Abstract Interpretation of Porlog
Based on OLDT Resolution

Pascal Van Hentenryck[1]
Olivier Dagimbe[2]
Baudouin Le Charlier[3]
Laurent Michel[4]

Technical Report No. CS-93-05
Febuary 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]University of Namur 21 rue Grandgagnage, B-5000 Namur (Belgium)
[3]University of Namur 21 rue Grandgagnage, B-5000 Namur(Belgium)
[4]University of Namur 21 rue Grandgagnage, B-5000 Nmur (Belgium)

Abstract Interpretation of Prolog Based on OLDT Resolution

Pascal Van Hentenryck
Brown University,
Box 1910, Providence, RI 02912 (USA)
Email: pvh@cs.brown.edu

Olivier Degimbe, Baudouin Le Charlier and Laurent Michel[1]
University of Namur,
21 rue Grandgagnage, B-5000 Namur (Belgium)
Email: ble@info.fundp.ac.be

Abstract

`OLDT`-resolution as a basis for abstract interpretation is a recurring theme in logic programming and was proposed by many authors (e.g. [6, 15, 18, 40]) because of its completeness properties for various classes of programs. In addition, an interesting property of `OLDT`-resolution for abstract interpretation is the very fine granularity it provides for the analysis. This comes from the fact that `OLDT`-resolution associates with each input pair a set of output abstract substitutions contrary to most algorithms which stores a single abstract substitution per input pair by using, say, least upper bound operation. However, despite its popularity, few theoretical and no experimental results are available on the accuracy and efficiency of `OLDT`-resolution for abstract interpretation.

This paper is a systematic study of `OLDT`-resolution for abstract interpretation from the semantic framework to an efficient fixpoint algorithm and its experimental evaluation. The semantic framework demonstrates that pure `OLDT`-resolution is not an appropriate setting for abstract interpretation and proposes an appropriate foundation using the idea of output subsumption. The algorithmic framework presents an efficient fixpoint algorithm by successive refinements. The algorithm includes optimizations such as the caching of abstract operations, output subsumption, and the use of sophisticated dependencies to avoid redundant computations. Comprehensive experimental results about the accuracy and efficiency of the algorithm are presented, using finite and infinite abstract domains. The algorithm is also compared to other analyzers, working at a coarser granularity [19, 21, 14]. Choices and variations of the algorithm are also carefully studied, including the impact of caching, dependencies, output subsumption, widening, and generalization. The results shed a new light on the usefulness of `OLDT`-resolution for abstract interpretation and the impact of granularity in abstract interpretation of logic programs.

1 Introduction

Abstract interpretation of Prolog has attracted many researchers in recent years. 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 realised in this area in terms of the frameworks

[1]This research was done while Olivier Degimbe and Laurent Michel were visiting Brown University.

(e.g. [28, 42, 25, 26, 4, 2, 8, 1]), the algorithms (e.g. [31, 2, 6, 19, 18]), the abstract domains (e.g. [30, 3, 17]) and the implementation (e.g. [41, 16, 21, 14]).

A recurring theme in abstract interpretation of Prolog is the use of `OLDT`-resolution as a starting point (e.g. [6, 15, 18, 40]). `OLDT`-resolution is often advocated as a basis for abstract interpretation because of its completeness properties for a variety of program classes. Another interesting aspect of `OLDT`-resolution as a framework for abstract interpretation is the very fine granularity it offers. In contrast to many abstract interpretation frameworks and algorithms which store a single result (e.g. an abstract substitution) per program point considered, `OLDT`-based abstract interpretation associates a set of results with each program point. In addition, with suitable optimizations such as the caching of abstract operations [14], it can be shown that the worst-case complexity of `OLDT`-based abstract interpretation is no worse than more traditional algorithms storing a single result per program point and is quadratic in the size of the abstract domain (assuming a finite abstract domain). However, worst case analysis is not always very relevant in abstract interpretation as shown in [21] and experimental results may differ significantly from the expected complexity. In addition, to the best of our knowledge, no experimental results have been reported on the efficiency of `OLDT`-based abstract interpretation.

This paper is devoted to a systematic study of abstract interpretation based on `OLDT`-resolution from the semantic framework to the fixpoint algorithm and its experimental evaluation. The semantic framework proposes three semantics which are all instances of a generic semantics: a concrete and two abstract semantics which are consistent approximations of the concrete semantics. The first abstract semantics demonstrates that pure `OLDT`-resolution is not a proper basis for abstract interpretation, since it is only applicable to finite domains and may entail much redundancy. The second semantics remedies those limitations and is shown to have the same information content than the first abstract semantics in the context of abstract interpretation, providing a more appropriate setting from a computational standpoint. A fixpoint algorithm is then presented by successive refinements, starting from a naive algorithm which is then optimized by including sophisticated dependencies to avoid redundant computations, output subsumption, and memoizing of all abstract operations. The algorithm is then analyzed systematically on a finite and an infinite abstract domain and compared with a "more traditional" algorithm [21, 14]. The experimental results cover both accuracy and efficiency as well as other measures to understand the behaviour of the algorithm. Finally, design choices and variations of the algorithms are evaluated. The impact of output subsumption, dependencies, caching, and widening techniques are quantified and variations such as generalization are evaluated.

