $dp: UD \not\to \mathcal{P}(UD)$, i.e. a set of tuples of the form $\langle(\beta_{in}, p), lt\rangle$ where lt is a set $\{(\alpha_1, q_1), \ldots, (\alpha_n, q_n)\}$ $(n \geq 0)$ such that for each (β_{in}, p) there exists at most one lt such that $\langle(\beta_{in}, p), lt\rangle \in dp$.

We denote by $dp(\beta_{in}, p)$ the set lt such that $\langle(\beta_{in}, p), lt\rangle \in dp$ if it exists. We also denote by $dom(dp)$ the set of all (β_{in}, p) such that $\langle(\beta_{in}, p), lt\rangle \in dp$ and by $codom(dp)$ the set of all (α_{in}, q) such that there exists a tuple $\langle(\beta_{in}, p), lt\rangle \in dp$ satisfying $(\alpha_{in}, q) \in lt$.

The basic intuition is that $\mathtt{dp}(\beta_{in}, \mathtt{p})$ represents at some point the set of pairs which $(\beta_{in}, \mathtt{p})$ depends directly upon. This is why we also need to define the transitive closure of the dependency graph.

Definition. Let dp be a dependency graph and assume that $(\beta_{in}, \mathtt{p}) \in \mathtt{dom(dp)}$. The set $\mathtt{trans\_dp}(\beta_{in}, \mathtt{p}, \mathtt{dp})$ is the smallest subset of $\mathtt{codom(dp)}$ closed by the following two rules:

1. if $(\alpha_{in}, \mathtt{q}) \in \mathtt{dp}(\beta_{in}, \mathtt{p})$ then $(\alpha_{in}, \mathtt{q}) \in \mathtt{trans\_dp}(\beta_{in}, \mathtt{p}, \mathtt{dp})$;

2. if $(\alpha_{in}, \mathtt{q}) \in \mathtt{dp}(\beta_{in}, \mathtt{p})$, $(\alpha_{in}, \mathtt{q}) \in \mathtt{dom(dp)} \wedge (\alpha'_{in}, \mathtt{q'}) \in \mathtt{trans\_dp}(\alpha_{in}, \mathtt{q}, \mathtt{dp})$ then $(\alpha'_{in}, \mathtt{q'}) \in \mathtt{trans\_dp}(\beta_{in}, \mathtt{p}, \mathtt{dp})$.

$\mathtt{trans\_dp}(\beta_{in}, \mathtt{p}, \mathtt{dp})$ represents all the pairs which if updated would require reconsidering $(\beta_{in}, \mathtt{p})$. Hence $(\beta_{in}, \mathtt{p})$ will not be reconsidered unless one of these pairs is updated.

The main intuition is that the algorithm makes sure that the elements $(\beta_{in}, \mathtt{p})$ needing to be reconsidered are such that $(\beta_{in}, \mathtt{p}) \notin \mathtt{dom(dp)}$. Conversely, elements of $\mathtt{dom(dp)}$ do not require reconsideration. It's now possible to specify the last three operations needed to expose the algorithm.

- **EXT_DP**$(\beta_{in}, \mathtt{p}, \mathtt{dp})$: extends the domain of the dependency graph by inserting a new tuple $\langle(\beta_{in}, \mathtt{p}), \emptyset\rangle$ in dp;

- **ADD_DP**$(\beta_{in}, \mathtt{p}, \alpha_{in}, \mathtt{q}, \mathtt{dp})$: updates dp to include the dependency of $(\beta_{in}, \mathtt{p})$ upon $(\alpha_{in}, \mathtt{q})$, i.e. after this operation $(\alpha_{in}, \mathtt{q}) \in \mathtt{dp}(\beta_{in}, \mathtt{p})$;

- **REMOVE_DP**$(\{(\alpha^1_{in}, \mathtt{q}^1), \ldots, (\alpha^n_{in}, \mathtt{q}^n)\}, \mathtt{dp})$: removes from the dependency graph all elements $\langle(\beta_{in}, \mathtt{p}), \mathtt{lt}\rangle \in \mathtt{dp}$ as soon as one element they depend upon has been updated, i.e. removes all elements $\langle(\beta_{in}, \mathtt{p}), \mathtt{lt}\rangle \in \mathtt{dp}$ for which there is a $(\alpha^i_{in}, \mathtt{q}^i) \in \mathtt{trans\_dp}(\beta_{in}, \mathtt{p}, \mathtt{dp})$.

Let us schematically summarize the concepts of the procedure calls dependency graph, let us depict an imaginary $\mathtt{dp}(\beta_1, \mathtt{p})$:

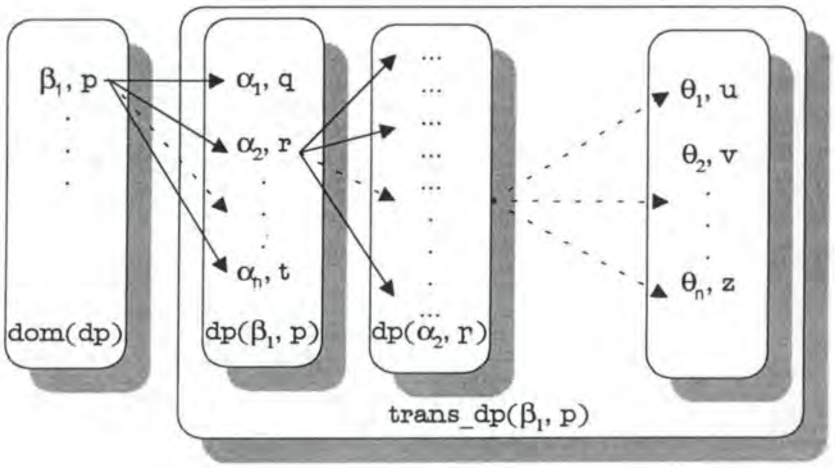


Figure 4.5 Schematic view of the dependency graph.

Generic abstract interpretation algorithm

Given an input substitution β_{in} and a predicate symbol p, the top-level procedure **solve** returns the final dependency graph and the set of abstract tuples sat containing (β_{in}, p, β_{out}) $\in \mu$(TSAT). From these results, it is straightforward to compute the set of pairs (α, q) used by (β_{in}, p), their values in the postfixpoint as well as the abstract input/output substitutions at any program point. A formal characterization of such a postprocessing algorithm is described in [LCMVH91].

The set suspended received by **solve_call** contains all pairs (α, q) for which a subcomputation has been initiated and not completed yet. The procedure considers or reconsiders the pair (β_{in}, p) and updates sat and dp accordingly. The core of the procedure is only executed when (β_{in}, p) is not suspended and not in the domain of the dependency graph. If (β_{in}, p) is suspended, no subcomputation should be initiated. If (β_{in}, p) is in the domain of the dependency graph, this means that none of the elements upon which it is depending have been updated. Otherwise, a new computation with (β_{in}, p) is started. This subcomputation may extend sat if it is the first time (β_{in}, p) is considered.

The core of the procedure is a repeat-until loop which computes the lower approximation of (β_{in}, p) given the elements of the suspended set. Local convergence is reached when (β_{in}, p) belongs to the domain of the dependency graph, meaning that dp(β_{in}, p) = base(β_{in},p). One iteration of the loop amounts to execute each of the clauses defining the procedure p and computing the union of the results. If the result produced is greater or not comparable to the current value of sat(β_{in}, p), then the set of abstract tuples is updated by the operation ADJUST. The dependency graph is also updated accordingly by removing all elements which depend (directly or indirectly) on (β_{in}, p). Note that the calls to the solve_clause are done with an extended suspended set since a subcomputation has been started for (β_{in}, p). Note also, that before calling solve_clause, the dependency graph has been brought up to date to include (β_{in}, p) — which is guaranteed not to be in the domain of the dependency graph before that update. (β_{in}, p) can be removed from the domain of the dependency graph during the computation of the loop if a pair it is depending upon is updated. The very first statement of solve_call, the widening operation, is explained later on.

solve_clause executes a single clause on an input pair and returns an abstract substitution representing the execution of the clause on that input pair. The procedure first extends the input substitution with the variables occurring in the body of the clause. Afterwards it executes the body of the clause further decomposed in literals ℓ_i and terminates by restricting the resulting substitution to the variables occurring in the head of the clause. The execution of a literal consists in choosing the right operation to perform accordingly to its form; this requires three main steps.

- The computation of an abstract substitution representing all the concrete substitutions of the extended input substitution, β_{ext}, restricted to the variables occurring in the literal: this is done by the operation RESTRG, giving β_{aux}.
- The execution of the literal on β_{aux}. If the literal is concerned with unification, the operation AI_VAR or AI_FUNC is performed, depending on the form of the literal. If it is a sub-goal then procedure solve_call is recursively called and the result is afterwards retrieved in sat. Moreover, if (β_{in}, p) is in the domain of the dependency graph it is necessary to add a new dependency; otherwise it means that (β_{in}, p) needs to be reconsidered anyway and no dependency needs to be recorded.
- The propagation of the result of the literal to the variables occurring in the body of the clause: this is why EXTG is made for.

```
procedure solve (in $\beta_{in}$, p; out sat, dp)
begin
    sat:= $\varnothing$;
    dp:= $\varnothing$;
    solve_call($\beta_{in}$, p, $\varnothing$, sat, dp)
end;


procedure solve_call(in $\beta_{in}$, p, suspended; inout sat, dp)
begin
    $\beta_{in}$:= WIDEN($\beta_{in}$, p, suspended);
    if ($\beta_{in}$, p) $\notin$ (dom(dp) $\cup$ suspended) then begin
        if ($\beta_{in}$, p) $\notin$ dom(sat) then sat:= EXTEND($\beta_{in}$, p, sat);
        repeat
            $\beta_{out}$:= $\bot$;
            EXT_DP($\beta_{in}$, p, dp);
            for i:=1 to n with proc(p) being $c_1$, $c_2$, ..., $c_n$ in p do begin
                solve_clause($\beta_{in}$, p, $c_i$, suspended $\cup$ {($\beta_{in}$, p)}, $\beta_{aux}$, sat, dp);
                $\beta_{out}$:= UNION($\beta_{out}$, $\beta_{aux}$)
            end
            (sat, modified):= ADJUST($\beta_{in}$, p, $\beta_{out}$, sat);
            REMOVE_DP(modified, dp)
        until ($\beta_{in}$, p) $\in$ dom(dp)
    end
end;


procedure solve_clause(in $\beta_{in}$, p, c, suspended; out $\beta_{out}$; inout sat, dp)
begin
    $\beta_{ext}$:= EXTC(c, $\beta_{in}$);
    for i:=1 to m with $\ell_1$, ..., $\ell_m$ body-of c do begin
        $\beta_{aux}$:= RESTRG($\ell_i$, $\beta_{ext}$);
        switch $\ell_i$ of
            case $x_j = x_k$ :
                $\beta$:= AI_VAR($\beta_{aux}$);
            case $x_j = f(\ldots)$ :
                $\beta$:= AI_FUNC($\beta_{aux}$, $f$);
            case q(...) :
                solve_call($\beta_{aux}$, q, suspended, sat, dp);
                $\beta$:= sat($\beta_{aux}$, q);
                if ($\beta_{in}$, p) $\in$ dom(dp) then ADD_DP($\beta_{in}$, p, $\beta_{aux}$, q, dp)
        end;
        $\beta_{ext}$:= EXTG($\ell_i$, $\beta_{ext}$, $\beta$)
    end;
    $\beta_{out}$:= RESTRCS(c, $\beta_{ext}$)
end;
```

Termination

The use of widening is useful to limit the number of abstract inputs to be considered. In the case of an infinite domain, the algorithm may not terminate. To guarantee the termination, the algorithm is enhanced with a widening technique. The intuition behind this is that an element cannot be refined infinitely often.

Each time a call (β_{in}, p) is encountered, the last element of the form (β'_{in}, p) inserted in the suspended set (which should be considered as a stack[4]) is searched. If such an element exists, the computation continues with ($\beta'_{in} \cup \beta_{in}$, p) instead of ($\beta_{in}$, p); otherwise the computation proceeds normally.

This processing takes place at the beginning of the procedure solve_call and guarantees that the elements in the suspended stack with the same predicate are always increasing. The WIDEN operation can be defined as follows:

```
function WIDEN(in $\beta_{new}$, p, suspended): AS
begin
    $\beta_{old}$:= GET_PREVIOUS(p, suspended);
    WIDEN:= $\beta_{old} \nabla \beta_{new}$
end;
```

The function GET_PREVIOUS returns the substitution β of the last pair (β, p) inserted in the stack suspended or $\perp$ if there is no such pair. ∇ is the widening operator on substitutions which does, in this case, nothing more than computing the upper bound of the two abstract substitutions.

After having presented in detail the algorithm, let us summarize it with an intuitive and schematic view highlighting how all the abstract operations interact. Assume we are computing the fixpoint of the predicate p(...).

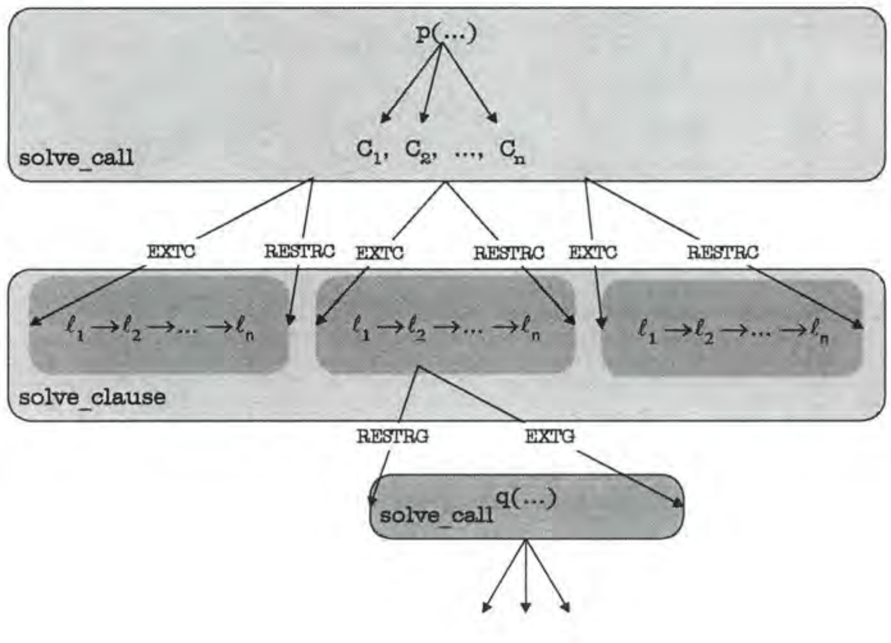


Figure 4.6 Schematic view of how the abstract operation interacts.

[4] Since the order of the elements is important for widening.

Example

Let us illustrate the algorithm with the trace of an execution of its implementation upon the append program, depicted at Figure 1.1, with the query:

$\leftarrow$ append(Var,Var,Ground).

The first iteration is shown at lines 1 to 30 . In the abstract substitutions, the first three positions represent the arguments of the procedure while the remaining ones represent the variables in the body.

The first clause first extends the substitution (it has no effect since all the variables appear in the head) – line 4. It then binds the first argument to [] – lines 5, 6 – and the second to the third – lines 7, 8. At this point – line 8 –, note that both the second and the third argument refer to the same subterm. The result of the first clause is thus the projection on the head variables – line 9 . Finally the UNION of the result with the current β_{out} (equal to $\perp$) is effected – line 10. It is worth mentioning that the traces shown in this paper do not show the RESTRG and EXTG operations for built-ins. The second clause first extends the input substitution to include the new variables – lines 13, 14. It then performs a number of unifications and calls itself recursively with the same input substitution – line 24. Since the same computation is pending, that is already initiated but not yet terminated, its current value, i.e. $\perp$, is simply picked up in the abstract domain sat. The whole clause then returns $\perp$ and the first iteration produces the same result as the one of the first clause.

```
1   SOLVE_CALL ITERATION#1: append/3
2      EXTEND ((Var(1),Var(2),Ground(3)),bottom)
3      SOLVE_CLAUSE#1
4         EXIT EXTC:  (Var(1),Var(2),Ground(3))  ps: {1,1}{2,2}
5         CALL AI_FUNC:  (Var(1),Var(2),Ground(3))  ps: {1,1}{2,2}
6         EXIT AI_FUNC:  (Ground(1):[],Var(2),Ground(3))  ps: {2,2}
7         CALL AI_VAR:  (Ground(1):[],Var(2),Ground(3))  ps: {2,2}
8         EXIT AI_VAR:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
9         EXIT RESTRC:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
10        EXIT UNION:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
11     EXIT CLAUSE#1
12     SOLVE_CLAUSE#2
13        EXIT EXTC:  (Var(1),Var(2),Ground(3),Var(4),Var(5),Var(6))
14             ps: {1,1}{2,2}{4,4}{5,5}{6,6}
15        CALL AI_FUNC:  (Var(1),Var(2),Ground(3),Var(4),Var(5),Var(6))
16             ps: {1,1}{2,2}{4,4}{5,5}{6,6}
17        EXIT AI_FUNC:  (Ngv(1):.(Var(2),Var(3)),Var(4),Ground(5),Var(2),
18             Var(3),Var(6))  ps: {2,2}{3,3}{4,4}{6,6}
19        CALL AI_FUNC:  (Ngv(1):.(Var(2),Var(3)),Var(4),Ground(5),Var(2),
20             Var(3),Var(6))  ps: {2,2}{3,3}{4,4}{6,6}
21        EXIT AI_FUNC:  (Ngv(1):.(Ground(2),Var(3)),Var(4),
22             Ground(5):.(Ground(2),Ground(6)),Ground(2),Var(3),
23             Ground(6))  ps: {3,3}{4,4}
24        CALL GOAL:  append(Var(1),Var(2),Ground(3))  ps: {1,1}{2,2}
25        EXIT GOAL:  append bottom
26        EXIT EXTG:  bottom
27        EXIT RESTRC:  bottom
28        EXIT UNION:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
29     EXIT CLAUSE#2
30     ADJUST: (Ground(1):[],Ground(2),Ground(2))  ps: {}
```

A second iteration is necessary since (β_{in}, append) depends on itself and sat(β_{in}, append) has been updated by the first clause. The second iteration is shown below, note that the first clause produces the same result. The interesting point is that the look-up in the sat now produces a result – line 24 – and that this result is propagated through all variables in the clause by the EXTG operation – line 25, 27. The union operation accumulates the results produced so far – line 30. Consequently, the analysis yields Ground for all arguments. A third iteration is in fact necessary to ensure termination but it will not add any new information.

```
1   SOLVE_CALL ITERATION#2: append/3
2     SOLVE_CLAUSE#1
3       EXIT EXTC:  (Var(1),Var(2),Ground(3))  ps: {1,1}{2,2}
4       CALL AI_FUNC:  (Var(1),Var(2),Ground(3))  ps: {1,1}{2,2}
5       EXIT AI_FUNC:  (Ground(1):[],Var(2),Ground(3))  ps: {2,2}
6       CALL AI_VAR:  (Ground(1):[],Var(2),Ground(3))  ps: {2,2}
7       EXIT AI_VAR:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
8       EXIT RESTRC:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
9       EXIT UNION:  (Ground(1):[],Ground(2),Ground(2))  ps: {}
10    EXIT CLAUSE#1
11    SOLVE_CLAUSE#2
12      EXIT EXTC:  (Var(1),Var(2),Ground(3),Var(4),Var(5),Var(6))
13          ps: {1,1}{2,2}{4,4}{5,5}{6,6}
14      CALL AI_FUNC:  (Var(1),Var(2),Ground(3),Var(4),Var(5),Var(6))
15          ps: {1,1}{2,2}{4,4}{5,5}{6,6}
16      EXIT AI_FUNC:  (Ngv(1):.(Var(2),Var(3)),Var(4),Ground(5),Var(2),
17          Var(3),Var(6))  ps: {2,2}{3,3}{4,4}{6,6}
18      CALL AI_FUNC:  (Ngv(1):.(Var(2),Var(3)),Var(4),Ground(5),Var(2),
19          Var(3),Var(6))  ps: {2,2}{3,3}{4,4}{6,6}
20      EXIT AI_FUNC:  (Ngv(1):.(Ground(2),Var(3)),Var(4),
21          Ground(5):.(Ground(2),Ground(6)),Ground(2),Var(3),
22          Ground(6))  ps: {3,3}{4,4}
23      CALL GOAL:  append(Var(1),Var(2),Ground(3))  ps: {1,1}{2,2}
24      EXIT GOAL:  append(Ground(1):[],Ground(2),Ground(2))  ps: {}
25      EXIT EXTG:  (Ground(1):.(Ground(2),Ground(3):[]),Ground(4),
26          Ground(5):.(Ground(2),Ground(4)),Ground(2),
27          Ground(3):[],Ground(4))  ps: {}
28      EXIT RESTRC:  (Ground(1):.(Ground(2),Ground(3):[]),Ground(4),
29          Ground(5):.(Ground(2),Ground(4)))  ps: {}
30      EXIT UNION:  (Ground(1),Ground(2),Ground(3))  ps: {}
31    EXIT CLAUSE#2
32    ADJUST : (Ground(1),Ground(2),Ground(3))  ps: {}
```

References

[BJCD87] M. Bruynooghe, G. Janssens, A. Callebaut and B. Demoen; *Abstract Interpretation: Towards the global optimization of Prolog programs*; In Proceedings of the 1987 Symposium on Logic Programming, pp. 192-204, San Francisco, California, August 1987; Computer Society Press of the IEEE.

[ELCRVH92] V. Englebert, B. Le Charlier, D. Roland and P. Van Hentenryck; *Generic abstract interpretation algorithms for Prolog: Two optimizations techniques and their experimental evaluation*; In M. Bruynooghe and M. Wirsing, editors, Proceeding of the Fourth International Workshop on Programming Language Implementation and Logic Programming (PLILP'92), Lecture Notes in Computer Science, Leuven, August 1992, Springer-Verlag.

[LCMVH90] B. Le Charlier, K. Musumbu and P. Van Hentenryck; *Efficient and accurate algorithms for the abstract interpretation of Prolog programs*; Technical Report 37/90, Computer Science Institute of Namur, Belgium, 1990.

[LCMVH91] 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.

[LCVH94] B. Le Charlier and P. Van Hentenryck; *Experimental evaluation of a generic abstract interpretation for Prolog*; TOPLAS, January 94.

Chapter

5

Abstract Interpretation of Prolog with Cut

This chapter is mainly based on [LCR94], and also on [BLCMVH94], [BM94] and [LCRVH94]. It presents a novel framework for the abstract interpretation of Prolog in which a particular attention has been devoted to Prolog control including the treatment of the cut primitive. The new framework is more accurate than the previous one and relies on the concept of *abstract sequence of substitutions*. It collects information about Prolog programs executing accordingly to the leftmost selection rule and the depth-first search strategy. That is why it is now adequate for determinacy analysis, detection of unending procedures and cardinality analysis.

It is thus a generic abstract interpretation algorithm for *almost full* Prolog. It just cannot handle the system predicates `assert` and `retract` and of course it cannot cope with the dynamic predicates, that is the asserted ones.

The operational semantics for Prolog with cut is derived from the one for pure Prolog by augmenting it with the cut. And so are the generic abstract semantics and algorithm.

Contents of this chapter

Concrete Semantics

This section presents a new operational semantics for Prolog with cut which can be proved equivalent to the well-accepted operational semantics exposed in the first chapter. Basically, the semantics maps procedure calls on sequences of answer substitutions, assuming an *empty context* for the calls. In a non empty context, subsequent literals can introduce loops and cuts can shorten the sequence. Therefore, in a empty context the sequence will be as complete as possible. In actual Prolog computations, some sequences will not be computed completely, but for the sake of abstract interpretation it is acceptable to assume a complete computation because infinite sequences are abstracted by finite objects. On the other hand, considering the cut at the abstract level remains desirable for optimizations purposes. This framework comes with the necessary materials to achieve this. The purpose of defining this concrete semantics is not to provide a better semantics for understanding Prolog but rather to define stuff easier to abstract.

The definition of a normalized program is enhanced to take into account the cut primitive beyond the already existing three literal forms. Recall that a normalized program is a sequence of clauses of the form $p(X_1, \ldots, X_n) \leftarrow g$ where $n \geq 0$, $p \in \mathbb{P}_n$ and g is a sequence $\ell_1, \ldots, \ell_m$ $(m \geq 0)$. Each ℓ_i is either an atom of the form $p(X_{i_1}, \ldots, X_{i_n})$, where $X_{i_1}, \ldots, X_{i_n}$ are all distinct variables and $p \in \mathbb{P}_n$, or a built-in. A built-in is either $X_i = X_j$ $(i \neq j)$, or $X_i = f(X_{j_1}, \ldots, X_{j_n})$ where $X_{j_1}, \ldots, X_{j_n}$ are all distinct variables and $f \in \mathbb{F}_n$, or the control primitive cut (!).

Substitutions sequence

A sequence of substitutions is derived from the previous notion of substitution; such a sequence is either:

- a *finite* sequence of the form $\langle\theta_1, \ldots, \theta_n\rangle$ $(n \geq 0)$,
- an *incomplete* sequence of the form $\langle\theta_1, \ldots, \theta_n, \perp\rangle$ $(n \geq 0)$,
- an *infinite* sequence of the form $\langle\theta_1, \ldots, \theta_i, \ldots\rangle$ $(i \in \mathbb{N})$

where the θ_i are substitutions, and $\perp$ models non termination. We also denote the empty sequence as $\langle\rangle$.

The usual concatenation of two sequences S_1 and S_2 is denoted $S_1{::}S_2$. subst(S) denotes the set of all substitutions occurring in the sequence S. dom(S) denotes the domain of the sequence S, that is the union of the domains of θ for every $\theta \in$ subst(S); the codomain of S, codom(S), is similarly defined. In the following, $\mathbb{PSS}$ will denote the set of all *program substitutions sequences* whose domain and codomain are subsets of $\mathbb{PV}$ and $\mathbb{RV}$ respectively. $\mathbb{RSS}$ will denote the set of all *standard substitutions sequences* whose domain and codomain are subsets of $\mathbb{RV}$.

Let D be a finite subset of $\mathbb{PV}$. A *program substitutions sequence* S on D is a substitutions sequence such that for every $\theta \in$ subst(S), $\theta \in \mathbb{PS}$ and dom(q) = D. $\mathbb{PSS}_D$ denotes the set of all substitutions sequences on D. Let $S_1, S_2 \in \mathbb{PSS}_D$, $\mathbb{PSS}_D$ can be endowed with a structure of complete partial order by defining:

- $\perp_{PSS_D} = \langle\perp\rangle$;
- $S_1 \sqsubseteq S_2$ iff either $S_1 = S_2$ or there exists $S, S' \in \mathbb{PSS}_D$ such that $S_1 = S{::}\langle\perp\rangle$ and $S_2 = S{::}S'$.