The novel contributions of this paper are as follows:

- it presents a semantic framework for the use of `OLDT`-resolution in abstract interpretation, highlighting some fundamental properties of this approach;
- it proposes an efficient fixpoint algorithm for abstract `OLDT`-resolution, including many optimizations.
- it describes comprehensive experimental results on the algorithm covering accuracy and efficiency on finite and infinite domains and evaluating the impact of design choices and variations on the algorithm.

The rest of this paper is organized as follows. Section 2 describes the semantic framework and fixes the notation used throughout the paper. Section 3 describes the fixpoint algorithm. Section

4 describes the experimental results and Section 5 studies variations of the algorithm and justifies some of the design choices. Section 6 contains the conclusions of this research.

2 The Semantic Framework

In this section, we study the semantic framework for `OLDT`-resolution. The framework allows to dissociate semantic and algorithmic aspects of the analysis, simplifying correctness proofs and comparisons between several possible approaches. In particular, it allows the algorithms to be viewed as instances of a universal fixpoint algorithm working on very general transformations [20].

The framework includes a concrete semantics equivalent to SLD-resolution and two abstract semantics which are consistent approximations of the concrete semantics. The first abstract semantics is a simple abstraction of the concrete semantics but it has several limitations for abstract interpretation. The second abstract semantics removes these inconvenients and uses *output subsumption*, an idea also present in natural language parsing (e.g. [33]). The two abstract semantics are shown to have the same information content for abstract interpretation, indicating the strong advantage of the second semantics from a computational standpoint.

Each of the three semantics can be seen as instances of a generic semantics for Prolog for some appropriate choice of domains [23]. To simplify presentation, we first present the generic semantics and then discuss the three semantics relevant to this paper. See [23] for other semantics not directly related to `OLDT`-resolution.

The rest of this section is organized in the following way. Section 2.1 presents the preliminaries, including the abstract syntax considered in this paper. Section 2.2 introduces the generic semantics. Sections 2.3, 2.4 and 2.5 present respectively the concrete semantics and the two abstract semantics and discuss their semantic relationships.

2.1 Preliminaries

In this section, we review the main concepts used in the paper, including normalized programs and substitutions and the abstract syntax.

2.1.1 Normalized Programs and Substitutions

We assume the existence of sets F_i and P_i $(i \geq 0)$ denoting sets of functors and predicate symbols of arity i and of an infinite set PV of program variables. Variables in PV are ordered and denoted by the $x_1, x_2, \ldots, x_i, \ldots$.

Our framework uses normalized programs for simplicity. It is a straightforward task to translate any pure logic program in normalized form. Normalized programs contain clauses with heads of the form $p(x_1, \ldots, x_n)$ where $n \geq 0$ and $p \in P_n$. Normalized clauses also contain bodies of the form $g_1, \ldots, g_n$ $(n \geq 0)$ with g_i of the form $p(x_{i_1}, \ldots, x_{i_n})$ where $x_{i_1}, \ldots, x_{i_n}$ are all distinct variables and $p \in P_n$ or $x_i = x_j$ $(i \neq j)$ or $x_i = f(x_{j_1}, \ldots, x_{j_n})$ where $x_i, x_{j_1}, \ldots, x_{j_n}$ are all distinct variables and $f \in F_n$.

The motivation behind these definitions is to allow the result of any predicate p/n to be expressed as a set of substitutions on program variables $x_1, \ldots, x_n$. Normalization may induce a loss of precision for abstract interpretation when the abstract domain does not contain a pattern component.

We assume the existence of another infinite set SV of standard variables. Terms and substitutions are constructed using program and standard variables. We distinguish two kinds of substitutions: *program substitutions* (*ps* for short) whose domain and codomain are subsets of PV and SV respectively, and *standard substitutions* (*ss* for short) whose domain and codomain are subsets of SV. In the following, PS denotes the set of *ps* and RS the set of *ss*. We use $var(o)$ to represent the set of variables in the syntactical object o and $dom(\theta)$ and $codom(\theta)$ to denote the domain and codomain of a substitution. PS_D is the set of θ such that $dom(\theta) = D$. We note CS_D, the set of subsets of PS_D which are *complete*. A set of substitutions is complete if it contains all variants of any of its elements.

Definition 1 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 $x\theta = x\sigma$ for all $x \in D$.

The definition of substitution composition is slightly modified to take into account the special role held by program variables. The modification occurs when $\theta \in PS$ and $\sigma \in RS$ for which $\theta\sigma \in PS$ is defined by $dom(\theta\sigma) = dom(\theta)$ and $x(\theta\sigma) = (x\theta)\sigma$ for all $x \in dom(\theta)$. Unifiers are defined as usual but only belong to RS hereafter. We use $mgu(t_1, t_2)$ to denote the set of most general unifiers of t_1 and t_2.

2.1.2 Syntax

The abstract syntax of the language can be defined by the following grammar:

$$
\begin{array}{lll}
P & \in & Programs \\
pr & \in & Procedures \\
c & \in & Clauses \\
g & \in & Goals \\
l & \in & Literals \\
bi & \in & Built-ins \\
pc & \in & ProcedureCalls \\
f & \in & Functors \\
p & \in & ProcedureNames \\
x_i & \in & ProgramVariables
\end{array}
\qquad
\begin{array}{lll}
P & ::= & \{pr_1, \ldots, pr_n\} \\
pr & ::= & c \mid pr.c \\
c & ::= & p(x_1 \; , \; \ldots \; , \; x_n) \leftarrow g \\
g & ::= & <> \mid l.g \\
l & ::= & pc \mid bi \\
bi & ::= & x_i = x_j \mid x_{i_1} = f(x_{i_2} \; , \; \ldots \; , \; x_{i_n}) \\
pc & ::= & p(x_{i_1} \; , \; \ldots \; , \; x_{i_n})
\end{array}
$$

2.2 The Generic Fixpoint Semantics

2.2.1 Generic Domains

Let D be a finite set of program variables. For each such set, we assume the existence of two cpos, denoted S_D and SS_D, and called *set of generic substitutions on* D and *set of generic sets of substitutions on* D, respectively. S_D is not required to be a pointed cpo (no minimal element is required). SS_D is not necessarily equal to the set of subsets of S_D, but it will always be a subset of it. SS_D must have a minimal element which must be equal to $\{\}$.

2.2.2 Generic Primitive Operations

We now define the basic operations which are required by the generic semantics. There are six families of generic operations. All of them are required to be monotonic.[2]

Extension and Restriction for a Clause Informally speaking, EXTC is used to extend a substitution to introduce the body variables at clause entry and RESTRC to restrict the substitution to remove these variables. Let c be a clause. Let us denote D_c and D'_c respectively the set of variables in the head of c and the set of all variables in c. Operations $\texttt{EXTC}(c, \_)$ and $\texttt{RESTRC}(c, \_)$ have the following signatures:

$$\texttt{EXTC}(c, \_) : S_{D_c} \rightarrow SS_{D'_c},$$

$$\texttt{RESTRC}(c, \_) : SS_{D'_c} \rightarrow SS_{D_c}.$$

Extension and Restriction for a Literal Informally speaking, EXTG is used to extend a clause substitution with the result of a literal and RESTRG to project a clause substitution on the variables of a literal. Let c be a clause and l be a literal in the body of c. Let us denote D_l, the set $\{x_1, \ldots, x_n\}$ where n is the number of variables occurring in l. Operations $\texttt{EXTG}(l, \_, \_)$ and $\texttt{RESTRG}(l, \_)$ have the following signatures:

$$\texttt{EXTG}(l, \_, \_) : S_{D_c} \times SS_{D_l} \rightarrow SS_{D_c},$$

$$\texttt{RESTRG}(l, \_) : S_{D_c} \rightarrow S_{D_l}.$$

Unification Operations Informally speaking, AI_VAR is used to unify two variables and AI_FUNC to unify a variable and a functor. Let D denote the set $\{x_1, x_2\}$. Let D_f denote the set $\{x_1, \ldots, x_{n+1}\}$, where f is a functor of arity n. Operations AI_VAR and AI_FUNC have the following signatures:

$$\texttt{AI\_VAR}: S_D \rightarrow SS_D,$$

$$\texttt{AI\_FUNC}(f, \_) : S_{D_f} \rightarrow SS_{D_f}.$$

Pseudo Union Let $S_{i \in I}$ be a family of generic sets of substitutions on domain D. For any such family, we assume the existence of a generic set of substitutions on the same domain, called the pseudo union of $S_{i \in I}$, and denoted by

$$\bigsqcup_{i \in I} S_i.$$

This operation must be monotonic in the following sense: Let $S_{i \in I}$ and $S'_{i \in I'}$ be two families such that for all $i \in I$, there exists some $i' \in I'$ such that $S_i \leq S'_{i'}$. Then:

$$\bigsqcup_{i \in I} S_i \;\leq\; \bigsqcup_{i \in I'} S'_i.$$

[2]It is possible to lift this hypothesis in some cases. See [23] and [20] for details.

2.2.3 Generic Semantic domains

In the following we assume an underlying program P and we define the semantic domains and equations with respect to this underlying program. We call *(generic) underlying domain*, denoted UD, the set of pairs (s,p) such that $s \in S_p$ and p is the name of a procedure in P.

Definition 2 A *(generic) set of triples* is any set sgt of triples (s,p,S), such that $(s,p) \in UD$ and $S \in SS_{D_p}$ respecting the conditions

1. (sgt is functional and total.) For all $(s,p) \in UD$, there exists one and only one $S \in SS_{D_p}$ such that $(s,p,S) \in sgt$. We will use $sgt(s,p)$ to note S in the following.
2. (sgt is monotonic.) For all $(s,p),(s',p) \in UD$, $s \leq s'$ implies $sgt(s,p) \leq sgt(s',p)$.

We note $SSGT$ the set of *(generic) sets of triples.* An ordering can be defined on $SSGT$ by

$$sgt \leq sgt' \;\text{ iff }\; \forall (s,p) \in UD : sgt(s,p) \leq sgt'(s,p).$$

Theorem 3 $SSGT$ is a pointed cpo for the above ordering.