Let S_i ($i \in \mathbb{N}$) be an increasing chain in $\mathbb{P}SS_D$, i.e. $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_i \subseteq \ldots$ The least upper bound sequence $\bigcup_{i=1}^{\infty} S_i \in \mathbb{P}SS_D$ can be constructed as follows:

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

In the following, nsub(S) denotes the number of substitutions in S, whereas leng(S) denotes the number of elements in S counting also the $\perp$ element when it occurs. If S is infinite then nsub(S) = leng(S) = ∞.

Concrete operations

Concatenation. This operation extends the usual concatenation operation in order to properly handle incomplete and infinite sequences. For instance if S is incomplete or infinite, S□S' should be equal to S because, intuitively, the computation of S never terminates; hence the computation of S' will never start. Concatenation of an infinite number of sequences is also adequately defined. For example, $\square_{i=1}^{\infty}\langle\rangle = \langle\perp\rangle$ because the computation of an infinite number of sequences (although empty) never terminates. Let $S_1, S_2 \in \mathbb{P}SS_D$.

$$S_1 \square S_2 = [\textbf{if } S_1 \text{ is finite } \textbf{then } S_1{::}S_2 \textbf{ else } S_1]$$

Observe that □ is associative and that $\langle\perp\rangle \square S = \langle\perp\rangle$. We also define:

$\square_{k=1}^{0} S_k = \langle\rangle$,
$\square_{k=1}^{i+1} S_k = (\square_{k=1}^{i} S_k) \square S_{i+1}$ $(i \geq 0)$,
$\square_{k=1}^{\infty} S_k = \bigcup_{i=0}^{\infty}((\square_{k=1}^{i} S_k) \square \langle\perp\rangle)$.

First Element Sequence. This operation is useful to model the behavior of the cut primitive. Let S, S' $\in \mathbb{P}SS_D$.

$$\text{first}(S) = \begin{cases} \langle\rangle & \text{if } S = \langle\rangle \\ \langle\perp\rangle & \text{if } S = \langle\perp\rangle \\ \langle\theta\rangle & \text{if } S = \langle\theta\rangle{::}S' \end{cases}$$

Restriction to a Set of Variables. This operation is used to model calls to literals: S stands for a sequence of call substitutions. All substitutions will be restricted to the variables in the call. Let S, S' $\in \mathbb{P}SS_D$ and $D' \subseteq D$.

$$S_{|D'} = \begin{cases} \langle\rangle & \text{if } S = \langle\rangle \\ \langle\perp\rangle & \text{if } S = \langle\perp\rangle \\ \langle\theta_{|D'}\rangle{::}S'_{|D'} & \text{if } S = \langle\theta\rangle{::}S' \end{cases}$$

Projection. This operation allows to extract each element from a substitutions sequence in order to apply it to a literal and execute it. Let S $\in \mathbb{P}SS_D$ and $k \in \mathbb{N}\backslash\{0\}$.

$$S\downarrow k = \begin{cases} \theta_k & \text{if } S = \langle\theta_1, \ldots \theta_k, \ldots\rangle \\ \perp & \text{if } S = \langle\theta_1, \ldots \theta_{k-1}, \perp\rangle \end{cases}$$

Composition with a Substitutions Sequence. This operation will be used to redefine operation EXTG, renamed EXTGS, which extends the sequence of output substitutions produced by a literal to all the variables in the clause. Let $\theta \in \mathbb{P}S$ and S, S' $\in \mathbb{R}SS$.

$$\begin{array}{ll} \theta S = \langle\rangle & \text{if } S = \langle\rangle \\ \quad\ \langle\bot\rangle & \text{if } S = \langle\bot\rangle \\ \quad\ \langle\theta\sigma\rangle{::}\theta S' & \text{if } S = \langle\sigma\rangle{::}S' \\ \bot S = \langle\bot\rangle & \end{array}$$

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

$$\mathrm{EXTC}(c, \theta) = \{\theta': \mathrm{dom}(\theta') = D,\ \theta'_{|D'} = \theta \wedge \forall X \in D\backslash D',\ X \text{ is free in } \theta'\}$$

$$\mathrm{RESTRC}(c, S) = \{S_{|D'}\}$$

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 EXTGS operation extends a substitution θ with a substitutions sequence S representing the result of executing a call ℓ on θ.

$$\mathrm{RESTRG}(l, \theta) = \{\theta': \mathrm{dom}(\theta') = \{X_1, \ldots, X_n\} \wedge X_j\theta' = X_{i_j}\theta\ (1 \le j \le n)\}$$

$$\begin{array}{l} \mathrm{EXTGS}(l, \theta, S) = \{\theta S': S' \in \mathrm{IRSS} \wedge \exists\theta' \in \mathrm{RESTRG}(l, \theta) \mid \theta' S' = S \\ \qquad\qquad \text{where } \mathrm{dom}(S') \subseteq \mathrm{codom}(\theta'), \\ \qquad\qquad\qquad (\mathrm{codom}(\theta)\backslash\mathrm{codom}(\theta')) \cap \mathrm{codom}(S') = \emptyset\} \end{array}$$

Unification operations. Operations AI_VAR and AI_FUNC have the same functionality than before, that is AI_VAR unifies $X_1\theta$ with $X_2\theta$ and AI_FUNC unifies $X_1\theta$ with $f(X_2, \ldots, X_n)\theta$. Recall that these operations do not have the X_i as arguments because the RESTRG operation has been performed before. However, besides returning an abstract substitution these operations now return an abstract substitutions sequence.

$$\begin{array}{l} \mathrm{AI\_VAR}(\theta) = [\ \textbf{if } \mathrm{mgu}(X_1\theta, X_2\theta) = \emptyset \\ \qquad\qquad\quad \textbf{then } \{\ \langle\rangle\ \} \\ \qquad\qquad\quad \textbf{else } \{\ \langle\theta\sigma\rangle: \sigma \in \mathrm{mgu}(X_1\theta, X_2\theta)\}\ ] \end{array}$$

$$\begin{array}{l} \mathrm{AI\_FUNC}(\theta, f) = [\ \textbf{if } \mathrm{mgu}(X_1\theta, f(X_2, \ldots, X_n)\theta) = \emptyset \\ \qquad\qquad\qquad \textbf{then } \{\ \langle\rangle\ \} \\ \qquad\qquad\qquad \textbf{else } \{\ \langle\theta\sigma\rangle: \sigma \in \mathrm{mgu}(X_1\theta, f(X_2, \ldots, X_n)\theta)\}\ ] \end{array}$$

Abstract Semantics

In the following, let IPS denote the set of abstract substitutions, IPSS denote the set of abstract substitutions sequences and IPSSC denote the set of abstract substitutions sequences with cut information. We assume the existence of three complete partial orders: AS, ASS and ASSC. Elements of AS are called abstract substitutions and denoted by β. Elements of ASS are called abstract substitutions sequences and denoted by S. Elements of ASSC are called abstract substitutions sequences with cut information and denoted by C. The meaning of these objects will be given later through monotonic concretization functions, Cc:AS → CS, Cc:ASS → CSS and Cc:ASSC → CSSC. CS = $\mathscr{P}$(IPS) and CSS = $\mathscr{P}$(IPSS) where all elements of CSS are upper-closed. CSSC is similarly defined but increasing chains only contain substitutions sequences with identical cut information. CS, CSS and CSSC are complete lattices with respect to set inclusion ⊆.

In the following, all sets that are considered are supposed to be complete. Let D be a subset of $\mathbb{P}\mathbb{V}$ and Θ be a subset of $\mathbb{P}S_D$ – the set of abstract substitutions restricted to D. We denote CS_D the set of Θs. CS_D is a complete lattice with respect to set inclusion $\subseteq$.

Definition. Let Σ be a subset of $\mathbb{P}SS$. We say that Σ is *upper-closed* iff, for all increasing chains, $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_i \subseteq \ldots$

$$S_i \in \Sigma \ (\forall\, i \in \mathbb{N}) \Rightarrow \bigcup_{i=0}^{\infty} S_i \in \Sigma.$$

Abstract tuples

The abstract semantics of a program P is defined as a set of abstract tuples (β_{in}, p, B_{out}) where p is a predicate symbol of arity n occurring in P, D is the set of program variables $\{X_1, \ldots, X_n\}$, $\beta_{in} \in AS_{|D}$, $B_{out} \in ASS_{|D}$. $AS_{|D}$ is the set of abstract substitutions defined on D and $ASS_{|D}$ is the set of abstract substitutions sequences defined on D. The *underlying domain* UD of the program P is the set of all (β_{in}, p) such that $\beta_{in} \in AS_{|D}$ and p occurs in P. We only consider sets of abstract tuples which are functional, total and monotonic, i.e. the sets satisfying:

1. $\forall(\beta_{in}, p) \in UD, \exists!\, B_{out} \in ASS_{|D}: (\beta_{in}, p, B_{out}) \in sat,$

2. $(\beta^1_{in}, p), (\beta^2_{in}, p) \in UD: \beta^1_{in} \leq \beta^2_{in} \Rightarrow sat(\beta^1_{in}, p) \leq sat(\beta^2_{in}, p).$

We note $B_{out} = sat(\beta_{in}, p)$ iff $(\beta_{in}, p, B_{out}) \in sat$. We denote SATT the set of all those sets; it is endowed with the following ordering:

$$sat_1 \leq sat_2 \text{ iff } \forall(\beta_{in}, p) \in UD: sat_1(\beta_{in}, p) \leq sat_2(\beta_{in}, p).$$

Abstract operations

Many of these operations are identical or simple generalizations of operations described previously. Others are simple 'conversion' operations between the three different objects manipulated, i.e. the abstract substitutions, the abstract substitutions sequences and the abstract substitutions sequences with cut information. The brand new operations are CONC, AI_CUT, EXTGS. Some of these operations are first or only presented as rewritten forms of the actual abstract operations in order to make their comprehension more intuitive; indeed, they should be considered as *concrete* versions of the actual abstract operations.

Substitutions Sequence from a Substitutions Sequence with Cut Information: SEQ(C) = B'. This operation forgets the cut information of a substitutions sequence. It is applied to the result of the last clause of a procedure before combining this with the result of the other clauses.

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

Sets of Substitutions from a Substitutions Sequence. This operation unmakes the sequential structure of a substitutions sequence. It is applied before the execution of a literal in a clause. Let $\langle S, cf\rangle \in \mathbb{P}CSS_D$.

$$SUBST(\langle S, cf\rangle) = \bigcup_{k=1}^{nsub(S)} \{\ proj_k(S)\ \}$$

Formally the operation denoted SUBST(C) = β, and its consistency condition is the following one:

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

Concatenation of two Substitutions Sequences. This operation combines the result of a clause in a procedure with the results of the remaining clauses. If a cut was executed in the first clause, other clauses are ignored. Otherwise the results are combined. Let $\langle S, cf\rangle \in$ IPCSS$_D$ and $S' \in$ IPSS$_D$.

$$CONC(\langle S, cf\rangle, S') = [\textbf{if } cf = \text{cut } \textbf{then } \{S\} \textbf{ else } \{S\square S'\}]$$

We could also extend our operation CONC with a third argument which is the value of the input abstract substitution for the procedure. This new operation should be able to detect incompatibility between abstract sequences when they result from different abstract input substitutions. Therefore, the analysis would also detect determinacy. Let us give the consistency condition of the new version of $CONC(\beta, C, B) = B'$:

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

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. Concatenation 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.)

An usual way to implement efficiently multi-directional procedures consists in designing several specialized versions which are called by the general "multi-directional" one. The clauses of the later contain test predicates followed by a cut and a call to the appropriate specialized version. Such procedures can be analyzed more accurately in this latest framework. The accurate handling of the cut will limit the analysis to a single specialized version. Test predicates are exposed in a comprehensive way in [BM94].

Extension of a Substitution to a Sequence. The abstract version of this operation is used at the beginning of a clause. It expresses the fact that the empty prefix (of the clause body) produces a one element sequence and that no cut has been executed so far. Let $\theta \in$ IPS$_D$.

$$EXT\_NOCUT(\theta) = \{\, \langle\, \langle\theta\rangle, \text{nocut}\,\rangle\,\}$$

Formally the operation is denoted $EXT\_NOCUT(\beta) = C$, and its consistency condition is the following one:

$$\theta \in Cc(\beta) \Rightarrow \langle\, \langle\theta\rangle, \text{nocut}\,\rangle \in Cc(C).$$

Extension at a 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 $\{X_1,\ldots,X_m\}$ $(m \geq n)$.

$$\left.\begin{array}{l}\theta \in Cc(\beta),\\ Y_1,\ldots,Y_{m-n} \text{ are distinct}\\ \qquad\text{standard variables,}\\ Y_1,\ldots,Y_{m-n} \notin codom(\theta)\end{array}\right\} \Rightarrow \{X_1/X_1\theta,\ldots, X_n/X_n\theta, X_{n+1}/Y_1,\ldots,X_m/Y_{m-n}\} \in Cc(\beta').$$

Restriction at a 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 $D = \{X_1,\ldots,X_n\}$:

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

Restriction before a Call: RESTRG(ℓ, β) = β'. 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 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 a Sequence with a Set of Sequences: EXTGS(ℓ, C, B) = C'. The EXTGS operation extends a pair ⟨S, *cf*⟩ with a set of sequences Σ representing the results of executing a call ℓ with all substitutions in S. (At the abstract level, Σ is abstracted by a single abstract sequence.)

$$\begin{array}{l} \mathrm{EXTGS}(\ell, \langle S, cf \rangle, \Sigma) = \\ \quad \{\, \langle S', cf \rangle : \exists\, S_1,\ldots,S_{leng(S)} \in \Sigma,\ S'_1,\ldots,S'_{leng(S)} \in \mathrm{IRSS}: \\ \qquad S' = \square_{k=1}^{leng(S)}\ proj_k(S)S'_k \wedge S_k = \theta_k S'_k \wedge \\ \qquad \theta_k \in \mathrm{RESTRG}(\ell, proj_k(S))\ (1 \le k \le leng(S)) \wedge \\ \qquad dom(S'_k) \subseteq codom(\theta_k) \wedge \\ \qquad (codom(proj_k(S)) \setminus codom(\theta_k)) \cap codom(S'_k) = \emptyset \,\} \end{array}$$

Abstract Execution of the Cut. The abstract operation AI_CUT is executed when a cut is called with input ⟨S, *cf*⟩ representing the result of the prefix leading to the cut. Let ⟨S, *cf*⟩ ∈ IPCSS_D. Formally, the operation can be specified as AI_CUT(C) = C' where the components of C' are defined as follows:

$$\begin{array}{ll} \beta' & = \beta \\ m' & = \min(1, m) \\ M' & = \min(1, M) \\ t' & = [\textbf{if } M{=}0 \textbf{ and } t{=}snt \textbf{ then } snt \\ & \qquad\qquad \textbf{else if } t{=}st \textbf{ or } m{\ge}1 \textbf{ then } st \\ & \qquad\qquad \textbf{else } pt] \\ cf' & = [\textbf{if } cf{=}cut \textbf{ or } m{\ge}1 \textbf{ then } cut \\ & \qquad\qquad \textbf{else if } cf{=}nocut \textbf{ and } M{=}0 \textbf{ then } nocut \\ & \qquad\qquad \textbf{else } weakcut] \end{array}$$

Let gs be the sequence of literals before a cut (!) in a clause. Execution of gs for a given input substitution θ either fails or loops without producing any result, or produces one or more results before failing, looping or producing results forever. Execution of the goals gs,! 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 gs) and then stops. At the abstract level, C represents a set of substitutions produced by gs, while C' represents the corresponding set of substitution sequences produced by gs,!. Clearly the sequence 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:

$$\langle\, \langle\rangle, cf \rangle \in Cc(C) \Rightarrow \langle\, \langle\rangle, cf \rangle \in Cc(C');$$

$$\langle\, \langle\bot\rangle, cf \rangle \in Cc(C) \Rightarrow \langle\, \langle\bot\rangle, cf \rangle \in Cc(C');$$

$$\langle\, \langle\theta\rangle{::}S, cf \rangle \in Cc(C) \Rightarrow \langle\, \langle\theta\rangle, cut \rangle \in Cc(C').$$

Abstract Unification of two Program Variables: AI_VAR(β) = B. This operation is similar to the operation presented in the previous chapter but returns an abstract substitutions sequence instead of an abstract substitution. As the concrete unification may only fail or succeed, Cc(B) should only contain *finite* sequences of length 0 or 1:

$$\left.\begin{array}{l}\theta \in \mathrm{Cc}(\beta),\\ \sigma \in \mathrm{mgu}(X_1\theta, X_2\theta)\end{array}\right\} \Rightarrow \langle\theta\sigma\rangle \in \mathrm{Cc}(B),$$

$$\left.\begin{array}{l}\theta \in \mathrm{Cc}(\beta),\\ \varnothing = \mathrm{mgu}(X_1\theta, X_2\theta)\end{array}\right\} \Rightarrow \langle\,\rangle \in \mathrm{Cc}(B).$$

Abstract Unification of a Variable and a Functor: AI_FUNC(β, *f*) = B. This operation is similar to the one above.

$$\left.\begin{array}{l}\theta \in \mathrm{Cc}(\beta),\\ \sigma \in \mathrm{mgu}(X_1\theta, f(X_2,\ldots,X_n)\theta)\end{array}\right\} \Rightarrow \langle\theta\sigma\rangle \in \mathrm{Cc}(B),$$

$$\left.\begin{array}{l}\theta \in \mathrm{Cc}(\beta),\\ \varnothing = \mathrm{mgu}(X_1\theta, f(X_2,\ldots,X_n)\theta)\end{array}\right\} \Rightarrow \langle\,\rangle \in \mathrm{Cc}(B).$$

Abstract semantics

The main difference with the abstract semantics proposed in the previous chapter is that answer abstract substitutions are replaced by abstract substitutions sequences. Abstract operations are modified accordingly. For example, formerly we used an operation UNION to collect the results of the clauses. This operation has been replaced by a more elaborate one: CONC. Moreover a new operation is introduced: AI_CUT.

In the following we assume an underlying program P, B_{out} to be an abstract sequence and C_{out} to be an abstract sequence with cut information. The abstract semantics is defined by means of four functions and one transformation depicted at Figure 5.1.

The first function Tp has the purpose to solve a predicate. It receives the input substitution β_{in}, the predicate symbol p, the set of abstract tuples sat and it returns the substitutions sequence being the result of the predicate.

The next function Tpr, is called by Tp to process all the clauses defining the procedure p on β_{in}. This function recursively concatenates the result of the first clause with the results of the remaining parts of the procedure. This is effected by the CONC operation which takes into account the possible non termination of the first clause or the fact that it could have been '*cut*'. In these cases, the others remaining clauses are ignored. In the definition of Tpr, we assume c to a be one clause and c.pr to be a sequence of clauses where c is the head of the sequence and pr its tail, that is the 'remaining procedure'.

Tc is the third function. It processes clauses and returns an abstract substitutions sequence with cut information. It first extends, EXTC, the abstract substitution to take into account the free variables of the body of the clause. Then the body of the clause is solved and ultimately the solution is restricted, RESTRC, to the variables occurring in the head of the clause.

The fourth function executes the body of a clause by considering each literal in turn, from the first to the last, that is from left to right accordingly to the SLD-Resolution. The empty prefix of the body produces a one element abstract sequence furthermore meaning that no cut has been executed. When the next literal to execute is a cut, operation AI_CUT is executed. Otherwise the next literal ℓ is executed with input β_2 approximating all substitutions in the concretization of C_{out}. Operation RESTRG expresses β_2 in terms of the formal parameters of ℓ. If ℓ is a procedure call, then only a lookup in sat is performed, otherwise either operation AI_VAR or AI_FUNC is executed. Operation EXTGS is performed after each call in order to obtain the result of the full goal.

Figure 5.1
The abstract semantics.

```
TSAT(sat) = {(β_in, p, B_out): (β_in, p) ∈ UD and B_out = Tp(β_in, p, sat)}

Tp(β_in, p, sat) = Tpr(β_in, pr, sat)
      where       pr is the procedure defining p

Tpr(β_in, c, sat) = SEQ(C_out)
      where       C_out = Tc(β_in, c, sat)
Tpr(β_in, c.pr, sat) = CONC(C_out, B_out)
      where       B_out = Tpr(β_in, pr, sat)
                  C_out = Tc(β_in, c, sat)

Tc(β_in, c, sat) = RESTRC(c, C_out)
      where       C_out = Tg(EXTC(c, β_in), g, sat)
                  g is the body of c

Tg(β_in, ⟨⟩, sat) = C_out
      where       C_out = EXT_NOCUT(β_in)
Tg(β_in, gs::!, sat) = AI_CUT(C_out)
      where       C_out = Tg(β_in, gs, sat)
Tg(β_in, gs::ℓ, sat) = EXTGS(ℓ, C_out, B)
      where       B =  sat(β_1, p)         if ℓ is p(...)
                       AI_VAR(β_1)         if ℓ is X_i = X_j
                       AI_FUNC(β_1, f)     if ℓ is X_i = f(...)
                  β_1 = RESTRG(ℓ, β_2)
                  β_2 = SUBST(C_out)
                  C_out = Tg(β_in, gs, sat)
```

Abstract domains

It is possible to define abstract domains for substitutions sequences simply by enhancing the domains designed for abstract substitutions with information typical of sequences such as minimal and maximal length, information about cut execution and information about termination or non termination. Note that the framework does not allow to infer sure termination of recursive procedures.

Abstract sequences

Let us assume that an abstract domain of abstract substitutions, $AS_{|D}$, is given for any set of program variables $D = \{X_1, \ldots, X_n\}$. This domain is endowed with an ordering $\leq$ and a concretization function Cc. An *abstract substitutions sequence*, on domain D, is a 4-tuple $B = \langle\beta, m, M, t\rangle$ such that $\beta \in AS_{|D}$, $m \in \mathbb{N}$, $M \in \mathbb{N} \cup \{\infty\}$ and $t \in \{st, pt, snt\}$.

Intuitively, β is the abstract substitution representing all the substitutions in the abstract sequence, m is the minimum length — or the minimum number of solutions —, M is the maximum length and t is a termination information. The sequence is finite if t = st and incomplete or infinite if t = snt — st stands for 'sure termination', pt for 'possible termination' and snt for 'sure non termination'. Formally, the meaning of B is defined by the concretization function Cc: $ASS_{|D} \rightarrow CSS_{|D}$ defined as follows:

$$\text{Cc}(B) = \text{Sseq}_1(\beta) \cap \text{Sseq}_2(m, M) \cap \text{Sseq3}(t)$$

where $\text{Sseq}_1(\beta) = \{S: \text{Subst}(S) \subseteq \text{Cc}(\beta)\}$,
$\text{Sseq}_2(m, M) = \{S: m \leq \text{nsub}(S) \leq M\}$,
$\text{Sseq3}(\text{snt}) = \{S: S \text{ is incomplete or infinite}\}$,
$\text{Sseq3}(\text{pt}) = \text{IPSS}_D$.

An *abstract sequence with cut information*, on the domain D, is a pair $C = \langle B, cf\rangle$ where $B \in \text{ASS}_{|D}$ and $cf \in \text{CF} = \{\text{cut}, \text{nocut}, \text{weakcut}\}$. Intuitively, cut means that for all possible executions (of a goal) a cut has been executed, nocut means that no cut has been executed, weakcut means that it was executed for all non empty sequences and undef means that all cases are possible. Formally the meaning of C is given by the concretization function $\text{Cc}: \text{ASSC}_{|D} \rightarrow \text{CSSC}_{|D}$ defined as follows:

$\text{Cc}(\langle B, \text{cut}\rangle) = \{(S, \text{cut}): S \in \text{Cc}(B)\}$,
$\text{Cc}(\langle B, \text{nocut}\rangle) = \{(S, \text{nocut}): S \in \text{Cc}(B)\}$,
$\text{Cc}(\langle B, \text{weakcut}\rangle) = \{(S, \text{cut}): S \in \text{Cc}(B)\} \cup \{(\langle\rangle, \text{nocut}), (\langle\bot\rangle, \text{nocut})\}$.

Ordering on abstract substitutions sequences

Let $B_1 = \langle\beta_1, m_1, M_1, t_1\rangle$ and $B_2 = \langle\beta_2, m_2, M_2, t_2\rangle$ be two abstract sequences.

$$B_1 \leq B_2 \Leftrightarrow \begin{cases} \beta_1 \leq \beta_2 \\ m_1 \geq m_2 \\ M_1 \leq M_2 \\ t_1 \leq t_2 \end{cases}$$

where the ordering on the β_i, m_i, M_i are the usual ones and $t_1 \leq t_2$ is defined as

$$\left.\begin{matrix} \text{st} \\ \text{snt} \end{matrix}\right\} < \text{pt}.$$

It is easy to show that $B_1 \leq B_2 \Rightarrow \text{Cc}(B_1) \subseteq \text{Cc}(B_2)$.

Generic Abstract Interpretation Algorithm

The abstract semantics is computed by a generic abstract interpretation algorithm very similar to the algorithm proposed in the previous chapter. The main difference with the previous algorithm is the following one. The novel algorithm uses enhanced abstract operations which return *abstract sequences* instead of abstract substitutions — it does not make necessarily a big difference from a computational standpoint because the abstract domain for abstract sequences are simply derived from the abstract domain for substitutions. This algorithm keeps using a dependency graph, it also has enhanced versions of the operations managing the set of abstract tuples... This is thus basically the same algorithm with enhanced abstract operations.

Overview of the approach

Recall that the purpose of the algorithm is to compute a subset of a post-fixpoint of the transformation TSAT that includes a tuple of the form (β_{in}, p, B_{out}) but as few other elements as possible. To achieve this, the algorithm computes a series of sets $\text{sat}_0, \ldots, \text{sat}_n$

such that $\text{sat}_i < \text{sat}_{i+1}$ and such that sat_n contains (β_{in}, p, B_{out}). The first approximation sat_0 contains a tuple (β, p, B_0) such that $\{\langle\bot\rangle\} \in \text{Cc}(B_0)$. The algorithm then moves from one partial set to another by selecting:

- an element (α_{in}, q) which is not yet present in the sat but needs to be computed, or
- an element (α_{in}, q) whose value sat(α_{in}, q) can be improved because the values of some elements upon which it is depending have been updated.