2.2.4 The Semantic Transformation

The transformation is depicted in Figure 1 and has the signature $TSGT : SSGT \rightarrow SSGT$. Informally speaking, the function $T_p(\beta,p,sat)$ executes all clauses defining p and takes the "union" of the results. The function T_c executes one clause by extending the input s, executing the body, and restricting the results. The function T_b executes the body of a procedure by considering each literal in turn. When the body is empty, the function simply returns the input. Otherwise, operation `RESTRG` restricts the input s to the variables of the literal. When the literal is a procedure call, a lookup in sat is performed; otherwise the operations `AI_VAR` and `AI_FUNC` are executed. The three possibilities return a set which is used recursively in function T_b after an extension.

Theorem 4 $TSGT$ is monotonic and continuous. Therefore $TSGT$ has a least fixpoint that we note $\mu(TSGT)$.

Proof Consequence of the monotonicity of the primitive operations. □

2.3 The Concrete Semantics

We now describe a fixpoint characterization of SLD-resolution and `OLDT`-resolution. More precisely, the semantics characterizes the set of success atoms which can be produced by applying SLD-resolution to the underlying program. Note that the search rule of Prolog is not taken into account, since the semantics characterizes all possible derivations.

2.3.1 The Basic Semantic Domains

The basic semantic domains, adorned with superscript 2, are as follows.

$S^1{}_D = PS_D$ and $SS^1{}_D = CS_D$.

$S^1{}_D$ is ordered pointwise and $SS^1{}_D$ is ordered by inclusion.

$TSGT(sgt) = \{(s,p,S) : (s,p) \in UD \text{ and } S = T_p(s,p,sgt)\}.$

$T_p(s,p,sgt) = \bigsqcup_{i=1}^{n} T_c(s,c_i,sgt)$
where $c_1, \ldots, c_n$ are the clauses of p.

$T_c(s,c,sgt) = \texttt{RESTRC}(c,S)$
where $S = \bigsqcup_{s_1 \in S_1} T_b(s_1,g,sgt)$,
$S_1 = \texttt{EXTC}(c,s)$,
g is the body of c.

$T_b(s,<>,sgt) = \{s\}.$

$T_b(s,l.g,sgt) = \bigsqcup_{s_1 \in S_1} T_b(s_1,g,sgt)$,
where $S_1 = \texttt{EXTG}(l,s,S_2)$,

$$
\begin{aligned}
S_2 &= sgt(s_2,p) && \textbf{if } l ::= p(\ldots), \\
&= \texttt{AI\_VAR}(s_2) && \textbf{if } l ::= x_i = x_j, \\
&= \texttt{AI\_FUNC}(f,s_2) && \textbf{if } l ::= x_i = f(\ldots),
\end{aligned}
$$

$s_2 = \texttt{RESTRG}(l,s)$.

Figure 1: The Generic Fixpoint Semantics

2.3.2 The Primitive Operations

$\texttt{EXTC}(c,\theta) = \{ \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n, x_{n+1} \leftarrow y_1, \ldots, x_m \leftarrow y_{m-n}\} : t_i = x_i\theta \ (1 \leq i \leq n),$
$y_1, \ldots, y_{m-n}$ are distinct renaming variables,
$var(t_1, \ldots, t_n) \cap \{y_1, \ldots, y_{m-n}\} = \emptyset$, and
m and n are the number of variables in c and the head of c respectively $\}$.

$\texttt{RESTRC}(c,\Theta) = \{\theta_{/D_c} : \theta \in \Theta\}.$

$\texttt{RESTRG}(l,\theta) = \{x_1 \leftarrow x_{i_1}\theta, \ldots, x_n \leftarrow x_{i_n}\theta\}.$
where $x_{i_1}, \ldots, x_{i_n}$ is the sequence of variables occurring in l.

$\texttt{EXTG}(l,\theta,\Theta) = \{ \theta\sigma : \exists\theta' \text{ such that } \theta' = \texttt{RESTRG}(l,\theta) \text{ and } \theta'\sigma \in \Theta \text{ where}$
$dom(\sigma) \subseteq codom(\theta') \text{ and } (codom(\theta) - codom(\theta')) \cap codom(\sigma) = \emptyset \}.$

$\texttt{AI\_VAR}(\theta) = \{\theta\sigma : \sigma \in mgu(x_1\theta, x_2\theta)\}$

$\texttt{AI\_FUNC}(f,\theta) = \{\theta\sigma : \sigma \in mgu(x_1\theta, f(x_2, \ldots, x_{n+1})\theta)\}$

$\bigsqcup_{i \in I} \Theta_i = \bigcup_{i \in I} \Theta_i.$

All operations are obviously monotonic with respect to the chosen orderings.

2.3.3 Equivalence with SLD-resolution

Theorem 5 Let p a procedure name of arity n, occurring in P. Let $\theta, \theta' \in PS_{D_p}$. Let $sgt^1 = \mu(TSGT^1)$. Then $\theta' \in sgt^1(\theta, p)$ iff $p(x_1, \ldots, x_n)\theta'$ is a success atom for a SLD-refutation of $p(x_1, \ldots, x_n)\theta$ with respect to P.

Proof Omitted. The proof of a very similar result is given in [29]. □

2.4 Abstract Semantics (version 1)

We now give a version of the generic semantics which is equivalent to the abstract computation model proposed by Codognet and Filé in [7] and can be used as a basis for abstract `OLDT`-resolution.

2.4.1 The Basic Semantic Domains

Let us assume that for each finite set of program variables there exists a set AS_D of so-called *abstract substitutions* over domain D, endowed with a *concretization function*:

$$Cc : AS_D \rightarrow CS_D.$$

Notice that we do not assume any particular structure over AS_D. The basic semantic domains are then defined as follows (we adorn them with superscript 3):

$S^2{}_D = AS_D$ and $SS^2{}_D = \mathcal{P}(AS_D)$.

$S^2{}_D$ is ordered pointwise and $SS^2{}_D$ is ordered by inclusion. The concretization functions for the S^2's are extended to the SS^2's as follows:

$$\forall S \in SS^2{}_D : Cc(S) \;=\; \bigcup_{\beta \in S} Cc(\beta).$$

2.4.2 The Primitive Operations

For each operation primitive of Section 2.3.2, we assume a corresponding abstract version, with the corresponding abstract signature. These operations must be *consistent*. Consistency is defined as follows:

$$\forall a \in A : o_C(Cc(a)) \;\subseteq\; Cc(o_A(a)).$$

where $o_C : C \rightarrow C'$ is a "concrete" version of an operation and $o_A : A \rightarrow A'$ is the corresponding abstract version.

In addition, when the argument is a set, we assume that the abstract operation can be defined from a basic "pointwise" version, i.e. formally

$$\forall S \in A : o_A(S) \;=\; \bigcup_{s \in S} o_A(\{s\}).$$

Those conditions are easily generalized for operations with two arguments. Note that the second condition implies that all operations are monotonic. Finally, pseudo union is once more defined as the normal union.

2.4.3 Consistency of the Abstract Semantics

Theorem 6 Let $sgt^1 = \mu(TSGT^1)$. Let $sgt^2 = \mu(TSGT^2)$. Let p a procedure name of the underlying program. Let $\theta \in PS_{D_p}$. Let $\beta \in AS_{D_p}$. Then, the following holds:

$$\theta \in Cc(\beta) \quad \Rightarrow \quad sgt^1(\theta, p) \subseteq Cc(sgt^2(\beta, p)).$$

Proof (Sketch) Consider the Kleene's sequences leading to both fixpoints. Consistency of the abstract operations implies that each element in the sequence leading to sgt^2 is a consistent approximation of the corresponding approximate in the sequence leading to sgt^1. The result follows. □

2.4.4 Discussion

It is interesting to review some of the properties of the above abstract semantics.

- No mathematical structure at all is assumed on the basic abstract domains (AS_D). The orderings required by the generic semantics are trivial ones (pointwise ordering and inclusion of sets). Monotonicity of the basic operations is also trivial. The only real requirement is consistency of the operations.
- There is no notion of approximation between abstract substitutions.
- The semantics is computable only if the basic domains are finite. Moreover, there is no way to ensure termination by resorting to approximations.

In many cases, it is more appropriate to use infinite domains to achieve more precision (see for instance [21]). This requires the use of approximations to guarantee the finiteness of the analysis. Such approximations require an upper bound operation for abstract substitutions and monotonicity of the concretization function. However, when those conditions are satisfied, the next semantics is a much more appropriate setting from a computational standpoint.

2.5 Abstract Semantics (version 2)

Assume that the sets AS_D of abstract substitutions can be endowed with a non trivial ordering to be cpos and to admit a monotonic concretization function. These properties allow to detect redundancy in sets of abstract substitutions. Assume that $\beta, \beta' \in S$ and $\beta' < \beta$. By monotonicity of Cc, $Cc(\beta') \subseteq Cc(\beta)$. Hence β subsumes β' which can be safely removed from S. It is therefore possible to define another abstract semantics using sets which are as small as possible. This property is computationally desirable and our experiments indicate that it improves efficiency dramatically. In addition, the gain in efficiency is obtained without any loss of accuracy, since the information content of both abstract semantics is the same. The price to pay for these advantages amounts to ensuring stronger monotonicity properties for the primitive operations. Experience in building practical abstract domains has shown that this is sometimes very difficult. But this drawback can be overcome by resorting to approximations. Note also that the idea of output subsumption has appeared in other contexts, in particular in natural language parsing [33].

2.5.1 The Basic Semantic Domains

We assume the same sets of abstract substitutions as before. In addition, we require that they be endowed with an ordering which reflects the inclusion of sets of concrete substitutions as closely as possible. As an immediate consequence, the concretization function must be monotonic with respect to this ordering.

Definition 7 [Non-redundant Sets of Abstract Substitutions]
let $S \subseteq AS_D$, S is said *non-redundant* if and only if for all $\beta \in S$, there exists no $\beta' \in S$ such that $\beta < \beta'$. In other words, all elements of S are maximal in S. We note $NRAS_D$, the set of subsets of AS_D which are non-redundant.

The following function maps arbitrary sets of abstract substitutions onto non-redundant sets.

Definition 8 [Set of Maximal Elements] For each set D of program variables, there is a function $\mathtt{MAXS} : \mathcal{P}(AS_D) \rightarrow NRAS_D$, such that $\mathtt{MAXS}(S)$ is the largest subset of S which is non-redundant.