The detection of termination and the choice of the element to work on are important choices to be made. The same strategy as before is adopted. The algorithm thus works the same way, that is to say, given an input pair (β_{in}, p) it executes the function Tp of the abstract semantics.

Sets of abstract tuples manipulations

Recall that at the contrary of the *abstract semantics* which does not need operations on the set of abstract tuples, the algorithm does need operations to construct a subset of the fixpoint containing the solution of the query. There are two operations that need to be defined: EXTEND and ADJUST. These two operations are almost unchanged, but however refined to cater for the requirements of the new framework.

- **EXTEND**(β_{in}, p, sat) = sat'. Given a set of abstract tuples sat, a predicate symbol p and an abstract substitution β_{in}, it returns a new set of abstract tuples sat' containing (β_{in}, p) in its domain. (β_{in}, p) was supposed not to already belong to sat. This operation simply inserts a new pair together with $\bot_B$ as its value, where $\bot_B$ denotes the smallest abstract sequence such that $\{\langle\bot\rangle\} \in \text{Cc}(\bot_B)$.

- **ADJUST**(β_{in}, p, B_{out}, sat) = sat'. This operation is responsible to update the set of abstract tuples. It relies on a special form of widening to do so (see Termination later on). Given the substitutions sequence B_{out} which represents the new result for sat(β_{in}, p), this operation returns a new set of abstract tuples sat' which is the sat updated in such a way that sat'(β_{in}, p) = sat(β_{in}, p) ∇ B_{out} and all other values stand unchanged.

Recall that a slightly more general version of ADJUST is used in the algorithm, it returns besides the new set of abstract tuples, the set of pair (β_{in}, p) whose values have been updated.

Procedure call dependencies

The purposes of a dependency graph are mainly the detection of redundant computations and efficient processing of mutually recursive programs. We still use the same concepts of *dependency graph* and *transitive closure* of the dependency graph, which were defined in the previous chapter.

Recall that the basic intuition is that the dependency graph dp(β_{in}, p) represents at some point the set of pairs which (β_{in}, p) depends directly upon. This is also why we need to define the transitive closure, trans_dp(bin, p, dp), which represents all the pairs which if updated would require reconsidering (β_{in}, p). The same three operations — EXT_DP, ADD_DP and REMOVE_DP — can be reused, without having to be altered.

Generic abstract interpretation algorithm

The top-level procedure **solve** receives an input substitution β_{in} and a predicate symbol p and returns the final dependency graph and the set of abstract tuples sat containing (β_{in}, p, B_{out}) $\in \mu$(TSAT). From these results, it is straightforward to compute the set of pairs (α, q) used by (β_{in}, p), their values in the post-fixpoint as well as the abstract substitutions at any program point.

The procedure **solve_call** receives as inputs an abstract substitution β_{in}, its associated predicate symbol p, a set suspended of pairs (α, q), a set of abstract tuples sat and a dependency graph dp. The set suspended contains all pairs (α, q) for which a subcomputation has been initiated and not been completed yet. The procedure considers or reconsiders the pair (β_{in}, p) and updates sat and dp accordingly. The core of the procedure is only executed when (β_{in}, p) is not suspended and not in the domain of the dependency graph. If (β_{in}, p) is suspended, no subcomputation should be initiated. If (β_{in}, p) is in the domain of the dependency graph, this means that none of the elements upon which it is depending have been updated. Otherwise, a new computation with (β_{in}, p) is started. This subcomputation may extend sat if it is the first time (β_{in}, p) is considered. It is important to mention that the EXTEND operation always assumes that a procedure loops when first met, i.e. it adds (β_{in}, p, $\langle\bot$, 0, 0, snt$\rangle$) in sat.

The core of the procedure is a repeat-until loop which computes the lower approximation of (β_{in}, p) given the elements of the suspended set. Local convergence is reached when (β_{in}, p) belongs to the domain of the dependency graph. One iteration of the loop amounts to execute solve_procedure with the same inputs and the procedure proc(p) consisting of all the clauses defining p. If the result produced, B_{out}, is greater or incomparable to the current value sat(β_{in}, p), then the set of abstract tuples is updated. The dependency graph is also updated accordingly by removing all elements which depend (directly or indirectly) on (β_{in}, p). Note that the calls to solve_procedure are done with an extended suspended set since a subcomputation has been started with (β_{in}, p). Note also, that before calling solve_procedure, the dependency graph has been brought up to date to include (β_{in}, p) (which is guaranteed not to be in the domain of the dependency graph before that update). (β_{in}, p) can be removed from the domain of the dependency graph during the execution of the loop if a pair it is depending upon is updated. The very first statement of solve_call, the widening operation, remains unchanged.

The procedure **solve_procedure** executes a procedure accordingly to the abstract semantics. The condition clause(pr) means that pr consists only of one clause. The statement c.sfx := pr decomposes pr into its head and tail (suffix). Finally the procedure contains an important optimization which amounts to test sure non termination or sure execution of a cut in the first clause. In that case, the remaining part of the procedure is not executed. This optimization does not change the accuracy of the result but can speed up the computation.

The procedure **solve_clause** executes a single clause for an input pair and returns an abstract sequence with cut information representing the execution of the clause on that input pair. The procedure first extends the substitution with the clause variables, then executes the body of the clause and terminates by restricting the abstract sequence to the head variables. The execution of a cut is simply performed by the abstract operation AI_CUT. The execution of another literal requires three steps.

- The computation of an abstract substitution representing all the concrete substitutions in C_{out} furthermore restricted to the variables occurring in the literal: this is done by the operation RESTRG, giving β_{aux}.

- The execution of the literal on β_{aux}, producing B_{out}. If the literal is concerned with unification, the operation AI_VAR or AI_FUNC is performed, depending on the form of the literal. If it is a goal then procedure solve_call is recursively called and the result is retrieved in sat. Moreover, if (β_{in}, p) is in the domain of the dependency graph it is necessary to add a new dependency; otherwise it means that (β_{in}, p) needs to be reconsidered anyway and no dependency needs to be recorded. If the literal is of none of these three forms, then solve_builtin_literal is called and does the same job as all the procedures appended to solve_clause to handle the system predicates (see [BM94]).

- The propagation of the result of the literal to the variables occurring in the body of the clause is made by the operation EXTGS.

```
procedure solve (in $\beta_{in}$, p; out sat, dp)
begin
    sat:= $\varnothing$;
    dp:= $\varnothing$;
    solve_call($\beta_{in}$, p, $\varnothing$, sat, dp)
end;


procedure solve_call(in $\beta_{in}$, p, suspended; inout sat, dp)
begin
    $\beta_{in}$:= WIDEN($\beta_{in}$, p, suspended);
    if ($\beta_{in}$, p) $\notin$ (dom(dp) $\cup$ suspended) then begin
        if ($\beta_{in}$, p) $\notin$ dom(sat) then sat:= EXTEND($\beta_{in}$, p, sat);
        repeat
            EXT_DP($\beta_{in}$, p, dp);
            solve_procedure($\beta_{in}$, p, proc(p), suspended $\cup$ {($\beta_{in}$, p)}, $B_{out}$, sat, dp);
            (sat, modified):= ADJUST($\beta_{in}$, p, $B_{out}$, sat);
            REMOVE_DP(modified, dp)
        until ($\beta_{in}$, p) $\in$ dom(dp)
    end
end;


procedure solve_procedure(in $\beta_{in}$, p, pr, suspended; out $B_{out}$; inout sat, dp)
begin
    if clause(pr) then begin
        solve_clause($\beta_{in}$, p, pr, suspended, $C_{out}$, sat, dp);
        $B_{out}$:= SEQ($C_{out}$)
    end else begin
        c.sfx:= pr;
        solve_clause($\beta_{in}$, p, c, suspended, $C_{out}$, sat, dp);
        if cut_or_nt($C_{out}$) then
            $B_{out}$:= SEQ($C_{out}$)
        else begin
            solve_procedure($\beta_{in}$, p, sfx, suspended, $B_{out}$, sat, dp);
            $B_{out}$:= CONC($\beta_{in}$, $C_{out}$, $B_{out}$)
        end
    end
end;
```

```
procedure solve_clause(in $\beta_{in}$, p, c, suspended; out $C_{out}$; inout sat, dp)
begin
    $C_{out}$:= EXT_NOCUT(c, $\beta_{in}$);
    for i:=1 to n with $\ell_1, \ldots, \ell_n$ body-of c do
        if $\ell_i$ is ! then
            $C_{out}$:= AI_CUT($C_{out}$)
        else begin
            $\beta_{aux}$:= RESTRG($\ell_i$, SUBST($C_{out}$));
            switch $\ell_i$ of
                case $x_j = x_k$ :
                    B:= AI_VAR($\beta_{aux}$);
                case $x_j = f(\ldots)$ :
                    B:= AI_FUNC($\beta_{aux}$, $f$);
                case q(...) :
                    solve_call($\beta_{aux}$, q, suspended, sat, dp);
                    B:= sat($\beta_{aux}$, q);
                    if ($\beta_{in}$, p) $\in$ dom(dp) then ADD_DP($\beta_{in}$, p, $\beta_{aux}$, q, dp);
                otherwise :
                    solve_builtin_literal($\beta_{aux}$, $\ell_i$, $C_{out}$)
            end;
            $C_{out}$:= EXTGS($\ell_i$, $C_{out}$, B)
        end;
    $C_{out}$:= RESTRC(c, $C_{out}$)
end;
```

Termination

Definition. Let sat $\in$ SATT. sat is said *pre-consistent* iff for all $(\beta, p) \in$ UD and for all $\theta \in Cc(\beta)$, $\langle\theta, p\rangle \longrightarrow S$ implies that there exists $S' \subseteq S$ and $S' \in Cc(sat(\beta, p))$.

Ordering for widening. $B_1 \prec B_2 \Leftrightarrow \beta_1 < \beta_2 \vee (\beta_1 = \beta_2 \wedge B_1 \leq B_2)$.

The major problem for the algorithm is to ensure termination without loosing accuracy of the analysis. In the previous algorithm termination was ensured by computing, for each required input pair (β_{in}, p), an increasing sequence $\beta_1 \leq \ldots \leq \beta_n$. This sequence was guaranteed to be finite because a widening technique was used. The ordering $\leq$, on abstract substitutions, reflects, at the abstract level, the inclusion relation between sets of concrete substitutions, i.e. $Cc(\beta_1) \subseteq \ldots \subseteq Cc(\beta_n)$. In this framework, the same technique is still safe, provided that the computation is started with *pre-consistent* values, but entails an unacceptable loss of precision because each iterate should include all incomplete sequences corresponding to previous iterates and, in particular, the *whole incomplete sequence* $\langle\perp\rangle$ — it is reasonable to assume that $\langle\perp\rangle$ belongs to the concretization of the initial abstract sequence. Therefore it should not be possible to compute for instance accurately the minimum length of the sequences of answer substitutions. To cope with that problem, a widening technique has been introduced. We assume that it is possible to define on the abstract domain ASS_{D} an ordering $\prec$, not related to inclusion, and a widening operation

$$\nabla : ASS_{D} \times ASS_{D} \rightarrow ASS_{D}$$

such that for every sequence $B_0, \ldots, B_i, \ldots$ and $B'_1, \ldots, B'_i, \ldots$ where $B'_{i+1} = B_{i+1} \nabla B'_i$ $(i \geq 0)$, the following holds:

- $B'_i \geq B_i$ $(i \geq 1)$;

- $B'_1 \prec \ldots \prec B'_i \prec \ldots$;
- the above sequence is stationary.

Thanks to the new ordering and operation we can achieve termination as follows. Let $(\beta, \mathtt{p})$ be an input pair for which we want to compute the corresponding abstract sequence. We compute two sequences $B_0,\ldots,B_i,\ldots$ and $B'_1,\ldots,B'_i,\ldots$ as follows:

- B_0 is such that $\langle\bot\rangle \in Cc(B_0)$ and is stored in the initial **sat** as the output for $(\beta, \mathtt{p})$;
- B_i results from the i-th abstract execution of procedure $\mathtt{p}$, with abstract input β;
- $B'_i = B_i \nabla B'_{i-1}$ is put in the current **sat** after the i-th abstract execution of procedure $\mathtt{p}$;
- execution stops when $B_{i+1} \leq B'_i$.

Termination of this process is ensured because a sequence $B'_1,\ldots,B'_i,\ldots$ is guaranteed to be stationary and because it cannot be the case if condition $B_{i+1} \leq B'_i$ never happens. A sample widening is defined hereafter.

Widening. The basic idea of the widening strategy is to iterate on the transformation TSAT as long as the abstract substitution part of the abstract sequence keeps increasing. When only the remaining parts continue to change, we set M to ∞ and build an increasing sequence (with respect to $\leq$) in order to reach a post-fixpoint in a finite number of steps — since the domain of abstract sequences is infinite.

$$\begin{array}{rll} B_1 \nabla B_2 & = \langle \mathtt{LUB}(\beta_1,\beta_2), \mathtt{m}_1, \mathtt{M}_1, \mathtt{t}_1\rangle & \text{if } \beta_1 \not\leq \beta_2 \\ & = \langle \beta_2, \mathtt{m}_1, \mathtt{M}_1, \mathtt{LUB}(\mathtt{t}_1, \mathtt{t}_2)\rangle & \text{if } \beta_1 \leq \beta_2 \text{ and } \mathtt{t}_1 \not\leq \mathtt{t}_2 \\ & = \langle \beta_2, \mathtt{min}(\mathtt{m}_1, \mathtt{m}_2), \infty, \mathtt{t}_2\rangle & \text{if } \beta_1 \leq \beta_2 \text{ and } \mathtt{t}_1 \leq \mathtt{t}_2 \text{ and } B_1 \not\leq B_2 \\ & = B_2 & \text{if } B_1 \leq B_2 \end{array}$$

Implementation

Actually, the implementation differs from the algorithm exposed here. The changes are due to loss of precision within this framework. A comprehensive discussion of the actual implementation is given in [BLCMVH94], [BM94] and [LCRVH94].

Generic Algorithm Revisited

The algorithm presented previously is slightly modified in order to introduce the next one capable of handling the system predicates `assert` and `retract`. Of course, it keeps the same features: early detection of termination and redundant computations, foiling the infinite nature of the abstract domain... And it has still the same functionality: determinacy analysis, detection of unending procedures...

We can also see the following algorithm as a simplified version of the algorithm presented in the next chapter. Indeed, the algorithm is exactly the same except that all the statements, procedures and other features handling the dynamic predicates — that is the asserted ones — have been removed. Hence this version of the algorithm is best tailored to present incrementally the transition towards a generic framework and algorithm for *full Prolog*. Note that it does exactly the same job, nothing more and nothing less.

In the following algorithm, we only specify the procedures which have been modified. The procedures `solve`, `solve_call` and `solve_procedure` remain unchanged, only the procedure `solve_clause` is different. In fact it has been split in three different procedures.

The procedure **solve_clause** executes a single clause on an input pair and returns an abstract sequence with cut information representing the execution of the clause on that input pair. The procedure first transforms the substitution to an abstract sequence whose substitution part is the input substitution extended with the variables occurring in the body of the clause. Afterwards it executes the body of the clause further decomposed in a series of prefixes and terminates by restricting the resulting sequence to the variables occurring in the head of the clause.

The procedure **solve_body** calls itself recursively in order to execute solve_literal with each of the literals forming the body of the clause, accordingly to the execution model of Prolog, from left to right. That is exactly what is done, solve_body calls recursively itself with smaller and smaller prefixes of literals. When only the leftmost literal remains, solve_literal is called with the extended sequence C_{out} as input parameter. And then each of the following literals is processed by solve_literal when unwinding the recursive calls, getting in input the sequence resulting from the execution of the previous literals. These executions of solve_literal — except the very first one, the base inductive case — are however submitted to a conditional test: if the result from the previous literals is bottom, we already know that the current clause fails and there is no need to process the literals left. The condition literal(body) is true if body is constituted of a single literal; the statement pfx.ℓ:=body decomposes body into its rightmost literal and its prefix.

The procedure **solve_literal** executes a literal by choosing the right operation to perform accordingly to its form. If the literal is a cut primitive, only the AI_CUT operation is performed, otherwise the execution of a literal requires three main steps.

- The computation of an abstract substitution representing all the concrete substitutions in C_{out} furthermore restricted to the variables occurring in the literal: this is done by the operation RESTRG.

- If the literal is concerned with unification, the operation AI_VAR or AI_FUNC is performed, depending on the form of the literal. If it is a goal then procedure solve_call is recursively called and the result is retrieved in sat. Moreover, if (β_{in}, p) is in the domain of the dependency graph it is necessary to add a new dependency; otherwise it means that (β_{in}, p) needs to be reconsidered anyway and no dependency needs to be recorded. If the literal is of none of these three forms, then solve_builtin_literal is called and does the same job as all the procedures appended to solve_clause to handle the system predicates (see [BM94]).

- The propagation of the result of the literal to the variables occurring in the body of the clause is made by the EXTGS operation.

```
procedure solve_clause(in $\beta_{in}$, p, c, suspended; out $C_{out}$; inout sat, dp)
begin
    $C_{out}$:= EXT_NOCUT(c, $\beta_{in}$);
    solve_body($\beta_{in}$, p, body(c), suspended, $C_{out}$, sat, dp);
    $C_{out}$:= RESTRC(c, $C_{out}$)
end;
```

```
procedure solve_body (in $\beta_{in}$, p, body, suspended; out $C_{out}$; inout sat, dp)
begin
    if literal(body) then
        solve_literal($\beta_{in}$, p, body, suspended, $C_{out}$, sat, dp)
    else begin
        pfx.$\ell$:= body;
        solve_body ($\beta_{in}$, p, pfx, suspended, $C_{out}$, sat, dp);
        if $C_{out}\downarrow\beta \neq \perp$ then
            solve_literal($\beta_{in}$, p, $\ell$, suspended, $C_{out}$, sat, dp)
    end
end;
```

```
procedure solve_literal(in $\beta_{in}$, p, $\ell$, suspended; out $C_{out}$; inout sat, dp)
begin
    if $\ell$ is ! then
        $C_{out}$:= AI_CUT($C_{out}$)
    else begin
        $\beta_{aux}$:= RESTRG($\ell$, SUBST($C_{out}$));
        switch $\ell$ of
            case $x_j = x_k$ :
                B:= AI_VAR($\beta_{aux}$);
            case $x_j = f(\ldots)$ :
                B:= AI_FUNC($\beta_{aux}$, $f$);
            case q(...) :
                solve_call($\beta_{aux}$, q, suspended, sat, dp);
                B:= sat($\beta_{aux}$, q);
                if ($\beta_{in}$, p) $\in$ dom(dp) then ADD_DP($\beta_{in}$, p, $\beta_{aux}$, q, dp);
            otherwise :
                solve_builtin_literal($\beta_{aux}$, $\ell$, $C_{out}$)
        end;
        $C_{out}$:= EXTGS($\ell$, $C_{out}$, B)
    end
end;
```

Rationale of the new algorithm

We insist on the same functionality between the two version of the algorithm. The revisited one is just another way of doing the same task. Initially solve_clause, solve_body and solve_literal were grouped together into one procedure whose purpose was to process each of the literals forming the body of the clause. The extension and restriction operations were part of that procedure.

Now we have split it in three distinct procedures. The first one copes only with the execution of a head of the clause. It first extends the input substitution, afterwards calls a specific procedure in charge of executing that clause and more specifically its body and finally restricts the resulting sequence to the variables occurring in the head of the clause. In short, what the procedure does is specific to the head of a clause, leaving the body to a specific procedure solve_body.

The key idea of solve_body is not to iterate over the literals in the body of the clause by the means of recursive calls — this would be an overcomplicated way of doing so. The aim of solve_body is to execute a literal in the context resulting from the execution of the literals before it, that is its prefix. We will see that this is an important concept to be exploited later when we present the full Prolog abstract interpretation algorithm. To put it another way, we could consider each literal as a fixpoint immediately reached, since the underlying program is supposed to remain unchanged. Moreover, the embedding of the literals inside the recursive calls allows to enforce the dependency of a literal to its prefix. That is to enforce execution of a literal within the context resulting from the execution of its prefix.

The purpose of solve_literal was to ease the execution of a literal in the resulting context of its prefix by interacting with solve_body.

Examples

On this first example, we reconsider the example exposed in the previous chapter. The program is depicted at Figure 1.1. No comment is necessary, since it is exactly the same trace, except that here, the length of each of the resulting sequences is given. Note that the substitutions are no longer given, since they are the same as in the previous execution trace. Note also that the trace for the first clause has been removed in the second iteration, since it was the same at the first iteration. The point of this example is that without the cut information, the output sequence has an infinite length. An actual computation of a query '← append(Var,Var,Ground)' would give all pairs of sub-lists which, when concatenated, form a list equal to the third argument; there is actually more than one solution in general depending, on the length of the third argument.

```
SOLVE_CALL ITERATION#1: append/3
SOLVE_PROCEDURE
   SOLVE_CLAUSE#1
      EXIT EXTC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
      EXIT AI_FUNC: <min=1, Max=1, ST>
      CALL AI_VAR: <min=1, Max=1, ST>
      EXIT AI_VAR: <min=1, Max=1, ST>
      EXIT RESTRC: <min=1, Max=1, ST>
   EXIT CLAUSE#1: <min=1, Max=1, ST>
   EXIT CONC: (Ground(1):[],Ground(2),Ground(2))
          <min=1, Max=1, ST>
   SOLVE_CLAUSE#2
      EXIT EXTC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
      EXIT AI_FUNC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
      EXIT AI_FUNC: <min=0, Max=1, ST>
      CALL GOAL append: (Var(1),Var(2),Ground(3))
      EXIT GOAL append: bottom
          <min=0, Max=0, SNT>
      EXIT EXTG: <min=0, Max=0, PT>
      EXIT RESTRC: <min=0, Max=0, PT>
   EXIT CLAUSE#2: <min=0, Max=0, PT>
   EXIT CONC: <min=1, Max=1, PT>
EXIT UNION: (Ground(1):[],Ground(2),Ground(2))
   <min=0, Max=1, PT>
```

```
SOLVE_CALL ITERATION#2: append/3
SOLVE_PROCEDURE
   SOLVE_CLAUSE#2
      EXIT EXTC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
      EXIT AI_FUNC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
      EXIT AI_FUNC: <min=0, Max=1, ST>
      CALL GOAL:  append(Var(1),Var(2),Ground(3))
      EXIT GOAL:  append(Ground(1):[],Ground(2),Ground(2))
          <min=0, Max=1, PT>
      EXIT EXTG: <min=0, Max=1, PT>
      EXIT RESTRC: <min=0, Max=1, PT>
   EXIT CLAUSE#2: <min=0, Max=1, PT>
   EXIT CONC:  (Ground(1),Ground(2),Ground(3))
         <min=2, Max=2, PT>
         (Ground(1):[],Ground(2),Ground(2))
         <min=1, Max=1, PT>
EXIT UNION:  (Ground(1),Ground(2),Ground(3))
   <min=1, Max=2, PT>
```

Execution completed:

```
append(Ground(1),Ground(2),Ground(3))
   <min=1, Max=Infinite, PT>
```

Let us consider the append program first presented at Figure 1.3, and now being depicted in normalized form at Figure 5.2. With this framework, we can infer that this version of the program is deterministic. We could also detect with our operation CONC that the abstract sequences resulting from the two clauses are incompatible with respect to the input abstract substitution, if the first argument is ground. Therefore, the analysis will also detect determinacy. Moreover, if the first and third arguments are bound to a variable, our system infers that no result at all is returned. Under the same assumptions and if the two clauses are permuted, our system detects sure non termination.

Recall that the actual implementation is slightly different from what has been exposed in this chapter — see [BLCMVH94], [BM94] and [LCRVH94]. That is why, there are two kinds of sequences processed by the CONC operation in the execution trace.

Figure 5.2 a deterministic program for appending two lists.

```
append(X1, X2, X3):- !,
      X1 = [X4|X5],
      X2 = [X4|X6],
      append(X5, X2, X6).
append(X1, X2, X3):-
      X1 = [],
      X3 = X2.
```

```
SOLVE_CALL ITERATION#1: append/3
SOLVE_PROCEDURE
   SOLVE_CLAUSE#1
      EXIT EXTC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
      EXIT AI_FUNC: <min=1, Max=1, ST>
      CALL AI_FUNC: <min=1, Max=1, ST>
```

```
        EXIT AI_FUNC: <min=0, Max=1, ST>
        CALL CUT:  (Ngv(1):.(Ground(2),Var(3)),Var(4),
            Ground(5):.(Ground(2),Ground(6)),Ground(2),Var(3),Ground(6))
            <min=0, Max=1, ST>
        EXIT CUT:  (Ngv(1):.(Ground(2),Var(3)),Var(4),
            Ground(5):.(Ground(2),Ground(6)),Ground(2),Var(3),Ground(6))
            <min=0, Max=1, ST>
        CALL GOAL append/3:  (Var(1),Var(2),Ground(3))
        EXIT GOAL append/3:  bottom
            <min=0, Max=0, SNT>
        EXIT EXTG: <min=0, Max=0, PT>
        EXIT RESTRC: <min=0, Max=0, PT>
    EXIT CLAUSE#1: <min=0, Max=0, PT>
    EXIT CONC:     SEQ without CUT:  bottom
                       <min=0, Max=0, PT>
                   SEQ with CUT:  bottom
                       <min=0, Max=0, PT>
    SOLVE_CLAUSE#2
        EXIT EXTC: <min=1, Max=1, ST>
        CALL AI_FUNC: <min=1, Max=1, ST>
        EXIT AI_FUNC: <min=1, Max=1, ST>
        CALL AI_VAR: <min=1, Max=1, ST>
        EXIT AI_VAR: <min=1, Max=1, ST>
        EXIT RESTRC: <min=1, Max=1, ST>
    EXIT CLAUSE#2:  (Ground(1):[],Ground(2),Ground(2))
        <min=1, Max=1, ST>
    EXIT CONC:     SEQ without CUT:  (Ground(1):[],Ground(2),Ground(2))
                       <min=1, Max=1, ST>
                   SEQ with CUT:  bottom
                       <min=0, Max=0, PT>
EXIT UNION:  (Ground(1):[],Ground(2),Ground(2))
    <min=0, Max=1, PT>


SOLVE_CALL ITERATION#2: append/3
SOLVE_PROCEDURE
    SOLVE_CLAUSE#1
        EXIT EXTC: <min=1, Max=1, ST>
        CALL AI_FUNC: <min=1, Max=1, ST>
        EXIT AI_FUNC: <min=1, Max=1, ST>
        CALL AI_FUNC: <min=1, Max=1, ST>
        EXIT AI_FUNC: <min=0, Max=1, ST>
        CALL CUT:  (Ngv(1):.(Ground(2),Var(3)),Var(4),
            Ground(5):.(Ground(2),Ground(6)),Ground(2),Var(3),Ground(6))
            <min=0, Max=1, ST>
        EXIT CUT:  (Ngv(1):.(Ground(2),Var(3)),Var(4),
            Ground(5):.(Ground(2),Ground(6)),Ground(2),Var(3),Ground(6))
            <min=0, Max=1, ST>
        CALL GOAL:  append(Var(1),Var(2),Ground(3))
        EXIT GOAL:  append(Ground(1):[],Ground(2),Ground(2))
            <min=0, Max=1, PT>
        EXIT EXTG: <min=0, Max=1, PT>
        EXIT RESTRC: <min=0, Max=1, PT>
    EXIT CLAUSE#1:  (Ground(1):.(Ground(2),Ground(3):[]),Ground(4),
        Ground(5):.(Ground(2),Ground(4)))
```

```
            <min=0, Max=1, PT>
         EXIT CONC:   SEQ without CUT: bottom
                        <min=0, Max=0, PT>
                      SEQ with CUT(Ground(1):.(Ground(2),Ground(3):[]),
                        Ground(4),Ground(5):.(Ground(2),Ground(4)))
                        <min=0, Max=1, PT>
         SOLVE_CLAUSE#2
            EXIT EXTC: <min=1, Max=1, ST>
            CALL AI_FUNC: <min=1, Max=1, ST>
            EXIT AI_FUNC: <min=1, Max=1, ST>
            CALL AI_VAR: <min=1, Max=1, ST>
            EXIT AI_VAR: <min=1, Max=1, ST>
            EXIT RESTRC: <min=1, Max=1, ST>
         EXIT CLAUSE#2:  (Ground(1):[],Ground(2),Ground(2))
            <min=1, Max=1, ST>
         EXIT CONC:   SEQ without CUT:  (Ground(1):[],Ground(2),Ground(2))
                        <min=1, Max=1, ST>
                      SEQ with CUT:  (Ground(1):.(Ground(2),Ground(3):[]),
                        Ground(4),Ground(5):.(Ground(2),Ground(4)))
                        <min=0, Max=1, PT>
      EXIT UNION:  (Ground(1),Ground(2),Ground(3))
         <min=0, Max=1, PT>


      Execution completed:

      append(Ground(1),Ground(2),Ground(3))
         <min=0, Max=1, PT>
```

References

[BLCMVH94] C. Braem, B Le Charlier, S. Modart and P. Van Hentenryck; *Cardinality analysis of Prolog;* Proc. of International Logic Programming Symposium (ILPS'94), Ithaca NY, November 1994, MIT Press.