Now we are in position to define the third version of the basic semantic domains:

$S^3{}_D = AS_D$ and $SS^3{}_D = NRAS_D$

The ordering for $S^3{}_D$ is the assumed ordering on AS_D. The ordering for $SS^3{}_D$ is as follows:

$$\forall S, S' \in SS^3{}_D : S \leq S' \text{ iff } (\forall \beta \in S\ \exists \beta' \in S' : \beta \leq \beta').$$

The above relation is an ordering only because sets are non-redundant. It is easy to show that $SS^3{}_D$ is a cpo for this ordering. The concretization function on $SS^3{}_D$ is defined as previously.

Note also that the above construction is essentially similar to the relational powerdomain [32] and suggests a simple way of implementing all operations as we discuss in the next section.

2.5.2 The Primitive Operations

All primitive operations are defined by taking primitive operations from the preceding framework and applying function MAXS to their result. So consistency is automatically ensured. Monotonicity with respect to the new orderings is also required. It can be proven that a sufficient condition for monotonicity of the operations is monotonicity of their pointwise versions (see Section 2.4.2). Note that, in this last instantiation,

$\bigsqcup_{i \in I} \Theta_i \ = \ \mathtt{MAXS}(\bigcup_{i \in I} \Theta_i)$.

2.5.3 Informational Equivalence of the Abstract Semantics

Theorem 9 Let $sgt^2 = \mu(TSGT^2)$. Let $sgt^3 = \mu(TSGT^3)$. Let p a procedure name of the underlying program. Let $\beta \in AS_D$. Under the assumptions, the following holds

$$Cc(sgt^2(\beta, p)) \quad = \quad Cc(sgt^3(\beta, p)).$$

Proof The result is a consequence of the fact that for any set S of abstract substitutions, $Cc(S) = Cc(\mathtt{MAXS}(S))$. $\square$

2.5.4 Discussion

It is also interesting to review some of the properties of this second abstract semantics.

- The ordering on the basic abstract domains reflects closely inclusion of sets of substitutions. Monotonicity of the basic operations is often difficult to prove.
- The ordering makes it possible to express approximations between abstract substitutions.
- The semantics is computationally applicable to larger (potentially infinite) domains because of non-redundancy. Termination of analysis can be ensured by approximating some sets of abstract substitutions. Such an operation is called a *widening* in Cousot's terminology (see [11] for a complete discussion).

3 The Algorithmic Framework

In this section, we present novel top-down fixpoint algorithms for `OLDT`-based abstract interpretation of Prolog. These algorithms can be seen as instantiations of the universal top-down fixpoint algorithm presented and proven correct in [20]. As a consequence, they share some of the spirit of our previous algorithms [19, 21, 22].

The algorithms are presented by stepwise refinements. We start by a naive algorithm `OLDT-Naive` which can be seen as an instantiation of the universal algorithm to the first abstract semantics. The inclusion of suffix dependencies gives rise to the algorithm `OLDT-Local` which avoids many redundant computation, by avoiding regenerating, for an input pattern, suffixes of clauses which would not lead to any new result. This improvement is generalized further in the algorithm `OLDT-GL` which enables different input patterns to share common suffixes. `OLDT-GL` can be seen as an instantiation of the universal algorithm for an instrumental semantics based on the first abstract semantics and containing information on all program points. The final algorithm `OLDT-GL-SU` is the extension of `OLDT-GL` with output subsumption. It can be seen as an instantiation of the universal algorithm for an instrumental semantics based on the second abstract semantics and containing information on all program points. All algorithms also include the caching optimizations proposed in [14] which amounts to memoizing the results of all abstract operations. This optimization has both theoretical and practical interest, since it reduces the worst case complexity and the execution time for many applications.

The algorithms have some commonalities with recursive query processors for deductive databases. `OLDT-Naive` is related to the unoptimized version of `QSQ` [38] and the multistage depth-first strategy for `OLDT`-resolution [35]. `OLDT-Local` performs the local optimization of the `QoSaq` framework [39] while `OLDT-GL` performs the global optimization of the same framework.[3] There are however important differences between the algorithms.

- Contrary to the `QoSaq` algorithm, our algorithm works one tuple[4] at a time instead of manipulating sets of tuples. This is possible, since abstract interpretation manipulates few abstract substitutions, and desirable, since it allows fine-grained optimizations. As a consequence,

[3] These optimizations can be viewed, in some ways, as generalizations of some techniques included in Earley's context-free grammar [13].

[4] In the context of abstract interpretation, a tuple is an abstract substitution on the clause variables.

our algorithm does not need query patterns, since a new subgoal is created for each abstract substitution.

- The elimination of redundant computations is performed at the tuple level in our algorithms, by keeping sophisticated dependencies between goals, clauses, and suffixes of clauses. No subsumption on input patterns (often called generalization in the context of `OLDT`-resolution) is used for detecting redundancies, since this may entail a loss of precision in the context of abstract interpretation (see Section 5.5 for a discussion of this aspect and experimental results).

- Some of our algorithms use output subsumption, exploiting special properties of abstract interpretation.

- Our algorithms use memoizing of all abstract operations.