[BM94] C. Braem, S. Modart; *Abstract interpretation for Prolog with cut: cardinality analysis;* Mémoire de licence et maîtrise en informatique, June 1994.

[LCR94] B. Le Charlier and S. Rossi; *An accurate abstract interpretation framework for Prolog with cut*; Rapport Interne, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, 1993.

[LCRVH94] B. Le Charlier, S. Rossi and P. Van Hentenryck; *An abstract interpretation framework which accurately handles Prolog search rule and the cut;* Proc. of International Logic Programming Symposium (ILPS'94).

Chapter

6

Abstract Interpretation of Full Prolog

« Self-modifying programs are a bygone concept in computer science. Modern programming languages preclude this ability, and good assembly language practice also avoids such programming tricks. It is ironic that a programming language attempting to open a new era in computer programming opens the front door to such arcane techniques, using the predicate `assert` *and* `retract`. *»*

L. Sterling

The framework and algorithm exposed in this chapter are enhanced versions of the two landmark frameworks and algorithms developed in the previous chapters. The generic abstract interpretation algorithm, hereafter presented, has been designed to analyze full Prolog programs, that is to say programs possibly containing system predicates `assert` or `retract`, but also arithmetic or meta-logical predicates — see [BM94]. It takes still advantage of the same features that make the previous algorithms efficient: early detection of termination, detection of redundant computations thanks to a dependency graph and it overcomes the infinite nature of the abstract domains by means of widening techniques.

Some of the operations used previously are enhanced to fit in the new framework. Therefore, this chapter sketches these enhancements but also describes comprehensively the new abstract operations, together with the new algorithm making the originality of our framework.

Contents of this chapter

Concrete Semantics

Abstract interpretation of Prolog would have been complete if we had allowed general clauses to be asserted or retracted. However, this broad approach would have induced severe limitations on the analyzing capacity of the abstract interpreter. Indeed dynamic clauses could themselves contain `assert` or `retract` predicates; therefore asserted clauses could recursively assert or retract other clauses. Such programming techniques are of course not common place in computer science. That is why we took a simpler approach by preventing general clauses to be asserted or retracted, only facts can be.

To put it another way, without this limitation the semantic analysis would have been much more intricate, or even impossible for there would be nothing we could be sure about to reason. A broader approach would bring perverse effects in the analytic precision, since nothing could be sure and abstract interpretation aims at collecting information about all possible executions in once.

Moreover this restriction is not so severe that it could appear at first. Indeed most of the programs asserting 'trivial clauses' can be rewritten to assert facts, such that they can be processed by the abstract interpreter. For instance, you will see later how the lemma construct could be rewritten to comply with the requirements of our generic abstract interpretation framework. Consequently, if we try to analyze a program which asserts general clauses, the analysis fails because of a syntactical error.

In the following, references to dynamic clauses should be understood as references to *dynamic facts*.

Dynamic context

It should be clear by now that the difficulty in static analysis of Prolog programs constructed with `assert` or `retract` predicates lies in the fact that these predicates can modify the underlying program during execution. Hence the point of this framework is to capture this dynamically changing information.

We introduce the concept of dynamic context to ease the comprehension of the abstract semantics. Note that though being a 'pedagogic' aid, dynamic context is not for all that an artificial concept in Prolog area. Similar concepts exist in debugging environment. The dynamic context aims at pointing out how a dynamically changing program interacts with the SLD-Resolution mechanisms.

Intuitively the context symbolizes the universe of discourse. We had not stressed the existence of such contexts in the previous frameworks because the context was considered to be an implicit data; indeed it consisted of the program text which remains unchanged at each execution point. Therefore we could so far implicitly rely on a 'static' context...

Within our perspective, the context may change at any moment of the execution. This '*ever-changing*' nature brings out two problems. The first one is simply the question of how to know the latest context at every moment. This is not a relevant question for it can be solved at the implementation level and the solution amounts to build a kind of repository where the context could be recorded — you will see later that the abstract versions of this context will be collected in the set of abstract tuples together with the input substitution and output substitutions sequence. The second problem is the most relevant; it considers the context under the execution perspective: at which program point should this context be available. Of course under the concrete semantics perspective, it is at the literal level.

Moreover, the operational semantics of the `retract` predicate tells us that the context should be considered under different aspects. Recall that the operational semantics of the predicate is mainly based on four components. The list of asserted clauses which are alive at the time of its *first* execution — these clauses are recorded in what we called the *logical view*. Afterwards further reconsiderations of the predicate will simply remove one at a time all the clauses recorded in its logical view. When there is no more clause left in its view the predicate fails. It could also have failed because none of the alive dynamic clauses unified with its argument at the time of its first execution. Therefore the literal to its left, and more generally the previous resolvent, should be reconsidered in this particular context that made it fail.

To enlighten the ideas, several context components can be derived from this brief summary. There is obviously the first context, the *input* one, which is unique. The predicate when executed within it will produce another context, the *output* one. When this predicate is reconsidered on the event of backtracking, it will always be in a unique context; let us call it the *redo* context. As there might be several reconsiderations, there will be a sequence of redo contexts. Each of them will produce a different output context. Hence there is also a sequence of output contexts. There is ultimately the context in which the predicate fails – let us call it the *failure* context – and in which the previous resolvent should be reconsidered on backtracking. These ideas can be easily generalized to any literal. And hence to any procedure, for a literal is generally resolved by a procedure which defines it – at this point the distinction between procedure and clause is important.

Indeed we cannot state exactly the same facts for any clause of a Prolog program. Let us consider the case of a procedure `p` defined by the clauses c_1, c_2, ..., c_n in that order in the program. When the procedure `p` is *first* executed, this is the clause c_1 which is executed and its input context is the output one of the previous resolvent. As we are considering the first execution of the procedure — that is the first node developed for `p` in the derivation tree — it is impossible to be in a redo context, for redo contexts are only associated with backtracking events. These events concern only the remaining clauses c_i $(2 \le i \le n)$ of `p`, since they will be considered only as an alternative for c_{i-1}, that is to say when this latter fails. Therefore c_2 will be executed in the failure context of c_1, similarly c_3 will be considered in the failure context of c_2, and so on...

Let us summarize the contexts, or the situations, in which clauses could be considered during SLD-Derivation. There is only one input context and no redo context associated with a clause, for it was said that backtracking will never lead to reconsider the same clause, it will rather lead to consider the others clauses defining the procedure which is being executed. There is also a sequence of output contexts, for it was demonstrated previously that among the literals defining a clause, backtracking could happen, and therefore return a series of outputs. There is ultimately the context in which the clause fails and in which the following one will be considered. The fundamental reason for which there is only three context components – the dynamic contexts – associated with clauses and four with literals is intrinsic to them: the body of a clause is a *conjunction* of literals, and the definition of a procedure is a *disjunction* of clauses.

The dynamic contexts will be later formalized by the concept of *dynamic programs*. For now, let us present how this four components interact with the SLD-Resolution mechanisms. Consider the program depicted at Figure 6.1. This is a counter implementation supported by the predicates `assert` and `retract`. They allow to have a counter 'variable' which is a kind of simulation of a `for`-loop control variable in an imperative language, though our counter does not actually control the loop. The counter variable is modeled by the dynamic functor `c/1`. We give at Figure 6.2 a trace of the program execution for the query '← `count(V)`.'.

Figure 6.1
A counter program example of the predicates assert and retract.

```
p(1).       p(2).       p(3).

count(_):-
      asserta(c(0)),
      p(_),
      retract(c(N)),
      M is N+1,
      asserta(c(M)),
      fail.
count(X):-
      call(c(X)).
```

In fact, we only give some fragments of the trace produced during a debugging session. You immediately see the 'existence' of the four contexts. The number in front of each line is the derivation tree depth at which the goal is solved. The '[system]' tag means that the literal is a system predicate. We expose just the first 'iteration' and a few lines more of the second one in order to stress the existence of the counting variable. The existence of the logical view of the retract predicate is also clearly manifested; the predicate can only remove one predicate at a time though other ones have been asserted in the meantime. The propagation of the failure from the fail system predicate to the others above it is also clearly demonstrated. This propagation goes so far as the latest choice point, that is the predicate p(_) in the source program.

The last three lines of the trace highlight the fact that the first clause finitely fails and that the second one is chosen – last but one line – to return the content of the 'control variable'. In this trace is also obvious the fact that a *switch* of context is operated within the SLD-Derivation. Indeed the context in which a literal fails is the one in which the previous literal will be reexecuted. Similarly, the output context of a predicate such as assert(c(0)) is its input one augmented with the new dynamic fact c(0) which is moreover the input one of the next literal.

The context in which a literal is executed has been added to the right of the trace. Note that only the dynamic clauses that are alive are detailed in these contexts. This is not a problem because at most only one clause is alive during the computation. As presented hereafter we notice that only one context is necessary. This is true in an actual computation because only one execution is performed at a time, and consequently the 'same physical' context can be independently used whatever the situation. But remember that the four contexts were presented as a pedagogic instrument to ease the transition towards the abstract semantics. Simply recall that the purpose of abstract interpretation is to collect information about all possible executions at the same time. That is why we will have to consider all the situations in which the literals could be considered during an actual execution.

Figure 6.2 Some fragments of the trace given by a debugging session of the 'counter' program.

```
0 CALL: count(_64)                          {}
1 CALL: asserta(c(0)) [system]              {}
1 EXIT: asserta(c(0))                       {c(0)}
1 CALL: p(_726)                             {c(0)}
1 EXIT: p(1)                                {c(0)}
1 CALL: retract(c(_790)) [system]           {c(0)}
1 EXIT: retract(c(0))                       {}
1 CALL: _1304 is 0+1 [system]               {}
1 EXIT: 1 is 0+1                            {}
1 CALL: asserta(c(1)) [system]              {}
1 EXIT: asserta(c(1))                       {c(1)}
1 CALL: fail [system]                       {c(1)}
1 FAIL: fail                                {c(1)}
1 REDO: asserta(c(1))                       {c(1)}
1 FAIL: asserta(c(1))                       {c(1)}
1 REDO: 1 is 0+1                            {c(1)}
1 FAIL: _1304 is 0+1                        {c(1)}
1 REDO: retract(c(0))                       {c(1)}
1 FAIL: retract(c(_790))                    {c(1)}
1 REDO: p(1)                                {c(1)}
1 EXIT: p(2)                                {c(1)}
1 CALL: retract(c(_790)) [system]           {c(1)}
1 EXIT: retract(c(1))                       {}
1 CALL: _1304 is 1+1 [system]               {}
1 EXIT: 2 is 1+1                            {}
1 CALL: asserta(c(2)) [system]              {}
1 EXIT: asserta(c(2))                       {c(2)}
...
1 CALL: c(_64)                              {c(3)}
1 EXIT: c(3)                                {c(3)}
0 EXIT: count(3)                            {c(3)}
```

Concrete domains

A concrete dynamic program can be seen as a sequence of facts, which evolves dynamically with the execution of the Prolog program. The system predicate **assert** adds facts to the sequence. A fact can be removed from the concrete dynamic program by the system predicate **retract**. The same fact can be present more than once in a concrete dynamic program.

Clause substitution

To be able to make use of the work that has been done previously on abstract interpretation of Prolog, we will introduce the notion of concrete clause substitution and its abstract counterpart. This allows us to define almost all operations on abstract dynamic programs in terms of operations on abstract substitutions, hence reusing the previous abstract domain.

A concrete substitution θ is a *concrete clause substitution* iff it is of the form

$$\{X_1/p(t_1, \ldots , t_n)\}$$

where p is a predicate of arity n and t_1, ..., t_n are terms. We say that p/n, denoted by pred(θ), is the *predicate* of the concrete clause substitution.

A concrete clause substitution represents one concrete fact, which we obtain by applying the concrete clause substitution simply to the variable X_1:

$$\mathtt{X}_1\theta = \mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n).$$

Dynamic program

We will now suggest a formalism for concrete dynamic programs, but let us first introduce an informal view of concrete dynamic programs.

A *concrete dynamic program* is a sequence of facts. With each fact of this sequence is associated a unique identification (called the *time stamp*) and a mark indicating whether the fact is 'alive' or not in the dynamic program. As explained in the first chapter, facts will never be actually removed from the dynamic program by a `retract` operation, but the mark will simply be set to 'not alive'. The use of the time stamp is the following; it is somewhat comparable to the dynamic fact identificator. When a `retract` is *first* executed, it looks for all the facts in the dynamic program which match the retracted pattern, and inserts their time stamps in a list —its *logical view*. Then the fact associated with the first time stamp in the list is unified with the argument of the retract, the fact is marked 'not alive' and the time stamp is removed from the list. At each backtracking step on the `retract`, the same procedure is executed (unification with the fact associated with the first time stamp in the list, set the mark of the fact to 'not alive', remove the time stamp from the list). Moreover it does not matter if the facts associated with the time stamps in the logical view are already marked 'alive' or not. In fact another retract could have marked them 'not alive' before they are treated at some backtracking step. New facts, which have been added to the dynamic program in the meantime, that is after the first execution of the `retract`, will not be considered at any backtracking step because they were not recorded in the logical view of this particular `retract` predicate. Backtracking ends when the list of stamps is empty, that is when the retract fails.

Essentially, we will model a concrete dynamic program by a sequence of tuples of the form $\langle f_1, \mathtt{i}, \mathtt{alive}_i\rangle$, where f_1 is a fact, $\mathtt{i}$ is the time stamp and $\mathtt{alive}_i$ is a Boolean which is true iff f_1 is 'alive' in P. To this sequence we add some information about the next time stamp to assign. Hence, a concrete dynamic program P is a couple

$$\langle\langle\langle f_1, 1, \mathtt{alive}_1\rangle, \ldots, \langle f_n, \mathtt{n}, \mathtt{alive}_n\rangle\rangle, \mathtt{n}+1\rangle$$

where $\mathtt{n}+1$ is the next time stamp to assign.

The *domain* of a concrete dynamic program P, denoted $\mathtt{dom(P)}$, is the set of all the couples $\langle f_1, \mathtt{i}\rangle$ for which alive_i is true. Formally $\mathtt{dom(P)} = \{\langle f_1, \mathtt{i}\rangle : \langle f_1, \mathtt{i}, \mathtt{alive}_i\rangle \in \mathtt{P}{\downarrow}1 \wedge \mathtt{alive}_i\}$.

The *predicate* of a concrete fact f, denoted $\mathtt{pred}(f)$, is formally defined by the following:

$$\mathtt{pred}(f) = \mathtt{p/n}$$

$$\Updownarrow$$

$$f \text{ is of the form } \mathtt{p}(\mathtt{t}_1,\ldots,\mathtt{t}_n)$$

Let us define the function $\mathtt{facts(P)}$, which extracts of P the sequence of facts that are 'alive'. We need this function in the formalism of the concretization function of dynamic programs. Formally, let P be $\langle\langle\langle f_1, 1, \mathtt{alive}_1\rangle, \ldots, \langle f_t, \mathtt{t}, \mathtt{alive}_t\rangle\rangle, \mathtt{t}+1\rangle$, then

$$\texttt{facts(P)} = \langle f_{j_1}, \ldots, f_{j_m} \rangle$$

where $\texttt{m} = \#(\texttt{dom (P)})$
$1 \leq j_1 < \ldots < j_m \leq t$
$\forall k$: $1 \leq k \leq m$, $\langle f_{j_k}, j_k \rangle \in \texttt{dom(P)}$,
$\#(S)$ denotes the cardinal of the set S, i.e. the number of its elements.

Dynamic procedure

The *concrete dynamic procedure* of predicate p/n occurring in a dynamic program P is the sequence of all the facts corresponding to predicate p/n and 'alive' in P. To define this concept formally, we introduce the function proc(P, p/n) which returns the concrete dynamic procedure for predicate p/n of program P. The function proc(P, p/n) extracts of P the longest sub-sequence of facts with predicate p/n.

Formally, let $P = \langle\langle\langle f_1, 1, \text{alive}_1\rangle, \ldots, \langle f_t, t, \text{alive}_t\rangle\rangle, t+1\rangle$ be a concrete dynamic program, then

$$\texttt{proc(P, p/n)} = \langle f_{j_1}, \ldots, f_{j_m} \rangle$$

with $m = \#\{\langle f_i, i\rangle \in \texttt{dom (P)}: \texttt{pred}(f_i) = \texttt{p/n}\}$
$1 \leq j_1 < \ldots < j_m \leq t$
$\texttt{pred}(f_{j_k}) = \texttt{p/n}$, $\langle f_{j_k}, j_k\rangle \in \texttt{dom(P)}$, for $1 \leq k \leq m$

Note that the sub-sequence $\langle f_{j_1}, \ldots, f_{j_m}\rangle$ is empty if (and only if) there is no fact f_i 'alive' in P with $\texttt{pred}(f_i) = \texttt{p/n}$ and that alive_{j_k} is implicitly true for $1 \leq k \leq m$, since in dom(P) there are only elements that are 'alive'.

Concrete unification sequence

For the specification of operation AI_RETRACT we will need the concept of *concrete unification sequence* of a dynamic program. A concrete unification sequence of a dynamic program P for a fact f is the sequence of all the facts unifying with f and 'alive' in P. To define this concept formally, we introduce the function seq_unif(P, f), which returns the concrete unification sequence for fact f of program P. The function seq_unif(P, f) extracts from P the longest sub-sequence of facts, which unify with f.

Formally, let $P = \langle\langle\langle f_1, 1, \text{alive}_1\rangle, \ldots, \langle f_t, t, \text{alive}_t\rangle\rangle, t+1\rangle$ be a concrete dynamic program, then

$$\texttt{seq\_unif}(P, f) = \langle \sigma_{j_1}, \ldots, \sigma_{j_n} \rangle$$

with $n = \#\{\langle f_i, i\rangle \in \texttt{dom (P)}: \exists\, \texttt{mgu}(f, f_i)\}$,
$1 \leq j_1 < \ldots < j_n \leq t$,
$\sigma_{j_k} = \texttt{mgu}(f, f_{j_k})$,
$\langle f_{j_k}, j_k\rangle \in \texttt{dom(P)}$, for $1 \leq k \leq n$.

Concrete operations

Asserting and Retracting Clauses. To be able to specify the abstract operations AI_ASSERT and AI_RETRACT, it is convenient to introduce two more operators $\lhd$ and $\rhd$. Let us specify these two operators. The first operator is used to add a fact at the end of a concrete dynamic program. Assume f is a fact and P is a concrete dynamic program of the form $\langle\langle\langle f_1, 1, \mathtt{alive}_1\rangle, \ldots, \langle f_n, \mathtt{n}, \mathtt{alive}_n\rangle\rangle, \mathtt{n+1}\rangle$, then $\mathrm{P'} = \mathrm{P} \lhd \{ f \}$.

$$\mathrm{P'} = \langle\langle\langle f_1, 1, \mathtt{alive}_1\rangle, \ldots, \langle f_n, \mathtt{n}, \mathtt{alive}_n\rangle, \langle f, \mathtt{n+1}, \mathtt{true}\rangle\rangle, \mathtt{n+2}\rangle.$$

The second operator is used to retract a fact from a concrete dynamic program. Recall that a fact is never actually removed, it is just marked 'not alive' independently from the fact it could already be marked 'not alive'. Assume $\langle f_i, \mathtt{i}, \mathtt{alive}_i\rangle \in \mathrm{P}$, and P is a concrete dynamic program of the form $\langle\langle\langle f_1, 1, \mathtt{alive}_1\rangle, \ldots, \langle f_n, \mathtt{n}, \mathtt{alive}_n\rangle\rangle, \mathtt{n+1}\rangle$, then $\mathrm{P'} = \mathrm{P} \rhd \{\langle f_i, \mathtt{i}\rangle\}$

$$\mathrm{P'} = \langle\langle\langle f_1, 1, \mathtt{alive}_1\rangle, \ldots, \langle f_i, \mathtt{i}, \mathtt{false}\rangle, \ldots, \langle f_n, \mathtt{n}, \mathtt{alive}_n\rangle\rangle , \mathtt{n+1}\rangle.$$

Abstract Semantics

Notations

In the following, let $\kappa_{\mathtt{in}} \in \mathtt{dom(sat)}$ — i.e. the input 'κ'ontext — be the tuple $\langle\beta_{\mathtt{in}}, \pi_{\mathtt{in}}, \pi_{\mathtt{redo}}\rangle$ and $\kappa_{\mathtt{out}} \in \mathtt{codom(sat)}$ be the tuple $\langle \mathrm{B}_{\mathtt{out}}, \pi_{\mathtt{out}}, \pi_{\mathtt{failure}}\rangle$; similarly $\kappa_{\mathtt{out}_c}$ denotes the tuple $\langle \mathtt{C}_{\mathtt{out}}, \pi_{\mathtt{out}}, \pi_{\mathtt{failure}}, \pi_{\mathtt{out}}\rangle$. Moreover, we use the ↓-notation. For instance, $\kappa_{\mathtt{in}}{\downarrow}\beta_{\mathtt{in}}$ denotes the value of the $\beta_{\mathtt{in}}$ component of $\kappa_{\mathtt{in}}$. The ↓-notation is easily extended to the other components of $\kappa_{\mathtt{in}}$, and so on... Similarly $\kappa'_{\mathtt{out}}$ denotes the tuple $\langle \mathrm{B}'_{\mathtt{out}}, \pi'_{\mathtt{out}}, \pi'_{\mathtt{failure}}\rangle$. $\perp_\pi$ denotes $\{\langle\perp, \mathtt{O}, \mathtt{O}\rangle\}$.

The notation $\kappa'_{\mathtt{out}_c} := \kappa_{\mathtt{out}_c} \{\pi_{\mathtt{out}} \leftarrow \perp_\pi\}$ means that $\kappa'_{\mathtt{out}_c}$ is assigned $\kappa_{\mathtt{out}_c}$ where its $\pi_{\mathtt{out}}$ component has been replaced by $\perp_\pi$. In other words all the components of $\kappa'_{\mathtt{out}_c}$ will be identical to their corresponding component of $\kappa_{\mathtt{out}_c}$ except for $\pi'_{\mathtt{out}}$ which becomes $\perp_\pi$. Moreover, all the components of $\kappa_{\mathtt{out}_c}$ remain unchanged.

Dynamic contexts

Recall that the basic idea of the previous frameworks was to consider each predicate defined in the analyzed program as a '*functional mapping*'. So a predicate was mapping an input substitution first to an output substitution and next to an output substitutions sequence. We will obviously keep this idea as the basic principle of this new framework. Consequently we will also keep the same granularity, that is to say the analyzed program is still considered at the procedure level.

However, this functional view of the predicates has to be enriched to take into account the fact that the 'universe of discourse' is changing. As in the concrete semantics this is the aim of the dynamic contexts. These are abstractions of the concrete contexts, whose four components – input, redo, output and failure – will be abstracted by the concept of "*abstract dynamic program*". As before, two of these components are entry contexts, these are the *input* and *redo dynamic program*, and the two others are result contexts, these are the *output* and *failure dynamic program*. Recall that two of these components are singletons and that the two others are sequences. However all of them will be abstracted by the same object, a set of abstract clauses which is the most general abstraction – see later on at the subsection 'Abstract clause'. It remains us to determine the program points where the contexts will be associated with. Because we keep for simplicity the same granularity, they will be associated with each procedure, that is to say with each predicate symbol defined in the program. Each of the abstract programs has therefore to be seen as a way of remembering the dynamically changing universe of discourse at each stage of the computation by giving access to the "sequence" of dynamic clauses — which precisely constitute the *dynamic context* — upon which the current literal is being executed.

These four components present another advantage. As abstract interpretation aims at collecting information about numerous executions, a predicate can be analyzed more accurately under these four 'situations'. Indeed, we can derive properties about the first execution of a predicate, as about its series of reexecutions. We can therefore collect information about all of its possible results and even know what makes it fail. We are that way capable to synthesize information about the four relevant SLD-Resolution mechanisms we pointed out in the previous section. We will now expose precisely how the concrete contexts are abstracted.