- Our algorithms use widening techniques to guarantee termination in the case of infinite abstract domains. Once again, this is made possible by the nature of abstract interpretation which only requires a consistent approximation of the concrete semantics.

The design choices behind our algorithms were dictated by the nature of abstract interpretation, which rarely manipulates large collections of abstract substitutions and for which some of the techniques used, i.e. dependencies and caching, have also proved successful in previous algorithms [21, 14]. See the experimental results for an empirical validation of these choices. The algorithm can be applied as well to recursive query processing but we have not evaluated its adequacy in that context.

The rest of this section is organized as follows. Section 3.1 presents some basic data structures and procedures used in the algorithm. Section 3.2 presents algorithm `OLDT-Naive`. Section 3.3 presents algorithm `OLDT-Local` which is illustrated in Section 3.4 with a simple trace of the execution. Section 3.5 presents algorithm `OLDT-GL`. Section 3.6 presents the final algorithm integrating output subsumption into `OLDT-GL` and Section 3.7 revisits the example with this new optimization. Section 3.8 discusses the use of widening in the case of infinite abstract domains.

3.1 Preliminaries

Manipulation of the Set of Tuples Our algorithm needs two operations to extend and update the set of tuples. These operations are defined as follows:

`EXTEND(`$(\beta, p), sat$`)` = $sat \cup \{(\beta, p, \emptyset)\}$;

`ADJUST(`$(\beta, p), Set, sat$`)` =
  let $S = sat(\beta, p)$ **in**
    $\langle sat \ \setminus \ \{(\beta, p, S)\} \ \cup \ \{(\beta, p, Set \cup S)\}\ ,\ S \cup Set \neq S \rangle$.

Operation `EXTEND` simply extends the set of tuples with a new tuple while operation `ADJUST` updates an existing tuple with an additional result. Operation `ADJUST(`β, p, Set, sat`)` returns a pair $\langle sat', \mathit{modified} \rangle$ where sat' is the new set of tuples and *modified* is true iff $sat' \neq sat$.

Dependencies The algorithm also contains goal dependencies to avoid redundant computations. The goal dependencies will be generalized in the optimized versions. We only introduce the basic notions here. See [19] for more discussion.

Definition 10 A dependency graph is a set of tuples of the form $\langle(\beta,p),lt\rangle$ where lt is a set $\{(\alpha_1,q_1),\ldots,(\alpha_n,q_n)\}$ $(n \geq 0)$ such that, for each (β,p), there exists at most one lt such that $\langle(\beta,p),lt\rangle \in dp$.

We denote by $dp(\beta,p)$ the set lt such that $\langle(\beta,p),lt\rangle \in dp$ if it exists. We also denote by $dom(dp)$ the set of all (β,p) such that $\langle(\beta,p),lt\rangle \in dp$ and by $codom(dp)$ the set of all (α,q) such that there exists a tuple $\langle(\beta,p),lt\rangle \in dp$ satisfying $(\alpha,q) \in lt$.

The basic intuition here is that $dp(\beta,p)$ represents at some point the set of pairs which (β,p) depends directly upon. To be complete, we need to define the transitive closure of the dependencies.

Definition 11 Let dp be a dependency graph and assume that $(\beta,p) \in dom(dp)$.
The set $trans\_dp(\beta,p,dp)$ is the smallest subset of $codom(dp)$ closed by the following two rules:

1. if $(\alpha,q) \in dp(\beta,p)$ then $(\alpha,q) \in trans\_dp(\beta,p,dp)$;

2. if $(\alpha,q) \in dp(\beta,p)$, $(\alpha,q) \in dom(dp)$, and $(\alpha',q') \in trans\_dp(\alpha,q,dp)$ then $(\alpha',q') \in trans\_dp(\beta,p,dp)$.

Now $trans\_dp(\beta,p,dp)$ represents all the pairs which, if updated, would require reconsidering (β,p). (β,p) will not be reconsidered unless one of these pairs is updated.

In addition to the dependency graph, the algorithm makes use of a set *stable* of pairs (β,p). The intuition here is that $(\beta,p) \in stable$ iff (β,p) does not need to be reconsidered for a new iteration at some computation point.

We are now in position to specify the last three operations needed to present the algorithm:

- `REMOVE_DP(`$(\beta,p),dp,stable$`)`, where dp is a dependency graph and *stable* is a set of pairs, removes all elements $\langle(\alpha,q),lt\rangle$ from dp and all elements (α,q) from *stable* such that $(\beta,p) \in trans\_dp(\alpha,q,dp)$.

- `EXT_DP(`$(\beta,p),dp,stable$`)` inserts an element $\langle(\beta,p),\emptyset\rangle$ in dp and (β,p) in *stable*.

- `ADD_DP(`$(\beta,p),de,dp$`)` simply updates dp to include the dependency of (β,p) wrt de (after its execution $de \in dp(\beta,p)$).

3.2 The Fixpoint Algorithm `OLDT-Naive`

We are now in position to present algorithm `OLDT-Naive`. The algorithm is mainly composed of three mutually recursive procedures and is depicted in Figures 2 and 3. Procedure `solve_call` corresponds to function T_p in the abstract semantics, procedure `solve_clause` to function T_c, and procedure `solve_suffix` to function T_b.