Abstract domains

The abstract domains used in this abstract interpreter are derived from the previous ones. Besides handling abstract substitutions and abstract substitutions sequences, they moreover handles abstract dynamic programs.

As was said before, the current algorithm can only handle dynamic facts. A set of dynamic facts can either define a particular dynamic procedure p if all the facts are defined upon the same predicate symbol p or can define a dynamic procedure which is not defining any particular procedure. This undefined procedure is thus susceptible to contain information about any particular dynamic procedure since we do not have accurate information on it. We clearly see now that an abstract dynamic program must contain a set of abstract dynamic procedures relevant to a particular predicate symbol or not. First of all let us introduce the notion of abstract clause.

Abstract clause

An abstract substitution $\beta = \langle \mathtt{sv}, \mathtt{frm}, \mathtt{mode}, \mathtt{ps} \rangle$ is an *abstract clause substitution* – an abstract clause for short – iff

$$\mathtt{dom}(\beta) = \{X_1\}$$

Note that given the usual concretization function for abstract substitutions, we have that every concrete clause substitution belongs to the concretization of at least one abstract clause substitution and that an abstract clause substitution represents a set of concrete clause substitutions.

An abstract clause substitution is *head-defined* iff its head is known. More formally, let $\beta = \langle \mathtt{sv}, \mathtt{frm}, \mathtt{mode}, \mathtt{ps} \rangle$ be an abstract clause substitution. Then β is *head-defined* iff $\mathtt{frm}(\mathtt{sv}(X_1)) \neq \mathtt{undef}$. It is *head-undefined* otherwise.

Let $\beta = \langle \mathtt{sv}, \mathtt{frm}, \mathtt{mode}, \mathtt{ps} \rangle$ be a head-defined abstract clause substitution. Its *predicate*, denoted $\mathtt{pred}(\beta)$, is p/n iff there exist $t_1, \ldots, t_n$ such that $\mathtt{frm}(\mathtt{sv}(X_1)) = p(t_1, \ldots, t_n)$. If β is a head-undefined abstract clause substitution, we define the value of $\mathtt{pred}(\beta)$ as undef.

Abstract dynamic program

An abstract dynamic program is essentially a set of abstract substitutions, which satisfy some properties. Each abstract substitution represents one or more abstract facts representing on their turn sets of concrete facts. Before defining more formally dynamic programs, we need a number of concepts. Assume an abstract domain of abstract substitutions is given for any set D of program variables and let us denote it $AS_{|D}$. Assume also that this domain is endowed with an ordering $\leq$ and a concretization function Cc. Then we can define an *abstract dynamic program* π as being either a set

$$\{\gamma_1, \ldots, \gamma_n\}$$

or a pair

$$\langle\gamma_{undef}, \{\gamma_1, \ldots, \gamma_n\}\rangle$$

where the γ_i $(1 \leq i \leq n)$ are tuples of the form

$$\langle\beta_i, m_i, M_i\rangle$$

and γ_{undef} is a tuple of the form

$$\langle\beta, m, M\rangle.$$

where $\beta_i, \beta \in AS_{|D}$,
$m_i, m \in \mathbb{N}$,
$M_i, M \in \mathbb{N} \cup \{\infty\}$
$m \leq M$,
$m_i \leq M_i$.

Each γ_i describes an *head-defined abstract dynamic procedure* whose facts are described by the β_i, which are *head-defined abstract clause substitutions*. The number of these facts is not less than m_i and not greater than M_i. γ_{undef} describes the *head-undefined abstract dynamic procedure* whose facts are described by β, which is a *head-undefined abstract clause substitution*, also called the *general substitution*. The number of these facts is not less than m and not greater than M. Moreover we have that

$$\forall\, i \neq j: 0 \leq i, j \leq n, \text{pred}(\beta_i) \neq \text{pred}(\beta_j)$$

and

$$\text{pred}(\beta) = \text{undef}.$$

The upper bounds M_i and M can become infinite. This is necessary for our widening operation. Of course, we loose some accuracy because of the widening operation, but the really interesting values of the upper bounds are 0 and 1. If you have an upper bound superior to 1, then —in most cases— it does not matter if this upper bound is 2, 3 or infinite. Anyway, you cannot deduce determinism nor sure failure.

Formally, the meaning of an abstract dynamic procedure is given by the following concretization function. Consider an abstract dynamic procedure $\gamma = \langle\beta, m, M\rangle$, either head-defined or head-undefined. Its concretization function $Cc(\gamma)$ defines a set of facts sequences, i.e. a set of concrete dynamic procedures:

$$Cc(\gamma) = \{\langle f_1,\ldots,f_n\rangle: 1 \leq i \leq n,\ f_i = X_i\theta_i,\ \theta_i \in Cc(\beta)\ ,\ m \leq n \leq M\}$$

Note that if the lower bound m is 0, n can be 0 too, what means that in that case the empty sequence belongs to the set $Cc(\gamma)$. Note also that the θ_i do not need to be different.

The *domain* of an abstract dynamic program, denoted $dom(\pi)$, is the set of predicates of all the head-defined abstract clause substitutions. The *definite part* of an abstract dynamic program, denoted definite(π), is set of all head-defined dynamic procedures of the abstract dynamic program. Formally

$$\text{dom}(\pi) = \{p/r\text{: } \exists!\ i,\ \text{pred}(\beta_i) = p/r\ (1 \leq i \leq n)\},$$

$$\text{definite}(\pi) = \{\gamma_i\text{: } 1 \leq i \leq n\}.$$

Note that in an abstract dynamic program we do not have information on the ordering of facts, nor time stamps, nor a Boolean 'alive'. All this information is abstracted. Our experimental results show that an abstract domain built on this abstraction of concrete dynamic programs leads to a good compromise between performance and accuracy.

Formally, the meaning of an abstract dynamic program is given by the following concretization function Cc(π). This function defines for every abstract dynamic program π a set of concrete dynamic programs. We distinguish three cases according to the form of the abstract dynamic program:

- $\pi = \{\gamma_1, \ldots, \gamma_n\}$,

  $$\text{Cc}(\pi) = \text{Cc}(\{\gamma_1, \ldots, \gamma_n\}) = \{P\text{: } \forall\ \gamma \in \text{definite}(\pi),\ \text{proc}(P, \text{pred}(\gamma)) \in \text{Cc}(\gamma) \wedge \forall\ p/n \notin \text{dom}(\pi)\text{: proc}(P, p/n) = \emptyset\}$$

- $\pi = \langle\gamma_{\text{undef}}, \{\gamma_1, \ldots, \gamma_n\}\rangle$,

  $$\text{Cc}(\pi) = \text{Cc}(\{\gamma_1, \ldots, \gamma_n\}) \cup \{P\text{: facts}(P) \in \text{Cc}(\gamma_{\text{undef}})\}$$

- $\pi = \perp_\pi$,

  $$\text{Cc}(\pi) = \{\}$$

We introduced the special value $\perp_\pi$ to describe in the abstract domain situations, that the concrete execution of the Prolog program will never reach. For instance, if Prolog executes the dynamic predicate assert with a variable as its argument, then the execution of the program fails, but our abstract interpretation algorithm will continue its execution and assign to the abstract dynamic output program the value $\perp_\pi$, meaning that everything after the assert would never be executed by a Prolog interpreter.

Note that the existence of the head-undefined dynamic procedure implies a big loss of information. Consequently, the interesting cases for our analysis are the ones where the head-undefined dynamic procedure is not present. In practice, many tests of Prolog programs from different sources, have demonstrated this tradeoff between complexity and accuracy of the abstract domain on the one hand, and performance of the abstract interpretation on the other hand, to be satisfying.

Notations. Let us introduce some convenient notations. If π is an abstract dynamic program and $p \in \text{dom}(p)$, then we denote by $\pi(p)$ the head-defined abstract dynamic procedure in π corresponding to the predicate p. Formally this means,

$$\pi(p) = \gamma_i = \langle\beta_i, m_i, M_i\rangle \text{ iff } \gamma_i \in \text{definite}(\pi) \text{ and pred}(\beta_i) = p/n$$

We also denote by π(undef) the head-undefined abstract dynamic procedure of π when π is a pair. Moreover, let $\gamma = \langle\beta, m, M\rangle$ be an abstract dynamic procedure. We denote by $\gamma\downarrow\beta$ the substitution part of γ, i.e. β. Similarly we denote by $\gamma\downarrow m$ and $\gamma\downarrow M$, the second and third components of tuple γ. When combining these two notations, we obtain a quite powerful formalism. For instance, to designate the substitution part of the head-defined abstract dynamic procedure corresponding to predicate p/n in the abstract dynamic program π, we can simply write $\pi(\mathrm{p/n})\downarrow\beta$.

Ordering on dynamic programs

$$\pi_1 \le \pi_2 \Leftrightarrow \begin{cases} \mathrm{dom}(\pi_1) \subseteq \mathrm{dom}(\pi_2) \\ \forall \mathrm{p} \in \mathrm{dom}(\pi_1),\ \pi_1(\mathrm{p}) \le \pi_2(\mathrm{p}) \\ \gamma^1_{\mathrm{undef}} \le \gamma^2_{\mathrm{undef}} \text{ if either or both exist} \end{cases}$$

where

$$\gamma^1 \le \gamma^2 \Leftrightarrow \begin{cases} \beta_1 \le \beta_2 \\ \mathrm{m}_1 \ge \mathrm{m}_2 \\ \mathrm{M}_1 \le \mathrm{M}_2 \end{cases}$$

Assertion of a new abstract dynamic procedure

We define now the operation INSERT_DYN_PROC(π, p/n), which inserts a new abstract dynamic procedure with predicate p/n in program π. The inserted procedure is empty. This operation is just used for the manipulation of our abstract data structure. Formally it is specified as follows. Assume $\mathrm{p/n} \notin \mathrm{dom}(\pi)$, then

π' = INSERT_DYN_PROC(π, p/n)
iff $\mathrm{dom}(\pi') = \mathrm{dom}(\pi) \cup \{\mathrm{p/n}\}$,
$\pi'(\mathrm{p/n}) = \langle \perp, 0, 0 \rangle$,
$\pi'(\mathrm{undef}) = \pi(\mathrm{undef})$,
$\forall\ \mathrm{p/n} \in \mathrm{dom}(\pi)$: $\pi'(\mathrm{p/n}) = \pi(\mathrm{p/n})$.

Let us now illustrate the useful concepts of abstract and concrete dynamic programs. Consider the Prolog program depicted at Figure 6.3.

Figure 6.3 First illustration example.

```
p :-  assert(f(a,b,c)),
      assert(a),
      assert(f(d,b,c)),
      assert(a).
```

If we execute this clause with an empty dynamic program at the entry point, then at the end of execution, we get the following concrete dynamic program:

P = ⟨ ⟨⟨f(a,b,c), 1, true⟩, ⟨a, 2, true⟩, ⟨f(d,b,c), 3, true⟩, ⟨a, 4, true⟩⟩, 5⟩

The concrete dynamic program P is abstracted by the following abstract dynamic program:

π = {⟨ground:a, 2, 2⟩, ⟨ground: f(ground,ground:b,ground:c), 2, 2⟩}

Note that, despite that f(a, b, c) and f(d, b, c) are different facts, there is only one entry for both in the abstract dynamic program, since both terms have the same principal functor. Abstraction implies for both facts a loss of information: we loose the information about the pattern of their first argument; we only retain that the first argument of both facts is a ground term.

Consider now the Prolog program depicted at Figure 6.4, illustrating the use of the head-undefined abstract dynamic procedure.

Figure 6.4
Second illustration example.

```
p(a).      p(b).

q:-  p(X),
     assert(X).
```

When executed with the query '← q', the program will give us a first solution {X/a} and a first dynamic program P = ⟨⟨⟨a, 1, true⟩⟩, 2⟩. Then, on the event of backtracking, the second solution {X/b} will be computed and the concrete dynamic program will become, after the execution, P = ⟨⟨⟨a, 1, true⟩, ⟨b, 2, true⟩⟩, 3⟩.

If we execute the abstract interpretation algorithm with the query '← q', the execution of the goal p(X) will produce the following substitution: {X ← ground}. As we see the mode is ground, but there is no pattern associated with the substitution for variable X: the pattern of the substitution is undef (it can be a or b). After the execution of the goal assert(X), this undefined pattern will appear in the abstract dynamic program. The execution result of the query will be the following abstract dynamic program:

$$\pi = \langle\langle\beta, 2, 2\rangle, \{\}\rangle$$

with pred(β) = undef and mode(sv(X_1)) = ground.

Abstract operations

AI_VAR, AI_FUNC and AI_CUT have been enhanced to handle dynamic programs in addition to substitutions sequences. Furthermore, a UNION and a WIDENING operation were specifically designed to handle dynamic programs while the AI_ASSERT, AI_RETRACT, COMBINE and MAKEFUNC operations are brand new ones.

Abstract Unification Operations

These operations are derived from the previous ones and have exactly the same functionality, except that some case analysis has been appended to compute the resulting output contexts π_{out} and $\pi_{failure}$. They moreover rely on the previous versions of the operations AI_VAR, AI_FUNC as defined at page 43.

AI_VAR(ℓ, κ_{in}) = κ_{out}

```
where $\kappa_{in}$ ::= $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$
      $\kappa_{out}$ ::= $\langle B_{out}, \pi_{out}, \pi_{failure}\rangle$
      $B_{out}$ ::= $\langle\beta, m, M, t\rangle$
      $\langle B_{out}, success\rangle$ = AI_VAR($\ell$, $\beta_{in}$)
      $\pi_{out}$ = [if $\beta=\perp$ then $\perp_\pi$ else $\pi_{in}$]
      $\pi_{failure}$ = [if $\beta=\perp$ then $\pi_{in}$
                     else if success then $\pi_{redo}$
                     else $\pi_{in} \cup \pi_{redo}$]
```

AI_FUNC(ℓ, κ_{in}) = κ_{out}

```
where ⟨$B_{out}$, success⟩ = AI_FUNC($\ell$, $\beta_{in}$)
    $\pi_{out}$ = [if $\beta=\bot$ then $\bot_\pi$ else $\pi_{in}$]
    $\pi_{failure}$ = [if $\beta=\bot$ then $\pi_{in}$
                       else if success then $\pi_{redo}$
                       else $\pi_{in} \cup \pi_{redo}$]
```

Recall that one of the key idea of abstract interpretation is that we try to approximate undecidable properties. However Boolean logic is not capable to support approximation of undecidability since we cannot be sure of the approximation in all the cases. That is why in the particular case of static analysis of Prolog programs, we have to distinguish three kinds of computations. There are ones where we are always sure of either a success or a failure of the execution, and more particularly of all possible executions since abstract interpretation wants us to consider an infinite number of executions. And there are ones where we cannot be sure of neither the success nor the failure for all the executions, that is to say that there can be some actual concrete executions where failures will be derived or some others where successes will be derived.

That is what is done in the small case analysis performed at the end of the these operations. In the case where all possible executions fail — that is when the resulting abstract substitution is bottom — the dynamic context must reflect this failure. Therefore the output dynamic program will be assigned bottom and the failure dynamic program will be identical to the input one, reflecting this way that the execution fails in the context given in input. On the other hand if executions always succeed, the output dynamic program has to be simply the one upon which the literal is executed and the failure the same as the redo dynamic program — recall the *'finitely failing assumption'*[5]: later failure could only happen within the redo context, since in the input one, the literal surely succeeds. And finally when there is nothing sure, the most general context must be considered. That is, each of the context for either success or failure has to be taken into account. The output context will thus be the union of both the contexts produced in the case of success and failure. Similarly the failure dynamic program is assigned the union of the input and redo contexts. Note that the result of the union of $\bot_\pi$ and π_{in} is π_{in}, that is why the output context is once more assigned π_{in} in the operations implementation.

Execution of the cut

This operation still makes use of the previous version defined at page 43.

```
AI_CUT($\ell$, $\kappa_{in}$) = $\kappa_{out}$

where ⟨$B_{out}$, success⟩ = AI_CUT($\ell$, $\beta_{in}$)
    $\pi_{out}$ = [if $\beta=\bot$ then $\bot_\pi$ else $\pi_{in}$]
    $\pi_{failure}$ = [if $\beta=\bot$ then $\pi_{in}$ else $\pi_{redo}$]
```

Here, the case analysis is simpler because the cut predicate, if reached, will always succeed, otherwise, we simply propagate the failure.

Unification of two Abstract Clause Substitutions

It is used in operation S_COMBINE to compute the unification of two abstract clause substitutions. We only give its formal specification. The implementation is quite close to the implementation of the unification operation already used in previous frameworks.

[5] See page 18.

Specification: COMBINE(β_1, β_2) = (β_3, suc_unif). Assume dom(β_1) = dom(β_2) = {X1}.

$$\text{suc\_unif} \Leftrightarrow \left.\begin{array}{l}\forall\theta_1 \in Cc(\beta_1)\\ \forall\theta_2 \in Cc(\beta_2)\end{array}\right\}: mgu(f(X_1\theta_1), f(X_1\theta_2)) \neq \emptyset$$

$$\left.\begin{array}{l}\theta_1 \in Cc(\beta_1)\\ \theta_2 \in Cc(\beta_2)\\ \sigma \in mgu(f(X_1\theta_1), f(X_1\theta_2))\end{array}\right\} \Rightarrow \theta_1\sigma, \theta_2\sigma \in Cc(\beta_3)$$

The Boolean suc_unif is true when the unification *surely* succeeds. In other words, it is true iff every pair of concrete substitutions θ_1 and θ_2, respectively abstracted by the clause substitutions β_1 and β_2, has got a most general unifier. Otherwise unification may succeed or not, and thus suc_unif is false.

After a COMBINE, we always need to perform the same sequence of operations: computing the resulting substitutions sequence. That is why we define this new operation S_COMBINE, which is called in AI_RETRACT operation and solve_dynamic_procedure procedure to combine an abstract clause substitution with an abstract procedure, that is to abstract the concrete unifications performed when a dynamic predicate is either executed or retracted.

S_COMBINE receives an abstract clause substitution β_{in} and an abstract dynamic procedure γ which both can be head-defined or not and returns an abstract substitutions sequence B_{out} with three Boolean suc_unif, failure and success. B_{out} is the substitutions sequence resulting of the unification of β_{in} and the substitution part of the procedure γ. The Boolean suc_unif is true iff the unification of β_{in} and the substitution part of γ surely succeeds. The Boolean success is true iff the unification of β_{in} and the substitution part of the procedure is a *sure* success and concrete executions will produce at least one solution, i.e. there is at least one fact defining the procedure. The Boolean failure is true iff the unification of β_{in} and the substitutions part of the procedure is a **sure** failure, i.e. it produces no solution.

Let us just explain the sequence of operations performed in our implementation. S_COMBINE applies operation COMBINE to β_{in} and $\gamma{\downarrow}\beta$, which gives us a resulting substitution and the Boolean suc_unif (see operation COMBINE). This substitution will be the substitution part of the substitutions sequence B_{out}. failure is true iff the substitution computed by COMBINE is $\perp$ or the procedure γ is empty (i.e. the upper bound M is 0). success is true iff the unification in the COMBINE always succeeds (suc_unif is true) and the procedure γ contains at least one element (i.e. the lower bound m is greater or equal than 1).

For the computation of the bounds of the substitutions sequence, we use the two Booleans failure and success. In case of a sure success, we are sure that there are at least as many solutions as the minimum number of facts in the procedure γ. Otherwise we cannot be sure that there is a solution and we have to assign 0 to the lower bound of the substitutions sequence. In case of a sure failure, we are sure, that there is no solution and the upper bound of the sequence is 0. Otherwise there are at most as much solutions as there can be facts in the procedure γ.

```
procedure S_COMBINE(in $\beta_{in}$, $\gamma$; out $B_{out}$, suc_unif, failure, success)
let $B_{out}$ be $\langle\beta$, m, M, st$\rangle$ do begin
    $\beta$:= COMBINE($\beta_{in}$,$\gamma\downarrow\beta$, suc_unif);
    failure:= ($\beta = \bot$ or $\gamma\downarrow$M = 0);
    success:= (suc_unif and $\gamma\downarrow$m $\geq$ 1);
    m:= [if success then $\gamma\downarrow$m else 0];
    M:= [if failure then 0 else $\gamma\downarrow$M]
end;
```

Specialization of a Substitution by a Predicate

This operation has no concrete counterpart, since we need it only because of a particularity of our abstract domain, which is the possibility to have an head-undefined abstract procedure.

The operation builds a head-defined abstract clause substitution from a predicate symbol and a head-undefined abstract clause substitution. Thus, it specializes the head-undefined abstract clause substitution with the predicate.

Operation MAKEFUNC is used in AI_RETRACT operation each time COMBINE computes the unification of a head-defined abstract clause substitution with the head-undefined abstract procedure. Furthermore, it is used in AI_ASSERT operation before taking the union between the head-undefined procedure and a head-defined procedure, because otherwise all the head-defined procedures would become head-undefined.

Specification: MAKEFUNC(β_1, p/n) = β_2. We just give the formal specification of operation MAKEFUNC, the implementation is quite obvious. Assume $\beta_1 = \langle sv_1, frm_1, mode_1, ps_1\rangle$ is a head-undefined abstract clause substitution, then

$$\beta_2 = \langle sv_2, frm_2, mode_2, ps_2\rangle$$

where $dom(\beta_2) = \{X_1\}$,
$dom(frm_2) = dom(mode_2) = dom(ps_2) = \{1, \ldots, n+1\}$,
$sv_2(X_1) = 1$,
$frm_2(1) = p(2, \ldots, n+1)$, $frm_2(i) = undef$ $(2 \leq i \leq n+1)$,
$mode_2(1) = mode_1(1)$, $mode_2(i) = ExtrMode(mode_1(1))$ $(2 \leq i \leq n+1)$,

$$ps_2 = \begin{cases} \{(i, j): 2 \leq i, j \leq n+1\} & \text{if } ExtrMode(mode_1(1)) \neq ground \\ \varnothing & \text{otherwise,} \end{cases}$$

$$ExtrMode(m) = \begin{cases} \bot & \text{if } m \in \{\bot, var\} \\ ground & \text{if } m \in \{ground, gv\} \\ noground & \text{if } m \in \{noground, ngv\} \\ any & \text{otherwise.} \end{cases}$$

Union of two Abstract Dynamic Programs

Union of two Dynamic Procedures. Let γ_1 and γ_2 be two dynamic clauses.

$$\gamma_1 \cup \gamma_2 = \gamma$$

where the components of γ are defined as follows:

$\beta = \beta_1 \cup \beta_2$,
$\mathrm{m} = \min(\mathrm{m}_1, \mathrm{m}_2)$,
$\mathrm{M} = \max(\mathrm{M}_1, \mathrm{M}_2)$.

Union of two Dynamic Programs. Let π^1 and π^2 be two dynamic programs.

$$\pi^1 \cup \pi^2 = \pi$$

where the components of π are defined as follows:

$\mathrm{dom}(\pi) = \mathrm{dom}(\pi^1) \cup \mathrm{dom}(\pi^2)$,
$\quad$ if $\exists\, \gamma^1_{\mathrm{undef}} \vee \exists\, \gamma^2_{\mathrm{undef}}$ then $\gamma_{\mathrm{undef}} = \gamma^1_{\mathrm{undef}} \cup \gamma^2_{\mathrm{undef}}$
$\quad \forall \mathrm{p} \in \mathrm{dom}(\pi^1) \cap \mathrm{dom}(\pi^2),\ \pi(\mathrm{p}) = \pi^1(\mathrm{p}) \cup \pi^2(\mathrm{p})$
$\quad \forall \mathrm{p} \in \mathrm{dom}(\pi^1) \setminus \mathrm{dom}(\pi^2),\ \pi(\mathrm{p}) = \pi^1(\mathrm{p}) \cup \gamma^2_{\mathrm{undef}}\{\beta \leftarrow \mathrm{MAKEFUNC}(\beta, \mathrm{p})\}$
$\quad \forall \mathrm{p} \in \mathrm{dom}(\pi^2) \setminus \mathrm{dom}(\pi^1),\ \pi(\mathrm{p}) = \pi^2(\mathrm{p}) \cup \gamma^1_{\mathrm{undef}}\{\beta \leftarrow \mathrm{MAKEFUNC}(\beta, \mathrm{p})\}$

Widening of two Abstract Dynamic Programs

Widening of two Dynamic Procedures. Let γ_{old} and γ_{new} be two dynamic clauses.

$$\gamma_{\mathrm{old}} \nabla \gamma_{\mathrm{new}} = \gamma$$

where the components of γ are defined as follows:

$\beta = \beta_{\mathrm{old}} \cup \beta_{\mathrm{new}}$,
$\mathrm{m} = \min(\mathrm{m}_{\mathrm{old}}, \mathrm{m}_{\mathrm{new}})$,
$\mathrm{M} = [\textbf{if } \mathrm{M}_{\mathrm{new}} > \mathrm{M}_{\mathrm{old}} \textbf{ then } \infty \textbf{ else } \max(\mathrm{M}_{\mathrm{new}}, \mathrm{M}_{\mathrm{old}})]$.

Widening of two dynamic programs. Let π^{old} and π^{new} be two dynamic programs.

$$\pi^{\mathrm{old}} \nabla \pi^{\mathrm{new}} = \pi$$

where the components of π are defined as follows:

$\mathrm{dom}(\pi) = \mathrm{dom}(\pi^{\mathrm{old}}) \cup \mathrm{dom}(\pi^{\mathrm{new}})$
$\quad$ if $\exists\, \gamma^{\mathrm{old}}_{\mathrm{undef}} \vee \exists\, \gamma^{\mathrm{new}}_{\mathrm{undef}}$ then $\gamma_{\mathrm{undef}} = \gamma^{\mathrm{old}}_{\mathrm{undef}} \nabla \gamma^{\mathrm{new}}_{\mathrm{undef}}$
$\quad \forall \mathrm{p} \in \mathrm{dom}(\pi^{\mathrm{old}}) \cap \mathrm{dom}(\pi^{\mathrm{new}}),\ \pi(\mathrm{p}) = \pi^{\mathrm{old}}(\mathrm{p}) \nabla \pi^{\mathrm{new}}(\mathrm{p})$
$\quad \forall \mathrm{p} \in \mathrm{dom}(\pi^{\mathrm{old}}) \setminus \mathrm{dom}(\pi^{\mathrm{new}}),\ \pi(\mathrm{p}) = \pi^{\mathrm{old}}(\mathrm{p}) \nabla \gamma^{\mathrm{new}}_{\mathrm{undef}}\{\beta \leftarrow \mathrm{MAKEFUNC}(\beta,\mathrm{p})\}$
$\quad \forall \mathrm{p} \in \mathrm{dom}(\pi^{\mathrm{new}}) \setminus \mathrm{dom}(\pi^{\mathrm{old}}),\ \pi(\mathrm{p}) = \gamma^{\mathrm{old}}_{\mathrm{undef}}\{\beta \leftarrow \mathrm{MAKEFUNC}(\beta,\mathrm{p})\} \nabla \pi^{\mathrm{new}}(\mathrm{p})$

Assertion of an Abstract Dynamic Clause

First, we give an example of a concrete dynamic program to show, how Prolog treats the system predicate assert. Then we present a case analysis, which models the behavior of assert. This case analysis allows us to explain the formal specification of our abstract operation AI_ASSERT which is called by the abstract interpretation algorithm each time, the system predicate assert is encountered during the execution. After the specification, we explain, how the behavior of assert can be abstracted using our abstract domain and give the complete implementation of the AI_ASSERT operation.

Consider the following Prolog clause :

```
q:- ..., assert(X), ... .
```

When execution reaches the goal **assert(X)**, the variable **X** will be associated with a substitution θ depending on the previous unifications in the clause. Two cases are possible: either $X\theta$ is a variable or not. In the first case, the execution will fail, since the argument of a dynamic predicate cannot be a variable. In the second case, the goal **assert(X)** will succeed and add the fact $X\theta$ to the dynamic program.

Specification: AI_ASSERT$(\kappa_{in}) = \kappa_{out}$. Recall that κ_{in} and κ_{out} are respectively tuples of the form $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$ and $\langle B_{out}, \pi_{out}, \pi_{failure}\rangle$. β_{in} is an abstract clause substitution representing the argument of the goal **assert**. π_{in} and π_{redo} are respectively the input and redo abstract dynamic programs. The operation generates an abstract substitutions sequence, B_{out}, which represents the solutions of the goal and also computes the output and failure abstract dynamic programs, π_{out} and $\pi_{failure}$. (Remember that we adopt a functional view of the execution of a clause, and a literal in particular, hence the AI_ASSERT operation defines a mapping between its input arguments and its output arguments.)

Let us now specify the AI_ASSERT operation in terms of the concretization function of abstract dynamic programs. Recall that two cases are possible for the computation of the output tuple, κ_{out}, of the operation. If the substitution β_{in} represents a variable, then we have a special situation: the actual execution fails. Hence $P_{failure}$ is the same dynamic program than before the **assert** —i.e. P_{in}— and the output sequence becomes $\perp$. For the same reason, we can ignore P_{out} in this case, what means that $Cc(\pi_{out}) = Cc(\perp_\pi) = \{\}$. Formally, if $\theta \in Cc(\beta_{in})$ and $X_1\theta$ is a variable

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ P_{failure} = P_{in} \\ \Sigma = \langle\perp\rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B_{out}) \end{array}\right.$$

If the substitution does not represent a variable, then the **assert** succeeds and produces exactly one solution. Therefore the output sequence will contain this solution (i.e. the substitution), P_{out} will be assigned P_{in} augmented with the asserted fact and since we succeed, $P_{failure}$ is the same than P_{redo}. Formally, if $\theta \in Cc(\beta_{in})$ and $X_1\theta$ is not a variable

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ P_{out} = P_{in} \lhd \{X_1\theta\} \\ P_{failure} = P_{redo} \\ \Sigma = \langle\theta\rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B_{out}) \end{array}\right.$$

Implementation. We will now distinguish the different cases, that have to be considered in the implementation of the AI_ASSERT according to the arguments. Since in the implementation we have to deal with the abstract domain and in particular have to cope with head-undefined abstract procedures, the case analysis is somewhat more complicated than in the formal specification, but we try to show, that essentially it remains the same — in the following informal case analysis, we follow the same order as in the implementation so that the reader should be able to associate the lines of our implementation to the corresponding explanations of our informal analysis.

Our abstract interpretation algorithm continues its execution, even in cases, where Prolog would have failed (when for instance a variable is asserted). In this cases, we do not really compute a substitutions sequence and dynamic programs, but we simply pass through bottom values for the arguments. So the very first case to consider for AI_ASSERT is when the input context π_{in} is $\perp_\pi$: we just have to generate bottom outputs.

If $\pi_{in} \neq \bot_\pi$ then the computations depend on the fact whether β_{in}, the substitution representing the asserted fact, is a head-undefined or a head-defined clause substitution. Let us consider the latter case first. Since β_{in} is head-defined, we are sure that the argument of system predicate `assert` is not a variable. Hence, according to our formal specification, we generate a substitutions sequence $B_{out} = \langle \beta_{in}, 1, 1, \mathtt{st} \rangle$ and a program $\pi_{failure} = \pi_{redo}$. What remains to do is to approximate safely the abstract operation $\lhd$ to compute π_{out}.

For the implementation of the abstract version of this operation, we distinguish two cases. The first (simple) case is if there exists already a dynamic procedure in π_{in} with the same predicate than the one defined by the abstract clause substitution β_{in}: to obtain π_{out} from π_{in}, we only have to take the union of the clause substitution in the existing dynamic procedure with β_{in} and to increment the lower and upper bounds of the dynamic procedure.

If there is no dynamic procedure in π_{in} with predicate $\mathtt{pred}(\beta_{in})$, then to obtain π_{out} from π_{in} we have to do the following. First we insert a new dynamic procedure for $\mathtt{pred}(\beta_{in})$ using INSERT_DYN_PROC. Then we look at the head-undefined abstract dynamic procedure of π_{in}. If it does not exist we do the same than in the previous case (union of substitutions and incrementing of bounds). Otherwise, this means, that there may already exist a procedure for $\mathtt{pred}(\beta_{in})$ — since the head of the head-undefined procedure is not known, we cannot be sure that it is different from $\mathtt{pred}(\beta_{in})$. Consequently, to make a safe abstraction of operator $\lhd$, we must include the head-undefined procedure in the procedure for $\mathtt{pred}(\beta_{in})$. The MAKEFUNC operation constructs a clause substitution with predicate $\mathtt{pred}(\beta_{in})$ from the head-undefined clause substitution. We assign this substitution to the substitution part of the newly created procedure for $\mathtt{pred}(\beta_{in})$. After having included by this way the information of the head-undefined procedure in the new procedure for $\mathtt{pred}(\beta_{in})$, we can proceed exactly in the same way than in the previous cases.

There remains the case, when the substitution β_{in} is a head-undefined clause substitution. The first thing to do is to test the mode of β_{in}. If its intersection with `novar` is bottom, this means, that the mode of β_{in} is `var` or `bottom`. In both cases we generate — according to the specification — a bottom substitutions sequence, a bottom program π_{out} and the program $\pi_{failure}$ becomes the same than π_{in}.

If the intersection is not bottom, then the mode is possibly different from `var`. We avoid a longer case analysis[6] and assign 0 to the lower bound of the substitutions sequence, since we cannot be *sure* a priori that the mode is not `var` – it can still be `gv` for instance. For the same reason, the program $\pi_{failure}$ becomes the union between π_{in} and π_{redo}.

The concrete operator $\lhd$ is abstracted by the following. First we take the union between the new head-undefined clause substitution and the existing one (if any exists) and then we increment the upper bound by 1. We do not increment the lower bound, since we are not *sure*, that the clause substitution is different from `var`. What remains to do is to propagate the new information in the head-undefined procedure to all the head-defined procedures, since the head-undefined procedure could be a procedure for every head-defined procedure. This is done by the combination of the union operator with the MAKEFUNC operator and by incrementing the upper bound of each head-defined procedure.

Note, that the implementation of AI_ASSERT makes sure, if a head-undefined procedure exists, that the information it contains is also included in all the head-defined proce-

[6] We could distinguish some more cases to increase accuracy, but this is not very useful, since anyway the presence of a head-undefined procedure is source of a big loss of accuracy.

dures[7], since the head-undefined procedure could be a procedure for every head-defined procedure. This property is exploited during the implementation of the operation AI_RETRACT.

```
procedure AI_ASSERT(in $\kappa_{in}$; out $\kappa_{out}$)
let $\kappa_{in}$ be $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$ and $\kappa_{out}$ be $\langle B_{out}, \pi_{out}, \pi_{failure}\rangle$ do
    if $\pi_{in} = \bot_\pi$ then                           {in this case, only pass bottom through}
        $\kappa_{out} := \langle\langle\bot, 0, 0, st\rangle, \bot_\pi, \bot_\pi\rangle$
    else if $pred(\beta_{in}) \neq undef$ then begin             {head-defined clause substitution}
        $\kappa_{out} := \langle\langle\beta_{in}, 1, 1, st\rangle, \pi_{in}, \pi_{redo}\rangle$;
        if $pred(\beta_{in}) \notin dom(\pi_{in})$ then begin   {abstraction of $\lhd$: predicate exists not yet in $\pi_{in}$}
            $\pi_{out}$:= INSERT_DYN_PROC($\pi_{out}$, $pred(\beta_{in})$);
            if $\exists\ \pi_{in}(undef)$ then
                $\pi_{out}(pred(\beta_{in}))\downarrow\beta$:= MAKEFUNC($\pi_{in}(undef)\downarrow\beta$, $pred(\beta_{in})$)
        end;
        let $\pi_{out}$ be $\{\ldots, \gamma_p, \ldots\}$ where $pred(\gamma_p) = pred(\beta_{in})$ do
            let $\gamma_p$ be $\langle\beta, m, M\rangle$ do
                $\gamma_p := \langle\beta \cup \beta_{in}, m+1, M+1\rangle$
    end else                                             {head-undefined clause substitution}
        if $\beta_{in}\downarrow mode \cap novar = \bot$ then
            $\kappa_{out} := \langle\langle\bot, 0, 0, st\rangle, \bot_\pi, \pi_{in}\rangle$
        else begin
            $B_{out} := \langle\beta_{in}, 0, 1, st\rangle$;
            $\pi_{failure} := \pi_{in} \cup \pi_{redo}$;
            $\pi_{out} := \pi_{in}$;                          {begin of abstraction of operator $\lhd$}
            $\pi_{out}(undef) := \pi_{out}(undef) \cup \beta_{in}$;
            $\pi_{out}(undef)\downarrow M := \pi_{out}(undef)\downarrow M+1$;
            forall $p \in dom(\pi_{out})$ do begin
                $\pi_{out}(p)\downarrow\beta := \pi_{out}(p)\downarrow\beta \cup$ MAKEFUNC($\pi_{out}(undef)\downarrow\beta$, p);
                $\pi_{out}(p)\downarrow M := \pi_{out}(p)\downarrow M+1$
            end
        end;
```

Abstract Retraction of an Abstract Dynamic Clause

As for the AI_RETRACT operation, we first give an example using concrete dynamic programs to show, how Prolog treats the dynamic predicate retract. Then we present a case analysis, which models the behavior of retract and allows to explain the formal specification of our abstract operation AI_RETRACT. This operation is called by the abstract interpretation algorithm each time, the system predicate retract is encountered during the execution and abstracts its behavior. After the specification, we explain, how the behavior of retract can be abstracted using our abstract domain and give our implementation of the operation.

Consider the following Prolog clause :

[7] If a new procedure is created and the head-undefined procedure exists, then we initialize the new procedure with the MAKEFUNC of the head-undefined procedure and the new predicate. If the head-undefined procedure is modified, we take the union between all the head-defined procedures and the new head-undefined procedure.

```
q:- ..., retract(X), ... .
```

What will happen when execution reaches the literal `retract(X)`? With variable **X** will be associated a certain substitution θ depending on the previous unifications in the clause. Again, two cases are possible: either $X\theta$ is a variable or not. In the first case, the execution will fail, since the argument of a dynamic predicate cannot be a variable. In the second case, the execution of the literal `retract(X)` will have the following effects: a list of all the facts, that are 'alive' in the dynamic program and unify with the fact $X\theta$, will be constructed as explained before. If this list is empty, then the `retract` fails and the dynamic program does not change. If it contains at least one element, then the `retract` succeeds a first time, the substitution for X is unified with the first fact in the list, this fact is removed from the list and it is marked 'not alive' in the dynamic program. If later backtracking reaches the predicate, the same process is executed.

Specification: $\mathrm{AI\_RETRACT}(\kappa_{in}) = \kappa_{out}$. Exactly as for the AI_ASSERT operation, β_{in} is an abstract clause substitution representing the argument of the system predicate `retract`. π_{in} and π_{redo} are respectively the input and redo abstract dynamic programs, which we have already explained.

The operation generates an abstract substitutions sequence B_{out} which represents the solutions of the goal, and computes the resulting output and failure abstract dynamic programs. Remember that we adopt a functional view of the execution of a clause (and a literal in particular). Hence the AI_RETRACT operation defines a mapping between its input arguments and its output arguments.

Let us now specify formally the AI_RETRACT operation in terms of the concretization function of abstract dynamic programs. For the three outputs computation, three cases have to be distinguished. If the substitution β_{in} represents a variable, then we have a special situation: the execution fails.. Therefore the output sequence becomes $\perp$ and the failure dynamic program is the same than the input program (the execution fails without any manipulation of the dynamic program). For the same reason, we can ignore P_{out} in this case, what means that $Cc(\pi_{out}) = Cc(\perp_\pi) = \{\}$. Formally, if $\theta \in Cc(\beta_{in})$ and $X_1\theta$ is a variable, then

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ P_{failure} = P_{in} \\ \Sigma = \langle\perp\rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B_{out}) \end{array}\right.$$

If the substitution does not represent a variable, then we have to look in the input dynamic program for the facts that are unifying with the argument of the `retract`. If such a fact is found, the `retract` succeeds producing as many solutions as there are unifying facts in the input program. Hence the output sequence has to contain all these solutions. The first unifying fact in P_{in} has to be marked 'not alive' to obtain P_{out} and since we succeed, $P_{failure}$ is the same than P_{redo}. Formally, if

- $\theta \in Cc(\beta_{in})$,
- $\exists \langle f_1, i\rangle \in dom(P_{in})$: $mgu(X_1\theta, f_1) \neq \emptyset$,
- $X_1\theta$ is not a variable

then

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ P_{out} = P_{in} \rhd \{ \langle f_1, i\rangle \} \\ P_{failure} = P_{redo} \\ \Sigma = seq\_unif(P_{in}, X_1\theta) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B_{out}) \end{array}\right.$$

If the substitution does not represent a variable, but if there is no fact in the input dynamic program unifying with the argument of `retract`, then the retract fails. So the substitutions sequence becomes $\perp$, since a concrete execution would not proceed further than the `retract`, but would backtrack to the last choice point. For the same reason, we can ignore P_{out} in this case, what means that $Cc(\pi_{out}) = Cc(\perp_\pi) = \{\}$. The failure dynamic program is the same than the input program, because execution fails without any manipulation of the dynamic program. Formally, if

- $\theta \in Cc(\beta_{in})$
- $X_1\theta$ is not a variable
- $\nexists \langle f_1, i\rangle \in dom(P_{in}): mgu(X_1\theta, f_1) \neq \emptyset$

then

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ P_{failure} = P_{in} \\ \Sigma = \langle \perp \rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B_{out}) \end{array}\right.$$

Implementation. We will now distinguish the different cases that have to be considered in the implementation of the AI_RETRACT according to the arguments. Since in the implementation we have to deal with the abstract domain and in particular have to cope with head-undefined procedures, the case analysis is somewhat more complicated than in the formal specification, but we try to show that it essentially remains the same.

In the following informal case analysis, we follow once more the same order than in the case analysis of the implementation so that the reader should be able to associate the lines of our implementation to the corresponding explanations of our informal analysis. For the AI_RETRACT operation, we have split the code in two procedures RETRACT_DEF and RETRACT_UNDEF which respectively treat the case of a head-defined and of a head-undefined abstract clause substitution β_{in}.

The first thing we do in procedure AI_RETRACT is the treatment of the common case of a bottom input dynamic program. As already explained for AI_ASSERT, our abstract interpretation algorithm continues its execution, even in cases, where Prolog would not (for instance, if a variable is asserted/retracted). In this cases, we do not really compute a substitutions sequence and dynamic programs, but we simply pass through bottom values for the arguments. So, as for the AI_ASSERT, the very first case to consider for AI_RETRACT is when we enter with $\pi_{in} = \perp_\pi$: we just have to generate bottom outputs.

If $\pi_{in} \neq \perp_\pi$ then the computations depend on whether β_{in}, the substitution representing the argument of the `retract`, is a head-undefined or a head-defined clause substitution.

```
procedure AI_RETRACT(in $\kappa_{in}$; out $\kappa_{out}$)
let $\kappa_{in}$ be $\langle \beta_{in}, \pi_{in}, \pi_{redo} \rangle$ and $\kappa_{out}$ be $\langle B_{out}, \pi_{out}, \pi_{failure} \rangle$ do
    if $\pi_{in} = \bot_\pi$ then
        $\kappa_{out} := \langle\langle \bot, 0, 0, st \rangle, \bot_\pi, \bot_\pi \rangle$
    else if pred($\beta$) $\neq$ undef then
        RETRACT_DEF($\kappa_{in}$, $\kappa_{out}$)
    else
        RETRACT_UNDEF($\kappa_{in}$, $\kappa_{out}$);
```

Let us consider the latter case first. It is implemented by procedure RETRACT_DEF. Since β_{in} is head-defined, we are sure that the argument of **retract** is not a variable. Thus, according to our formal specification, we have to look for the facts in the dynamic input program, that are unifying with β_{in}. The only possibly unifying facts are the ones represented by the procedure for pred(β_{in}) and the head-undefined procedure.

If there is neither a head-defined procedure for pred(β_{in}) nor a head-undefined procedure, then we are sure that there is no unifying fact in the dynamic input program. So, the **retract** has to fail: we generate a bottom sequence, the output program is bottom and the failure program is the same as the input one, since the concrete execution does not proceed further than the **retract**.

If there exists a procedure for pred(β_{in}) then we do not worry about the existence of the head-undefined procedure, since we have the property that all the information contained in the head-undefined procedure is already included in the procedure for pred(β_{in}) (see the implementation of the AI_ASSERT). We then perform the unification operation S_COMBINE upon both β_{in} and the procedure for pred(β_{in}). This operation computes the substitutions sequence and three Booleans we explained at the implementation paragraph of operation S_COMBINE. Remember that the Booleans success and failure model a situation of sure success and sure failure respectively, whereas suc_unif stands for the success of the unification only.

The programs π_{out} and $\pi_{failure}$ are calculated by the procedures Comp_Dyn_Out_Def and Comp_Dyn_Fail_Def respectively, which are explained hereafter.

If there is no procedure for pred(β_{in}), but a head-undefined procedure, we create a new procedure for pred(β), since the head-undefined procedure could be a procedure for pred(β_{in}). This new procedure is created in a copy of π_{in}, i.e. π_{in}'. After this, we continue in the same way than in the previous case, but working with the copy of π_{in}.

```
procedure RETRACT_DEF(in $\kappa_{in}$; out kout)
let $\kappa_{in}$ be $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$ and $\kappa_{out}$ be $\langle B_{out}, \pi_{out}, \pi_{failure}\rangle$ do
    if pred($\beta_{in}$) $\notin$ dom($\pi_{in}$) and $\nexists$ $\pi_{in}$(undef) then
        $\kappa_{out}$:= $\langle \langle\perp, 0, 0, st\rangle, \perp_\pi, \pi_{in}\rangle$
    else begin
        $\pi'_{in}$:= $\pi_{in}$;
        if pred($\beta_{in}$) $\notin$ dom($\pi_{in}$) then begin     {pattern not asserted yet, but head-undefined exists}
            $\pi'_{in}$:= INSERT_DYN_PROC($\pi'_{in}$, pred($\beta_{in}$));
            $\pi'_{in}$(pred($\beta_{in}$))$\downarrow\beta$:= MAKEFUNC($\pi_{in}$(undef)$\downarrow\beta$, pred($\beta$));
            $\pi'_{in}$(pred($\beta_{in}$))$\downarrow$M:= $\pi_{in}$(undef)$\downarrow$M
        end;
        $B_{out}$:= S_COMBINE($\beta_{in}$,$\pi'_{in}$(pred($\beta_{in}$)), suc_unif, failure, success);
        $\pi_{out}$:= Comp_Dyn_Out_Def(failure, $\pi'_{in}$, pred($\beta_{in}$), $\pi_{redo}$);
        $\pi_{failure}$:= Comp_Dyn_Fail_Def(failure, success, suc_unif, $\pi'_{in}$, pred($\beta_{in}$), $\pi_{redo}$);
    end;
```

The procedure Comp_Dyn_Out_Def computes the output dynamic program. It receives four input arguments: the Boolean failure already explained above, the predicate symbol of the argument of the retract (remember that we are in the case where β_{in} is a head-defined clause substitution), the input and redo dynamic programs.

According to our specification, we have to distinguish two cases for the computation of the output dynamic program depending on whether there exists or not at least one unifying fact in the input program.

Let us take a look to the latter case. Remember that we treated already the case where there is no procedure for pred(β_{in}) and no head-undefined procedure in the procedure RETRACT_DEF. Remains the case, of an existing head-defined or head-undefined procedure for pred(β_{in}) which is surely not unifying with β_{in}. In this situation the Boolean failure is true and we just generate a bottom output dynamic program.

In every other case, there is at least one unifying fact in the input program. So we have to abstract the behavior of operation $\triangleright$. To do so, we build a dynamic program π'_{in} from π_{in} by decrementing the upper and lower bounds of the procedure for pred(β_{in}). The program π'_{in} is the most accurate output program we can calculate in the case, where we retract at most one fact of the input program. This case is described by the condition $\pi_{in}(p)\downarrow M \leq 1 \vee \pi_{redo} = \perp_\pi$. The first part of the condition is quite obvious: if there is at most one fact in the input program, we cannot retract more than one fact. The second part of the condition, $\pi_{redo} = \perp$, corresponds to an execution context within backtracking is impossible and, therefore without backtracking we cannot retract more than one fact.

If we possibly retract more than one fact, then we can distinguish two more cases for the computation of the lower bound of the procedure for pred(β_{in}) in π_{out} depending on whether there exists or not a procedure for pred(β_{in}) in the redo dynamic program. In the first of these two cases, we can improve the accuracy on the lower bound, since we are sure that the retract will succeed at least two times.

```
**function** Comp_Dyn_Out_Def(**in** failure, $\pi_{in}$, p, $\pi_{redo}$): $\pi_{out}$
**begin**
    **if** failure **then**
        $\pi_{out} := \perp_\pi$
    **else begin**
        $\pi'_{in} := \pi_{in}$;
        $\pi'_{in}(p)\downarrow M := \text{Maximum}(0, \pi_{in}(p)\downarrow M-1)$;
        $\pi'_{in}(p)\downarrow m := \text{Maximum}(0, \pi_{in}(p)\downarrow m-1)$;
        **if** $\pi_{in}(p)\downarrow M \leq 1$ **or** $\pi_{redo} = \perp_\pi$ **then**
            $\pi_{out} := \pi'_{in}$
        **else**
            **if** $p \in \text{dom}(\pi_{redo})$ **then begin**
                $\pi'_{redo} := \pi_{redo}$;
                $\pi'_{redo}(p)\downarrow m := \text{Maximum}(0, \pi_{redo}(p)\downarrow m - (\pi_{in}(p)\downarrow M - 2))$;
                $\pi_{out} := \pi'_{redo} \cup \pi'_{in}$
            **end else**
                $\pi_{out} := \pi_{redo} \cup \pi'_{in}$
        **end**
**end**;
```

The procedure Comp_Dyn_Fail_Def computes the failure dynamic program if β_{in} is a head-defined abstract clause substitution. It receives six arguments: the Booleans suc_unif, failure and success, returned by the operation S_COMBINE, the predicate symbol of the argument of retract, the input and redo dynamic programs.

According to our specification, we have to distinguish three cases for the computation of the failure dynamic program. The first case (β_{in} represents a variable) does not have to be considered here since β_{in} is head-defined. We are in presence of the second case if there exists at least one unifying fact in the input program and the third case is simply the remaining one.

If the Boolean failure is true, then we are sure that there is no unifying fact in π_{in}. Hence, according to our specification, the failure program becomes the same as the input one. If the Boolean success is true, then we are sure that there is at least one unifying fact in π_{in}. Hence, according to our specification, the failure program becomes the same as the redo one. If neither failure nor success is true, then we cannot be sure to be in one of the two last cases of the specification. So we must assign to the failure program the union of the input and redo programs. But if the Boolean suc_unif is true, we have to replace the procedure for p by $\langle\perp, 0, 0\rangle$, since the retract possibly succeeds with the effect of retracting the complete procedure for p.

```
**function** Comp_Dyn_Fail_Def(**in** failure, success, suc_unif, $\pi_{in}$, p, $\pi_{redo}$): $\pi_{failure}$
**begin**
    **if** failure **then**
        $\pi_{failure} := \pi_{in}$
    **else if** success **then**
        $\pi_{failure} := \pi_{redo}$
    **else if** suc_unif **then begin**
        $\pi'_{in} := \pi_{in}$;
        $\pi'_{in}(p) := \langle \perp, 0, 0 \rangle$;
        $\pi_{failure} := \pi'_{in} \cup \pi_{redo}$
    **end else**
        $\pi_{failure} := \pi_{in} \cup \pi_{redo}$
**end**;
```

We will not explain in detail the case where β_{in} is a head-undefined abstract clause substitution. This case is very similar to the one where β_{in} is head-defined. For the sake of completeness we give the implementation of procedure RETRACT_UNDEF.

```
**procedure** RETRACT_UNDEF(**in** $\kappa_{in}$;**out** $\kappa_{out}$)
**let** $\kappa_{in}$ **be** $\langle \beta_{in}, \pi_{in}, \pi_{redo} \rangle$ **and** $\kappa_{out}$ **be** $\langle B_{out}, \pi_{out}, \pi_{failure} \rangle$ **do**
    **if** $\beta_{in}\downarrow$mode $\cap$ novar $= \perp$ **or** $\nexists\ \pi_{in}$(undef) **then**
        $\kappa_{out} := \langle \langle \perp, 0, 0, st \rangle, \perp_\pi, \pi_{in} \rangle$
    **else begin**
        $B_{out}$ := S_COMBINE($\beta_{in}$, $\pi'_{in}$(undef), suc_unif, failure, success);
        $\pi_{out}$ := Comp_Dyn_Out_Undef(failure, $\pi_{in}$, $\pi_{redo}$);
        $\pi_{failure}$ := Comp_Dyn_Fail_Undef(failure, success, suc_unif, $\pi_{in}$, $\pi_{redo}$);
        **forall** p $\in$ dom($\pi_{in}$) do **begin**
            RETRACT_DEF(MAKEFUNC($\beta_{in}$, p), $\pi_{in}$, $\pi_{redo}$, B', $\pi'_{out}$, $\pi'_{failure}$);
            $B_{out} := B_{out} \cup B'$;
            $\pi_{out} := \pi_{out} \cup \pi'_{out}$;
            $\pi_{failure} := \pi_{failure} \cup \pi'_{failure}$
        **end**
    **end**;
```

```
**function** Comp_Dyn_Out_Undef(**in** failure, $\pi_{in}$, p, $\pi_{redo}$): $\pi_{out}$
**begin**
    **if** failure **then**
        $\pi_{out} := \perp_\pi$
    **else begin**
        $\pi'_{in} := \pi_{in}\ \{\pi_{in}(\text{undef}) \leftarrow \langle \beta, \max(0, m-1), \max(0, M-1) \rangle\}$;
        **if** $\pi_{in}(\text{undef})\downarrow M \leq 1$ **or** $\pi_{redo} = \perp_\pi$ **then**
            $\pi_{out} := \pi'_{in}$
        **else begin**
            $\pi'_{redo} := \pi_{redo}\ \{\pi_{redo}(\text{undef}) \leftarrow \langle \beta, \max(0, m - (\pi_{in}(\text{undef})\downarrow M - 2)), M \rangle$;
            $\pi_{out} := \pi'_{redo} \cup \pi'_{in}$
        **end**;
    $\pi_{out} := \pi_{out} \cup \pi_{in}$
    **end**
**end**;
```

```
function Comp_Dyn_Fail_Undef(in failure, success, suc_unif, $\pi_{in}$, p, $\pi_{redo}$): $\pi_{failure}$
begin
    if failure then
        $\pi_{failure} := \pi_{in}$
    else if success then
        $\pi_{failure} := \pi_{redo}$
    else if suc_unif then begin
        $\pi'_{in} := \pi_{in}\ \{\pi_{in}(\text{undef}) \leftarrow \langle \bot, 0, 0 \rangle\}$;
        $\pi_{failure} := \pi'_{in} \cup \pi_{redo}$
    end else
        $\pi_{failure} := \pi_{in} \cup \pi_{redo}$
end;
```

Enhancement of the meta-call

The system predicate call/1 has already been implemented in a previous version of the abstract interpreter (see [BM94]). Now the implementation has to deal with the call of dynamic facts. We give the specification and the implementation of the enhanced abstract operation META_CALL.

Specification: META_CALL(in κ_{in}, B; inout sat, dp) = κ_{out}. We distinguish two cases for the specification of the meta-call. If the argument of the meta-call is *static*, the result is the same as the resolution of the literal represented by its argument. If the argument is a dynamic predicate, the behavior of the meta-call is the same as the one of **retract**, except the fact that the dynamic output program is not computed in the same way. If there exists at least one fact in π_{in} which is unifying with β_{in}, then the dynamic output program is just a copy of π_{in} (the operation $\triangleright$ is thus *not* performed). Otherwise, as for the **retract**, the dynamic output program is ignored.

Formally, if $\mathbf{X}_1\theta$ is a variable, then

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ \theta \in Cc(\beta) \\ P_{failure} = P_{in} \\ \Sigma = \langle \bot \rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B) \end{array}\right.$$

If $\exists\langle f_1, \mathrm{i}\rangle \in \mathrm{dom}(P_{in})$: $\mathrm{mgu}(\mathbf{X}_1\theta, f_1) \neq \emptyset$, then

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ \theta \in Cc(\beta) \\ P_{out} = P_{in} \\ P_{failure} = P_{redo} \\ \Sigma = \mathrm{seq\_unif}(P_{in}, \mathbf{X}_1\theta) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B) \end{array}\right.$$

If $\nexists\langle f_1, \mathrm{i}\rangle \in \mathrm{dom}(P_{in})$: $\mathrm{mgu}(\mathbf{X}_1\theta, f_1) \neq \emptyset$, then

$$\left.\begin{array}{r} P_{in} \in Cc(\pi_{in}) \\ P_{redo} \in Cc(\pi_{redo}) \\ \theta \in Cc(\beta) \\ P_{failure} = P_{in} \\ \Sigma = \langle\bot\rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{l} P_{out} \in Cc(\pi_{out}) \\ P_{failure} \in Cc(\pi_{failure}) \\ \Sigma \in Cc(B) \end{array}\right.$$

Implementation. Let us now consider the implementation of the meta-call[8]. According to our specification, the first case to treat is the one where the argument of the call is a variable term. In this case, we only have to generate bottom outputs. Otherwise we just make a call to the procedure solve_call. But prior to this, we must restrict the substitution to the variable occurring only in the head – operation RESTRG. After the call we add the input pair to the dependency graph since its value has been updated. Then we extend the resulting sequence to obtain the output sequence – EXTG.

```
procedure META_CALL(in $\kappa_{in}$, B; inout sat, dp; out $\kappa_{out}$)
let $\kappa_{in}$ be $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$ and $\kappa_{out}$ be $\langle B_{out}, \pi_{out}, \pi_{failure}\rangle$ and B be $\langle\beta, m, M, t\rangle$ do
begin
    if $\beta\downarrow mode \sqcap var \neq \bot$ then
        $\kappa_{out} := \langle\langle\bot, 0, 0, t\rangle, \bot_\pi, \pi_{in}\rangle$
    else if pred($\beta$) $\neq$ undef then begin
        $\beta' := RESTRG(X_1, \beta)$;
        solve_call($\beta'$, $\pi_{in}$, $\pi_{redo}$, p, suspended, sat, dp);
        $\kappa_{out} := sat(\beta', \pi_{in}, \pi_{redo}, p)$;
        if ($\kappa_{in}$, p) $\in$ dom(dp) then
            ADD_DP($\kappa_{in}$, p, $\beta'$, $\pi_{in}$, $\pi_{redo}$, p, dp);
        $B_{out} := EXTG(X_1, B_{out})$
    end else
        {pattern is undef; this case has not been implemented}
```

Generic Abstract Interpretation Algorithm

Overview of the approach

Recall that the purpose of the algorithm is to compute a subset of a post-fixpoint of a transformation TSAT that includes a tuple of the form (κ_{in}, p, κ_{out}) but as few other elements as possible. To achieve this, the algorithm computes a series of sets $sat_0, \ldots, sat_n$ such that $sat_i < sat_{i+1}$ and such that sat_n contains (κ_{in}, p, κ_{out}). The algorithm moves from one partial set to another by selecting:

- an element (α_{in}, q) which is not yet present in the sat but needs to be computed, or
- an element (α_{in}, q) whose value sat(α_{in}, q) can be improved because the values of some elements upon which it is depending have been updated.

Moreover, the algorithm has now to take into account the eventual changes of the underlying program. Let us present intuitively the approach of the algorithm to achieve this.

[8] We implemented the meta-call only for a defined pattern, since the execution of the meta-call with an unknown pattern entails a big loss of accuracy and needs a complex case analysis. Anyway, all the examples treated did not use a meta-call with unknown pattern.

The principle is, essentially, simple. The computation of the fixpoint values necessary to solve the directing query is driven by a series of local fixpoints over the contexts — this is reflected in the algorithm by a series of fixpoints computed at different granularity level. The rationale of these fixpoints are the following one. When we start the computation we make the hypothesis we know everything which is possible to know – though we know nothing –, until we remark that our knowledge can be improved. We thus reconsider the computation in order to learn everything we can learn until the amount of our knowledge cannot be further improved, that is when the contexts change no more. In this case, we could consider we know everything that is possible to know... This is just a question of precision!

Sets of abstract tuples manipulations

Before specifying the two operations, the part played by the dynamic programs in the abstract interpretation algorithm vis-à-vis the set of abstract tuples must be clear. They do not directly represent in themselves a solution to the query directing the computation. Rather they support the computation, and hence they play a central part in the set of abstract tuples. So these sat-updating operations must take them into account.

Recall that at the contrary of the *abstract semantics* which does not need operations on the set of abstract tuples, the algorithm does need operations to construct a subset of the fixpoint containing the solution of the query. Hence it needs a number of operations on sets of abstract tuples. We keep the same two operations, EXTEND and ADJUST. Each of them is however refined to cater for the requirements of the new framework.

- **EXTEND**(κ_{in}, p, sat) = sat'. Given a set of abstract tuples sat, a predicate symbol p and a tuple κ_{in}, it returns a new set of abstract tuples sat' containing (κ_{in}, p) — supposed not to already belong to sat — in its domain. This operation simply inserts a new pair together with $\perp_\kappa$ as its value, where $\perp_\kappa$ denotes the 3-tuple $\langle\perp_B, \perp_\pi, \perp_\pi\rangle$ with $\perp_B$ being the smallest abstract sequence such that $\{\langle\perp\rangle\} \in \mathrm{Cc}(\perp_B)$ and $\perp_\pi$ being the smallest abstract dynamic program such that $\{\} = \mathrm{Cc}(\perp_\pi)$.

- **ADJUST**(κ_{in}, p, κ_{out}, sat) = sat'. This operation is responsible to update the sat of abstract tuples and to apply the widening operation – denoted ∇ – to the previous and latest result. Given κ_{out} which represents the new result for sat(κ_{in}, p), this operation returns a new set of abstract tuples sat' which is the sat updated in such a way that sat'(κ_{in}, p) = sat(κ_{in}, p) ∇ κ_{out} and all other values stand unchanged.

A slightly more general version of ADJUST is used in the algorithm, it returns besides the new set of abstract tuples, the set of pair (κ_{in}, p) whose values have been updated.

Initially, the algorithm was exploiting as much information as available in the set of abstract tuples when performing those operations. Its latest version neglects these information. At the beginning, the set was implemented as a complete partial order associated to each predicate symbol, as it was presented in the fourth chapter. The complete partial order structure now evolves towards a hash-table structure, for efficiency needs.

Procedure call dependencies

The purposes of a dependency graph is mainly the detection of redundant computations and efficient processing of mutually recursive programs. That is why we keep the same concepts of *dependency graph* and *transitive closure* of the dependency graph.

Recall that the basic intuition is that the dependency graph `dp`(β_{in}, `p`) represents at some point the set of pairs which (β_{in}, `p`) depends directly upon. This is also why we need to define the transitive closure, `trans_dp(bin, p, dp)`, which represents all the pairs which if updated would require reconsidering (β_{in}, p). The same three operations — `EXT_DP`, `ADD_DP` and `REMOVE_DP` — can be reused, without having to be altered.

The generic abstract interpretation algorithm

The top-level procedure **solve** has two input arguments, a tuple κ_{in} and a predicate symbol `p`. It returns the final dependency graph `dp` and the set of abstract tuples `sat` containing (β_{in}, `p`, B_{out}) $\in \mu$(`TSAT`). From these results it is straightforward to compute the set of pairs (α, `q`) used by (β_{in}, `p`), their values in the postfixpoint as well as the abstract substitutions at any program point.

The input tuple κ_{in} contains a substitution β_{in} which together with the predicate symbol `p` form the initial query directing the computation. Moreover, the two other components of κ_{in}, i.e. π_{in} and π_{redo}, are respectively initialized to the empty set and to the bottom element – $\perp_\pi$. The bottom value of the π_{redo} associated with π_{in} means simply that on the event of backtracking, reconsideration of dynamic predicates should always fail because they have not even been executed once, since their input context is empty.

The core of the procedure is a `repeat-until` loop whose termination depends on the stability of the redo dynamic context. In fact we consider that at each stage we know the dynamic context, but we cannot be sure to know it exactly, i.e. we cannot be sure it will not change. That is why we have to iterate until it changes no more. Note that only the redo dynamic program can change. Indeed the input dynamic program is always empty for the top goal, but it is possible that some clauses will be asserted, so the output context may change and further reconsideration of this predicate should take into account these changes. That is why the redo context is updated in order to take into account the eventual latest changes of the output context. Note that this updating is performed by a widening operation because the abstract domain of dynamic programs is infinite.

The procedure **solve_call** receives an abstract substitution and input and redo dynamic contexts embedded in the tuple κ_{in}, its associated predicate symbol `p`, a set `suspended` of pairs (α, `q`), the set of abstract tuples `sat` and a dependency graph `dp`. The set `suspended` contains all pairs (α, `q`) for which a subcomputation has been initiated and not been completed yet. The procedure considers or reconsiders the pair (κ_{in}, `p`) and updates `sat` and `dp` accordingly. The core of the procedure is only executed when (κ_{in}, `p`) is not suspended and not in the domain of the dependency graph. If (κ_{in}, `p`) is suspended, no subcomputation should be initiated. If (κ_{in}, `p`) is in the domain of the dependency graph, this means that none of the elements upon which it is depending have been updated. Otherwise, a new computation with (κ_{in}, `p`) is started. This subcomputation may extend `sat` if it is the first time (κ_{in}, `p`) is considered.

The core of the procedure is a `repeat-until` loop which computes the lower approximation of (κ_{in}, `p`) given the elements of the suspended set. Local convergence is reached when (κ_{in}, `p`) belongs to the domain of the dependency graph. One iteration of the loop amounts to execute solve_`procedure` with the same inputs and `proc(p)` consisting of all the clauses defining the procedure `p`. If the result produced, that is the components of κ_{out}, is greater or incomparable to the current value `sat`(κ_{in}, `p`), then the set of abstract tuples is updated. The dependency graph is also updated accordingly by removing all elements which depends (directly or indirectly) on (κ_{in}, `p`).

```
procedure solve (in $\kappa_{in}$, p; out sat, dp)
begin
    sat:= $\varnothing$;
    dp:= $\varnothing$;
    repeat
        $\pi'_{redo}$:= $\kappa_{in}\downarrow\pi_{redo}$;
        solve_call($\kappa_{in}$, p, $\varnothing$, sat, dp);
        $\kappa_{in}$:= $\kappa_{in}$ $\{\pi_{redo} \leftarrow \pi_{redo} \nabla$ sat($\kappa_{in}$, p)$\downarrow\pi_{out}\}$
    until ($\kappa_{in}\downarrow\pi_{redo} = \pi'_{redo}$)
end;


procedure solve_call(in $\kappa_{in}$, p, suspended; inout sat, dp)
begin
    $\kappa_{in}$:= WIDEN($\kappa_{in}$, p, suspended);
    if ($\kappa_{in}$, p) $\notin$ (dom(dp) $\cup$ suspended) then begin
        if ($\kappa_{in}$, p) $\notin$ dom(sat) then
            sat:= EXTEND($\kappa_{in}$, p, sat);
        repeat
            EXT_DP($\kappa_{in}$, p, dp);
            if p is dynamic then
                solve_dynamic_procedure($\kappa_{in}$, p, $\kappa_{out}$)
            else
                solve_procedure($\kappa_{in}$, p, proc(p), suspended $\cup$ {($\kappa_{in}$, p)}, $\kappa_{out}$,sat,dp);
            (sat, modified):= ADJUST($\kappa_{in}$, p, $\kappa_{out}$, sat);
            REMOVE_DP(modified, dp)
        until ($\kappa_{in}$, p) $\in$ dom(dp)
    end
end;


procedure solve_procedure(in $\kappa_{in}$, p, pr, suspended; out $\kappa_{out}$; inout sat, dp)
begin
    if clause(pr) then begin
        solve_clause($\kappa_{in}$, p, pr, suspended, $\kappa_{out_c}$, sat, dp);
        let $\kappa_{out_c}$ be $\langle C_{out}, \pi_{out}, \pi_{failure}, \pi_{cut}\rangle$ do
            $\kappa_{out}$:= $\langle$SEQ($C_{out}$), $\pi_{out}$, $\pi_{failure} \cup \pi_{cut}\rangle$
    end else begin
        c.sfx:= pr;
        solve_clause($\kappa_{in}$, p, c, suspended, $\kappa_{out_c}$, sat, dp);
        let $\kappa_{out_c}$ be $\langle C_{out}, \pi_{out}, \pi_{failure}, \pi_{cut}\rangle$ do
            if cut_or_nt($C_{out}$) then
                $\kappa_{out}$:= $\langle$SEQ($C_{out}$), $\pi_{out}$, $\pi_{failure} \cup \pi_{cut}\rangle$
            else begin
                $\kappa'_{in}$:= $\Xi$ $\{\pi_{in} \leftarrow \pi_{failure}\}$;
                solve_procedure($\kappa'_{in}$, p, sfx, suspended, $\kappa'_{out}$, sat, dp);
                let $\kappa'_{out}$ be $\langle B'_{out}, \pi'_{out}, \pi'_{failure}\rangle$ do
                    $\kappa_{out}$:= $\langle$CONC($\beta_{in}$, $C_{out}$,$B'_{out}$), $\pi_{out} \cup \pi'_{out}$, $\pi'_{failure} \cup \pi_{cut}\rangle$
            end
    end
end;
```

To execute the procedure defining the predicate p, we now have to check whether the predicate is dynamic or not. If it is dynamic then solve_dynamic_procedure is called, otherwise the 'usual' solve_procedure is called. Note that very few actual parameters are passed to solve_dynamic_procedure because the execution of dynamic procedures — as being formed only of facts — is almost immediate. Note also that the calls to solve_procedure are done with an extended suspended set since a subcomputation has been started with (κ_{in}, p). Before calling solve_procedure, the dependency graph has been brought up to date to include (κ_{in}, p) (which is guaranteed not to be in the domain of the dependency graph before that update). (κ_{in}, p) can be removed from the domain of the dependency graph during the execution of the loop if a pair it is depending upon is updated. The very first statement of solve_call, the widening operation, has also been enhanced to cope with the dynamic contexts; these changes will be explained later on.

The procedure **solve_procedure** recursively solves each of the clauses defining the procedure p. The condition clause(pr) is true when pr consists of only one single clause. The statement c.sfx := pr decomposes pr into its head and its tail. Recall that the procedure also contains an optimization which amounts to test sure non termination or sure execution of a cut in the first clause. In that case the remaining part of the procedure is not executed.

According to the SLD-Resolution mechanisms, a clause is always executed within the failure context of the former one, except for the very first one which is executed within the output context of the previous resolvent — it is obviously assumed that the clauses concerned define the same procedure. That is why, solve_procedure recursively calls itself with the tuple κ_{in} whose dynamic input context π_{in} has been replaced by the failure of the previous clause. Besides concatenating the output substitutions sequence of the current clause with the output of the remaining ones, we also accumulate the output and failure contexts of all the clauses defining the current procedure. Note that, for accuracy purposes, we distinguish from now on two kinds of failure context: the one before a cut has been executed π_{cut} and the one after the execution of a cut $\pi_{failure}$ – see solve_body.

The procedure **solve_clause** executes a single clause on an input pair (κ_{in}, p) and returns an abstract tuple 'with cut information' κ_{out_c} representing the execution of the clause on that input pair. The procedure first transforms the substitution to an abstract sequence whose substitution part is the input substitution extended with the variables occurring in the body of the clause. Afterwards it executes the body of the clause further decomposed in a series of prefixes and terminates by restricting the resulting sequence to the variables occurring in the head of the clause.

The procedure **solve_body** is an enhanced version of the procedure first presented in the previous chapter. It now receives a tuple κ_{in} instead of an abstract substitution and it returns a tuple κ_{out_c} instead of an abstract substitutions sequence C_{out}. Recall that the purpose of the procedure is to process each literal in the body of the current clause accordingly to the SLD-Resolution, i.e. from left to right. The condition literal(body) is true if body is constituted of a single literal; the statement pfx.ℓ:=body decomposes body into its rightmost literal and its prefix. The condition nt(C_{out}) amounts to test sure non termination, and in this case there is no need to consider the following literals.

```
**procedure** solve_clause(**in** $\kappa_{in}$, p, c, suspended; **out** $\kappa_{out_c}$; **inout** sat, dp)
**let** $\kappa_{in}$ **be** $\langle \beta_{in}, \pi_{in}, \pi_{redo} \rangle$ **and** $\kappa_{out_c}$ **be** $\langle C_{out}, \pi_{out}, \pi_{failure}, \pi_{cut} \rangle$ **do begin**
    $C_{out}$:= EXT_NOCUT(c, $\beta_{in}$);
    solve_body($\kappa_{in}$, p, body(c), suspended, $\kappa_{out_c}$, sat, dp);
    $C_{out}$:= RESTRC(c, $C_{out}$);
    **if** first_literal(c) **is** ! **then**
        $\pi_{cut}$:= $\pi_{cut} \cup \pi_{redo}$
**end**;


**procedure** solve_body(**in** $\kappa_{in}$, p, body, suspended; **inout** $\kappa_{out_c}$, sat, dp)
**let** $\kappa_{in}$ **be** $\langle \beta_{in}, \pi_{in}, \pi_{redo} \rangle$ **do begin**
    **if** literal(body) **then begin**
        solve_literal($\kappa_{in}$, p, body, suspended, $\kappa_{out_c}$, sat, dp);
        $\kappa_{out_c}$:= $\kappa_{out_c}$ $\{\pi_{cut} \leftarrow \bot_\pi\}$
    **end else begin**
        pfx.$\ell$:= body;
        $\pi^n_{redo}$:= $\bot_\pi$;
        **let** $\kappa_{out_c}$ **be** $\langle C_{out}, \pi_{out}, \pi_{failure}, \pi_{cut} \rangle$ **and** $\kappa'_{out}$ **be** $\langle C'_{out}, \pi'_{out}, \pi'_{failure} \rangle$ **do begin**
            **repeat**
                $\pi^o_{redo}$:= $\pi^n_{redo}$;
                solve_body($\langle \beta_{in}, \pi_{in}, \pi^o_{redo} \rangle$, p, pfx, suspended, $\kappa_{out_c}$, sat, dp);
                **if** nt($C_{out}$) **then**
                    $\kappa'_{out}$:= $\langle C_{out}, \bot_\pi, \bot_\pi \rangle$
                **else begin**
                    $C'_{out}$:= $C_{out}$;
                    solve_literal($\langle \beta_{in}, \pi_{out}, \pi_{redo} \rangle$, p, $\ell$, suspended, $\kappa'_{out}$, sat, dp);
                    **if** $\ell$ **is not** ! **then**
                        $\pi^n_{redo}$:= $\pi'_{failure}$
                **end**;
                $\pi^n_{redo}$:= $\pi^o_{redo} \nabla \pi^n_{redo}$
            **until** $\pi^o_{redo} = \pi^n_{redo}$;
            $\pi_{cut}$:= [**if** previous_literal($\ell$) **is** ! **then** $\pi_{cut} \cup \pi'_{failure}$ **else** $\pi_{cut}$];
            $\kappa_{out_c}$:= $\langle C'_{out}, \pi'_{out}, \pi_{failure}, \pi_{cut} \rangle$
        **end**
    **end**
**end**;
```

This procedure must be seen as a support for executing the literals in the body of the current clause p. Indeed you must understand clearly the interactions between this procedure and solve_literal. When a literal alters the context upon which it is executed, the following one (the one to the right) must take into account this context change. solve_literal will effect the change but, making a literal execute into the right context is the job of solve_body. A way to achieve this is to make each literal relying on the literal to its left. That is what solve_body does when recursively executing the literals one at a time. Each literal is '*embedded*' inside its prefix of literals, enforcing this way dependencies between literals. This embedding allows to *propagate* the context from a literal to the next one, i.e. the one to its right, when the recursive calls are unwound.

Moreover each literal is considered as a fixpoint. Indeed the core of the procedure is encompassed within a repeat-until loop. The base inductive case, that is the last literal of the body, is an exception to the 'local fixpoint rule' for no literal depends on it, so we can consider that it reaches immediately its fixpoint value. Besides, as it is the last literal we are sure there is no need to have an 'after cut failure'. Nevertheless, the purpose of the repeat-until loop is the same as the one in the solve procedure: to know the most accurately as possible the contexts associated with each of the literals forming the body. Similarly to the top goal, a body must have been first considered before entailing any backtracking. That is why all the literals are considered at first with a bottom dynamic context, meaning that we are collecting information about the first execution of the body and hence, backtracking is still impossible. Once the last literal[9] has been reached, backtracking becomes an eventuality, we can therefore recover the backtracking context of the whole clause. We are now able to collect as much information as possible about the sequence of redo contexts of the clause. Note that the second call to solve_literal clearly demonstrates that each literal is executed within the output context of its immediate neighbor literal, that is the one to its left.

The final results of the clause are the substitutions sequence and the output context resulting from the whole body. The resulting failure context is more subtle. Recall that the failure context is now split in two distinct elements: a 'before cut failure' π_{failure} and an 'after cut failure' π_{cut}. When a cut is executed, we will only backtrack as far as the cut, that is why the failure context – the one that would induce backtracking – of the literal above the cut can be forgotten, but must be propagated when it is not a cut. Hence the value of π^{n}_{redo} is submitted to a condition. Moreover, to prevent loosing the failure context existing at the level of the cut literal, we collect it in π_{cut} — after the repeat-until loop — so that, the complete failure context of the clause could be computed in solve_procedure, by collecting each of fragments of the split failure dynamic program, i.e. π_{failure} and π_{cut}. Similarly, the last condition in solve_clause has been added to prevent loss of the failure context associated with the cut literal, since solve_body fails to do so in the case where the first literal of a clause is a cut.

The procedure **solve_literal** executes a literal by choosing the right operation to perform accordingly to its form. If the literal is a cut primitive, only the AI_CUT operation is performed, otherwise the execution of a literal requires three main steps.

- The computation of an abstract substitution representing all the concrete substitutions in C_{out} furthermore restricted to the variables occurring in the literal: this is done by the operation RESTRG.

[9] In a concrete computation, maybe we would not have to reach the last literal before backtracking, but at the abstract level, it is reasonable to do so, since we do not follow the actual execution model of Prolog, but rather want to collect at the same time as much information as possible.

```
procedure solve_literal(in $\kappa_{in}$, p, l, suspended; inout $\kappa_{out}$, sat, dp)
let $\kappa_{in}$ be $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$ and $\kappa_{out}$ be $\langle C_{out}, \pi_{out}, \pi_{failure}\rangle$ do begin
    if $\ell$ is ! then
        $\kappa_{out}$:= AI_CUT($\kappa_{out}$)
    else begin
        $\kappa'_{in}$:= $\kappa_{in}$ {$\beta_{in} \leftarrow$ RESTRG($\ell$, SUBST($C_{out}$))};
        switch $\ell$ of
            case $X_j = X_k$ :
                $\kappa'_{out}$:= AI_VAR($\kappa'_{in}$);
            case $X_j = f(\ldots)$ :
                $\kappa'_{out}$:= AI_FUNC($\kappa'_{in}$, $f$);
            case asserta, assertz :
                $\kappa'_{out}$:= AI_ASSERT($\kappa'_{in}$);
            case retract :
                $\kappa'_{out}$:= AI_RETRACT($\kappa'_{in}$);
            case q(...) :
                solve_call($\kappa'_{in}$, q, suspended, sat, dp);
                $\kappa'_{out}$:= sat($\kappa'_{in}$, q);
                if ($\kappa_{in}$, p) $\in$ dom(dp) then
                    ADD_DP($\kappa_{in}$, p, $\kappa'_{in}$, q, dp);
            otherwise :
                solve_builtin_literal($\kappa'_{in}$, $\ell$, $\kappa'_{out}$)
        end;
        let $\kappa'_{out}$ be $\langle C'_{out}, \pi'_{out}, \pi'_{failure}\rangle$ do
            $\kappa_{out}$:= $\langle$EXTGS($\ell$, $C_{out}$, $C'_{out}$), $\pi'_{out}$, $\pi'_{failure}\rangle$
    end
end;


procedure solve_dynamic_procedure(in $\kappa_{in}$, p; out $\kappa_{out}$)
begin
    let $\kappa_{in}$ be $\langle\beta_{in}, \pi_{in}, \pi_{redo}\rangle$ and $\kappa_{out}$ be $\langle B_{out}, \pi_{out}, \pi_{failure}\rangle$ do
        if $\pi_{in} = \perp_\pi$ then
            $\kappa_{out}$:= $\langle\langle\perp, 0, 0, st\rangle, \perp_\pi, \perp_\pi\rangle$
        else
            if p $\in$ dom($\pi_{in}$) then begin
                $B_{out}$:= S_COMBINE($\beta_{in}$, $\pi_{in}$(p), suc_unif, failure, success);
                let $\gamma_p \in$ definite($\pi_{in}$) and pred($\gamma_p$)=p be $\langle\beta$, m, M$\rangle$ do begin
                    $\pi^*_{in}$:= $\pi_{in}$ {$\gamma_p \leftarrow \langle\beta$, maximum(1, m), M$\rangle$};
                    $\pi_{out}$:= [if failure then $\perp_\pi$
                                else if M = 1 then $\pi^*_{in}$
                                else $\pi^*_{in} \cup \pi_{redo}$]
                end;
                $\pi^0_{in}$:= $\pi_{in}$ {$\pi_{in}$(undef) $\leftarrow \langle\perp, 0, 0\rangle$};
                $\pi_{failure}$:= [if failure then $\pi_{in}$
                                else if success then $\pi_{redo}$
                                else if suc_unif then $\pi^0_{in} \cup \pi_{redo}$
                                else $\pi_{in} \cup \pi_{redo}$]
            end else
                $\kappa_{out}$:= $\langle\langle\perp, 0, 0, st\rangle, \perp_\pi, \pi_{in}\rangle$
end;
```

- If the literal is concerned with unification, the operations AI_VAR or AI_FUNC are performed, depending on the form of the literal. The literal can also be one of the system predicates asserta, assertz or retract. If it is a goal then procedure solve_call is recursively called and afterwards the result is retrieved in sat. Moreover, if (κ_{in}, p) is in the domain of the dependency graph it is necessary to add a new dependency; otherwise it means that (κ_{in}, p) needs to be reconsidered anyway and no dependency needs to be recorded. If the literal is of none of these three forms, then solve_builtin_literal is called and does the same job as all the procedures appended to solve_clause to handle the system predicates (see [BM94]).
- The propagation of the result of the literal to the variables occurring in the body of the clause is made by the EXTGS operation.

The procedure **solve_dynamic_procedure** is in charge to execute a dynamic fact within its dynamic context, that is the input one. If the input context is bottom, that means that we would face a problem in a concrete computation. Therefore, the dynamic clause will surely not produce any result, we thus propagate this situation and return $\perp_\kappa$, that is $\langle\langle\perp, 0, 0, \mathtt{st}\rangle, \perp_\pi, \perp_\pi\rangle$. On the other hand, if the fact has not been asserted yet, that is does not belong to the domain of π_{in}, we are sure the execution will fail and will not produce any result.

On the other hand, if the fact has already been asserted, the result that its execution will produce is computed by the unification operation S_COMBINE, and both its resulting failure and output contexts depend upon this result. If the unification surely fails – see the definition of S_COMBINE –, it is obvious that the output context will be $\perp_\pi$ to reflect this failure and that the failure context will be the one in which it fails, that is π_{in}. If there is at most one clause defining the fact, the execution will surely give one result. The output program will thus be the input one and will moreover reflect the fact that there is one and only one result – the output program will contain at least and at most one clause defining p. Otherwise, if there are more than one clause defining p, we have to take into account the fact we could be in a backtracking stage.

Similarly, if the clause will surely produce one and only one result, the failure context has to be the redo one, for, only it could lead to a later failure. If we are not sure there will always be a result, then we must take into account both contexts, π_{in} and π_{redo}. However, if there may be concrete computations where the dynamic fact fails, it should be stated that the input context may not even contain a general clause, susceptible to define the procedure p. Ultimately, if we can know nothing precisely, nor sure failure, nor sure success or even both, we return a failure context being the as general as possible.

Termination

The use of widening is useful to limit the number of abstract inputs to be considered. In the case of infinite domain, the algorithm may not terminate. To guarantee the termination, the algorithm relies on a widening technique. The intuition behind this is that an element cannot be refined infinitely often.

Each time a call (κ_{in}, p) is encountered, the last element of the form (κ'_{in}, p) inserted in the suspended set (which should be considered as a stack) is searched. If such an element exists, the computation continues with $(\kappa'_{in} \nabla \kappa_{in}, p)$ instead of (κ'_{in}, p); otherwise the computation proceeds normally.

This processing takes place at the beginning of the procedure solve_call and guarantees that the elements in the suspended stack with the same predicate are always increasing. The WIDEN operation can be defined as follows:

```
**function** WIDEN(**in** $\kappa_{in}$, p, suspended): $\langle \beta^1_{in}, \pi^1_{in}, \pi^1_{redo} \rangle$
**begin**
    $\kappa^0_{in}$:= GET_PREVIOUS(p, suspended);
    **let** $\kappa_{in}$ **be** $\langle \beta_{in}, \pi_{in}, \pi_{redo} \rangle$ **and** $\kappa^0_{in}$ **be** $\langle \beta^0_{in}, \pi^0_{in}, \pi^0_{redo} \rangle$ **do begin**
        $\beta^1_{in} := \beta^0_{in} \cup \beta_{in}$;
        $\pi^1_{in} := \pi^0_{in} \nabla \pi_{in}$;
        $\pi^1_{redo} := \pi^0_{redo} \nabla \pi_{redo}$
    **end**
**end**;
```

The function GET_PREVIOUS returns the tuple κ^0_{in} of the last pair (κ^0_{in}, p) inserted in the stack suspended or $\langle \perp, \perp_\pi, \perp_\pi \rangle$ if there is no such pair.

Examples

First example: counter implementation

Let us reconsider the program depicted at Figure 6.1. We now give some fragments of an execution trace produced by our implementation. The program has been analyzed with the query '← count(Var)'. The final result is first presented to give us an general idea. As we could expect, we see that there is an empty input context, and that each of the three other contexts contain one and only one dynamic fact, c/1. Moreover we also see that the program will produce one and only one result and that it surely terminates. To give you an idea of the number of iterations necessary to analyze this program, we also mention the number of iterations performed by each procedure. Note that these numbers are that low due to some optimizations which are unfortunately outside the scope of this report.

```
INPUT DYNAMIC: empty
REDO DYNAMIC:
    <c/1>: (Ground(1):c(Ground(2)))
        < min=1, max=1 >
OUTPUT DYNAMIC:
    <c/1>: (Ground(1):c(Ground(2)))
        < min=1, max=1 >
FAILURE DYNAMIC:
    <c/1>: (Ground(1):c(Ground(2)))
        < min=1, max=1 >

count (Ground(1))
            < min=1, max=1, ST>

loops  solve      : 2
       call       : 5
       procedure  : 7
       prefix     : 34
```

Now, let us investigate the execution trace. In the following, the 'DYNAMIC SAT' programs denotes the programs such as they are stored in the set of abstract tuples. Each time we enter the procedure solve_body, the prefix of the current literal is displayed since, recall, a clause body is decomposed in series of prefix in order to be executed. Note that the current iteration of the procedure solve is mentioned.

We skip the first series of embedded prefixes of literals until the second **assert** predicate; you clearly see that the fact asserted is ground. The next literal executed is obviously the fail system predicate where you clearly see the effect of the failure: we fail in the input context.

```
SOLVE: LOOP #1

DYNAMIC SAT INPUT: empty
DYNAMIC SAT REDO: bottom
DYNAMIC SAT OUTPUT: bottom
DYNAMIC SAT FAILURE: bottom

CALL IN:
    DYNAMIC INPUT: empty
    DYNAMIC REDO: bottom

    PROCEDURE IN, CLAUSE #1 <count/1>:
        DYNAMIC INPUT: empty
        DYNAMIC REDO: bottom

        CLAUSE/EXTC:
            DYNAMIC INPUT: empty
            DYNAMIC REDO: bottom

        BODY IN:
            DYNAMIC INPUT: empty
            DYNAMIC REDO: bottom
                 asserta(c(0))
                 p(_)
                 retract(c(N))
                 M:=N+1
                 asserta(c(M))
                 fail

        BODY IN:
            DYNAMIC INPUT: empty
            DYNAMIC REDO: bottom
                 asserta(c(0))
                 p(_)
                 retract(c(N))
                 M:=N+1
                 asserta(c(M))
...
            CALL ASSERT:
                 DYNAMIC INPUT: empty
                 DYNAMIC REDO: bottom
            EXIT ASSERT:
                 DYNAMIC OUTPUT:
                      <c/1>: (Ground(1):c(Ground(2)))
                          < min=1, max=1 >
                 DYNAMIC FAILURE: bottom

        BODY OUT:
            DYNAMIC OUTPUT:
                <c/1>: (Ground(1):c(Ground(2)))
            < min=1, max=1 >
    DYNAMIC FAILURE: bottom
    DYNAMIC FAILURE cut: bottom
        asserta(c(0))
        p(_)
        retract(c(N))
        M:=N+1
        asserta(c(M))

    CALL FAIL:
        DYNAMIC INPUT:
            <c/1>: (Ground(1):c(Ground(2)))
                < min=1, max=1 >
        DYNAMIC REDO: bottom

    EXIT FAIL: bottom
                < min=0, max=0, ST >
        DYNAMIC OUTPUT: bottom
        DYNAMIC FAILURE:
            <c/1>: (Ground(1):c(Ground(2)))
                < min=1, max=1 >
```

In the following, the body of the current clause is reconsidered since the redo context has changed. However, this iteration is basically the same as the previous one, that is why we only give the output produced just before the execution of the system predicate fail.

```
BODY IN:
    DYNAMIC ACC INPUT: empty
    DYNAMIC ACC REDO:
        <c/1>: (Ground(1):c(Ground(2)))
            < min=1, max=1 >
     UNIF-FUNC 0
     UNIF-FUNC c
        asserta(c(0))
        p(_)
        retract(c(N))
        M:=N+1
        asserta(c(M))

BODY OUT:
    DYNAMIC OUTPUT:
        <c/1>: (Ground(1):c(Ground(2)))
            < min=1, max=1 >
    DYNAMIC FAILURE:
        <c/1>: (Ground(1):c(Ground(2)))
            < min=1, max=1 >
    DYNAMIC FAILURE cut: bottom
        asserta(c(0))
        p(_)
        retract(c(N))
        M:=N+1
        asserta(c(M))

    CALL FAIL:
            DYNAMIC INPUT:
                <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >
            DYNAMIC REDO: bottom

        EXIT FAIL:
            DYNAMIC INPUT:
                <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >
            DYNAMIC REDO: bottom
            DYNAMIC OUTPUT: bottom
            DYNAMIC FAILURE:
                <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >

    BODY OUT:
        DYNAMIC INPUT: empty
        DYNAMIC REDO: bottom
        DYNAMIC OUTPUT: bottom
        DYNAMIC FAILURE:
             <c/1>: (Ground(1):c(Ground(2)))
                 < min=1, max=1 >
        DYNAMIC FAILURE cut: bottom
            asserta(c(0))
            p(_)
            retract(c(N))
            M:=N+1
            asserta(c(M))
            fail

    CLAUSE/RESTRC: bottom
                < min=0, max=0, ST >
        DYNAMIC OUTPUT: bottom
        DYNAMIC FAILURE:
             <c/1>: (Ground(1):c(Ground(2)))
                        < min=1, max=1 >
        DYNAMIC FAILURE cut: bottom
```

Immediately follows the result of the first clause after the first iteration. Obviously the clause produces no result because of the fail system predicate.

```
PROCEDURE OUT, CLAUSE #1 <count/1>: bottom
          < min=0, max=0, ST >
  DYNAMIC OUTPUT: bottom
  DYNAMIC FAILURE:
      <c/1>: (Ground(1):c(Ground(2)))
                < min=1, max=1 >
  DYNAMIC FAILURE cut: bottom
...
```

For the second clause, since it contains only one literal, we immediately expose its execution. The input context of this clause is obviously the failure of the previous one, and therefore the clause surely succeeds since there is one asserted clause in the input context.

```
    PROCEDURE IN, CLAUSE #2 <count/1>:
        DYNAMIC INPUT:
            <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >
        DYNAMIC REDO: bottom
...
            CALL GOAL:     (Var(1))
                        < min=1, max=1, ST>
                DYNAMIC INPUT:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >
                DYNAMIC REDO: bottom

            EXIT GOAL:     (Ground(1))
                        < min=1, max=1, ST>
                DYNAMIC INPUT:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >
                DYNAMIC REDO: bottom
                DYNAMIC OUTPUT:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >
                DYNAMIC FAILURE: bottom
...
    PROCEDURE OUT, CLAUSE #2 <count/1>: (Ground(1))
            < min=1, max=1, ST>
        DYNAMIC OUTPUT:
            <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >
        DYNAMIC FAILURE: bottom
        DYNAMIC FAILURE cut: bottom
...
ADJUST:  (Ground(1))
            < min=1, max=1, ST>
    DYNAMIC OUTPUT:
        <c/1>: (Ground(1):c(Ground(2)))
                < min=1, max=1 >
    DYNAMIC FAILURE: bottom
```

It is worth noticing that the redo context used in solve_call is the previous redo widened by the output context produced during the first iteration. Note that the first clause will produce the same result, we therefore omit its execution trace. Indeed the same redo context was taken into account previously.

```
SOLVE: LOOP #2
    DYNAMIC SAT INPUT: empty
    DYNAMIC SAT REDO: bottom
    DYNAMIC SAT OUTPUT:
        <c/1>: (Ground(1):c(Ground(2)))
            < min=1, max=1 >
    DYNAMIC SAT FAILURE: bottom

CALL IN:
    DYNAMIC INPUT: empty
    DYNAMIC REDO:
        <c/1>: (Ground(1):c(Ground(2)))
            < min=1, max=1 >

    PROCEDURE IN, CLAUSE #1 <count/1>:
        DYNAMIC INPUT: empty
        DYNAMIC REDO:
            <c/1>: (Ground(1):c(Ground(2)))
                        < min=1, max=1 >
...
    PROCEDURE OUT, CLAUSE #1 <count/1>: bottom
                < min=0, max=0, ST >
        DYNAMIC OUTPUT: bottom
        DYNAMIC FAILURE:
            <c/1>: (Ground(1):c(Ground(2)))
                        < min=1, max=1 >
        DYNAMIC FAILURE cut: bottom
```

For this second clause, the input context has changed, therefore, we give, still briefly, its execution trace. Note that the result is not fundamentally different, except the existence of a failure context.

```
    PROCEDURE IN, CLAUSE #2 <count/1>:
        DYNAMIC INPUT:
            <c/1>: (Ground(1):c(Ground(2)))
                        < min=1, max=1 >
        DYNAMIC REDO:
            <c/1>: (Ground(1):c(Ground(2)))
                        < min=1, max=1 >
...
            CALL GOAL: (Var(1))
                        < min=1, max=1, ST>
                DYNAMIC INPUT:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >
                DYNAMIC REDO:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >

            EXIT GOAL: (Ground(1))
                        < min=1, max=1, ST>
                DYNAMIC OUTPUT:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >
                DYNAMIC FAILURE:
                    <c/1>: (Ground(1):c(Ground(2)))
                                < min=1, max=1 >
...
    PROCEDURE OUT, CLAUSE #2 <count/1>: (Ground(1))
            < min=1, max=1, ST>
        DYNAMIC OUTPUT:
            <c/1>: (Ground(1):c(Ground(2)))
                        < min=1, max=1 >
        DYNAMIC ACC FAILURE:
            <c/1>: (Ground(1):c(Ground(2)))
```

```
                    < min=1, max=1 >
        DYNAMIC ACC FAILURE cut: bottom
...
ADJUST:  (Ground(1))
            < min=1, max=1, ST>
    DYNAMIC OUTPUT:
        <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >
    DYNAMIC FAILURE:
        <c/1>: (Ground(1):c(Ground(2)))
                    < min=1, max=1 >
```

Second example: Fibonacci function

Let us also reconsider the Fibonacci program first depicted at Figure 1.5, and now rewritten to comply with our frameworks requirements, Figure 6.5. We do not give an execution trace but only the final result of the abstract execution.

Figure 6.5 The Fibonacci rewritten to comply with our framework.

```
fibonacci(X,Y):-
      call(lemma(X,Y)), !.
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,R):-
      N >= 2,
      N1 is N-1,
      fibonacci(N1,R1),
      asserta(lemma(N1,R1)),
      N2 is N-2,
      fibonacci(N2,R2),
      R is R1+R2.
```

The abstract interpreter can concludes the program is deterministic since it produces zero or one solution, which is moreover ground. Of course, there can be an infinity of asserted facts because we cannot conclude exactly how many facts will be asserted since it depends upon a particular concrete execution.

```
DYNAMIC SAT INPUT: empty
DYNAMIC SAT REDO:
    <lemma/2>: (Ground(1):lemma(Ground(2),Ground(3)))
        < min=0, infinite >
DYNAMIC SAT OUTPUT:
    <lemma/2>: (Ground(1):lemma(Ground(2),Ground(3)))
        < min=0, infinite >
DYNAMIC SAT FAILURE:
    <lemma/2>: (Ground(1):lemma(Ground(2),Ground(3)))
        < min=0, infinite >
```

```
fibonacci (Ground(1),Ground(2))
   < min=0, max=1, PT> <X1,GEQ_CST,0>

loops solve : 2          time: 1.36
call        : 41
procedure   : 144
prefix      : 634
```

To demonstrate the accuracy of the analysis, let us present some figures detailing, among other things, the number of items created in the sat, as the number of procedures or calls which are deterministic, surely fails, surely succeeds and so on... In fact you will see that we have the best results possible to have with respect to our framework.

Table 6.1 Statistics about the SAT.

predicates	stable	foundation	SAT
fibonacci/2	13	4	14
totals	**13**	**4**	**14**
means	13	4	14

Table 6.2 Input/Output patterns.

Input predicates	Output predicates
fibonacci/2: (**Ground**, Var)	fibonacci/2: (**Ground**, **Ground**) < min=0, max=1, PT>

Table 6.3 ProceduresCalls statistics.

Procedures		**Calls**	
deterministic	1	deterministic	4
failing	0	failing	0
successful	0	successful	0
looping	0	looping	0
sure terminating	0	sure terminating	0
inifinity of solutions	0	inifinity of solutions	0
	0		0
total procedures	**1**	**total procedures**	**4**

Table 6.4 Modes statistics.

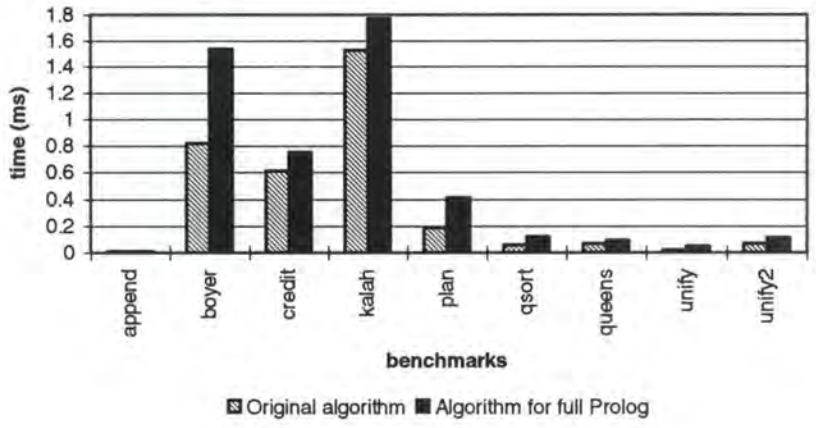
Input				**Output**			
Ground	1	Bottom	0	Ground	2	Bottom	0
NoVar	0	Ngv	0	NoVar	0	Ngv	0
Var	1	NoGround	0	Var	0	NoGround	0
		Gv	0			Gv	0
		Any	0			Any	0
Total	2	Total	0	Total	2	Total	0

Experimental results

Let us now present some experimental results of the implementation of the new abstract interpretation framework. As mentioned, to our knowledge there exists no other framework capable to analyze *full* Prolog (with system predicates **assert** and **retract**). We thus can not compare the efficiency of our results with another framework. Nevertheless we try to give an idea of the effect of the newly implemented features on the performance of the algorithm.

First we compare the performance of the previous version of the abstract interpretation algorithm with the new one by executing both on logic programs that do not contain one of the system predicates **assert** or **retract**.

Figure 6.6 Comparison between the Execution times.



As you can see, the overhead in execution time is small for all benchmarks, except for **boyer**. The explanation is that this benchmark uses the meta-call. Indeed, our algorithm implements some optimization techniques and in particular detects *pure* predicates statically before the execution of **solve_call**, i.e. predicates which use neither directly nor indirectly the system predicates **assert** or **retract**. When a meta-call is used in a clause, the static analysis automatically considers the predicate defined by this clause as *not* pure. For this reason, the optimizations implemented for pure predicates do not work for Boyer. Note, that a more refined static analysis of the program could detect cases for which we are sure that the meta-call will never have a dynamic predicate as its argument (for instance if the program does not contain any **assert** or **retract** at all).

Consider now two Prolog programs computing the Fibonacci function. The first one is depicted at **Figure 1.5**. It uses a memo-function to save the results of subcomputations and avoid unnecessary computations. The second one is depicted at Figure 6.7. The only difference between the first and the second program is that the latter does not make use of the memo-function, what makes it less efficient.

Figure 6.7 Fibonacci program without dynamic predicates.

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,R):-
        N >= 2,
        N1 := N - 1,
        fibonacci(N1, R1),
        N2 := N - 2,
        fibonacci(N2, R2),
        R := R1 + R2.
```

We performed the following executions: the second program with the original version of the algorithm described in [BM94]. (line *FibOA*), the second program with the new algorithm (line *FibNA*) and the first program with the new algorithm (line *FibMemNA*). For each execution, we show the number of iterations in the procedures solve, solve_call, solve_procedure, solve_prefix and the execution time, respectively.

Table 6.5 Different executions on Fibonacci.

Benchmark	solve	call	procedure	prefix	time (ms)
FibOA	-	-	-	-	0.03
FibNA	1	3	9	27	0.03
FibMemNA	2	41	144	634	1.20

As you can see there is no difference between the execution times of the program without memo-function for both versions of the algorithm. But the execution time explodes for the analysis of the program using the memo-function and this although the only difference between both programs is the presence of system predicate assert. If we compare the number of iterations in the different procedures, we discover the source of the important execution time overhead. In procedure solve we have to iterate two times now (since at the first iteration we do not know the redo program) and in procedure solve_prefix we have to iterate much more to compute the local fixpoints, which are not immediately reached anymore, since the redo program has to be approximated by the successive iterations.

Let us take a look to the two results depicted in figure Figure 6.8. cs is a benchmark that was already performed on the original framework (see [BM94]). In fact, in cs we had to simulate the system predicates assert and retract, since the original framework was not able to treat them. In cs_real we use these system predicates instead of simulating them. This allowed us to improve the precision of the results. Indeed, as you can see, for cs_real the analysis concludes that there is at most one solution, where for cs it cannot give an upper bound for the number of solutions. Note, that this gain in precision entails an important consequence on the performance.

Figure 6.8

```
cs
pgenconfig (Ground(1))
        < min=0, infinite, PT >
        time: 1.75

cs_real
pgenconfig (Ground(1))
        < min=0, max=1, PT>
        time: 7.18
```

Consider now the Table 6.6. We give it just as an illustration, because we cannot compare it to other frameworks. It shows you for four benchmarks the number of iterations in the procedures solve, solve_call, solve_procedure, solve_prefix, the number of lines of the program and the execution time. Mastermind is program taken from [STE94]. plm is a compiler .

Benchmark	solve	call	procedure	prefix	lines	time (ms)
classy	2	149	350	1046	189	2.93
mastermind	2	138	215	637	77	2.66
plm	2	3021	14457	55289	1488	263.40
tp	1	46	68	134	36	0.24

Table 6.6

References

[BM94] C. Braem, S. Modart; *Abstract interpretation for Prolog with cut: cardinality analysis;* Mémoire de licence et maîtrise en informatique, June 1994.

Conclusion

In this report, we have described the implementation of a novel framework for the abstract interpretation of *full* Prolog including not only the system predicates `asserta`, `assertz` and `retract`, but also arithmetic and meta-predicates, other built-ins like `abolish` or `retractall` and last but not least the cut primitive. The only limitation of our framework is that it is not able to treat the assertion and retraction of general clauses. The main practical and theoretical results related to this research are presented in this report. The framework was instantiated to a new abstract domain allowing us to extend the class of programs that can be analyzed by the abstract interpretation algorithm, to any kind of programs system predicates.

The new framework has a small execution overhead in comparison with the previous one for *pure* programs, i.e. programs not using the system predicates `asserta`, `assertz` and `retract`. For programs using these system predicates, the overhead of execution time is more important but still acceptable. Indeed, the performance of the abstract interpretation algorithm is reasonable even for large programs (counting up to 1500 lines and using intensively `asserta`, `assertz` and `retract`). But it could still be improved for some classes of programs by some optimization techniques, which we had not the time to implement. This could be the object of future work.

Moreover the abstract domain keeps the same precision as the previous one for *pure* programs. For the new class of programs that can be processed, the precision is also very good: in many cases, we can derive groundness or determinacy.

Ultimately let us say, that it was very interesting to contribute to the research of Prolog abstract interpretation. We never experienced as much motivation working on the resolution of a problem in computer science.