The top-level procedure is Procedure `solve` which, given an input substitution β_{in} and a predicate symbol p, returns the final dependency graph and the set of abstract tuples *sat* containing $(\beta_{in},p,S_{out}) \in \mu(TSAT)$. Given the results, it is straightforward to compute the set of pairs (α,q)

```
procedure solve(in βin, p; out sat, dp, stable)
begin
    sat := ∅;
    dp := ∅;
    solve_call(βin, p, ∅, sat, dp, stable)
end

procedure solve_call(in βin, p, suspended; inout sat, dp, stable)
begin
    if (βin, p) ∉ (stable ∪ suspended) then
    begin
        if (βin, p) ∉ dom(sat) then
            sat := EXTEND((βin, p), sat);
        repeat
            SetOut := sat(βin, p);
            EXT_DP((βin, p), dp, stable);
            for i := 1 to m with c1, ..., cm clauses-of p do
            begin
                solve_clause(βin, p, ci, suspended ∪ {(βin, p)}, SetAux, sat, dp, stable);
                SetOut := SetOut ∪ SetAux
            end;
            (sat, modified) := ADJUST(βin, p, SetOut, sat, stable);
            if modified then REMOVE_DP((βin, p), dp, stable)
        until (βin, p) ∈ stable
    end
end

procedure solve_clause(in βin, p, c, suspended; out SetOut; inout sat, dp, stable)
begin
    βext := EXTC(c, βin);
    SetOut := ∅;
    solve_suffix(βext, βin, BODY(c), p, c, suspended, SetOut, sat, dp, stable)
end;
```

Figure 2: The Naive Algorithm: Part I

```
procedure solve_suffix(in β,β_in,body,p,c,suspended; inout SetOut,sat,dp,stable)
begin
   if body = nil then
      SetOut := SetOut ∪ { RESTRC(c,β) }
   else begin
      g := HEAD(body);
      tail := TAIL(body);
      β_aux := RESTRG(β,g);
      switch (g) of
      case X_j = X_k:
         β_int := AI_VAR(β_aux);
         β_ext := EXTG(g,β,β_int);
         solve_suffix(β_ext,β_in,tail,p,c,suspended, SetOut,sat,dp,stable)
      case X_j = f(...):
         β_int := AI_FUNC(f,β_aux)
         β_ext := EXTG(g,β,β_int);
         solve_suffix(β_ext,β_in,tail,p,c,suspended,SetOut,sat,dp,stable)
      case q(...):
         solve_call(β_aux,q,suspended,sat,dp,stable);
         ADD_DP((β_in,p),(β_aux,q),dp);
         for all β_out ∈ sat(β_aux,q) do
         begin
            β_ext := EXTG(g,β,β_out);
            solve_suffix(β_ext,β_in,tail,p,c,suspended,SetOut,sat,dp,stable)
         end
      end
   end
end
```

Figure 3: The Naive Algorithm: Part II

used by (β, p), their values in the fixpoint, as well as the abstract substitutions in any program point.

Procedure `solve_call` receives as inputs an abstract substitution β_{in}, its associated predicate symbol, a set *suspended* of pairs (α, q), *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 stable (i.e. in the set *stable*. If (β_{in}, p) is suspended, no subcomputation should be initiated. If (β_{in}, p) is stable, it means that none of the elements which it is depending upon have been updated. Otherwise, a new computation with (β_{in}, p) is initiated. This subcomputation may extend *sat* if it is the first time (β_{in}, p) is considered. The core of the procedure is a `repeat` loop which computes the set of abstract substitutions for each clause given the results of the elements of the suspended set. Local convergence is attained when (β_{in}, p) is stable. One iteration of the loop amounts to executing each of the clauses defining p and computing the union of the results. If new results are produced, the dependency graph is updated to remove all elements which depends (directly or indirectly) on (β_{in}, p). Note that the calls to the clauses are done with an extended suspended set since a subcomputation has been started with (β_{in}, p). Note also that, before executing the clauses, the dependency graph and the set *stable* have been updated to include (β_{in}, p) (which is guaranteed not to be stable before that update). (β_{in}, p) can be removed from *stable* during the execution of the loop if a pair which it is depending upon is updated.

Procedure `solve_clause` executes a single clause for an input pair and returns a set of abstract substitutions. It begins by extending the input substitution to the variables of the clause and by initializing `SetOut` to the empty set. It then calls procedure `solve_call_suffix` on the body of the clause.

Procedure `solve_suffix` handles the body of the clause. If the body is empty, the restriction of the input substitution is simply added to `SetOut`. Otherwise, `solve_suffix` considers the first goal, restricts the substitution for the variables of this goal, and performs a case analysis on the goal. If the goal is a unification, the corresponding operation is executed and procedure `solve_suffix` is called recursively with the new substitution and the rest of the body. If the goal is a procedure call, procedure `solve_call` is called recursively, a dependency is added from the current goal to the initial call, and procedure `solve_suffix` is called recursively on the extensions of all abstract substitutions contained in $sat(\beta_{aux}, q)$.

Algorithm `OLDT-Naive` can be seen as an instantiation of our universal algorithm with the transformation defined by the first abstract semantics. Its correctness is thus a consequence of the correctness of the universal algorithm.

3.3 The Fixpoint Algorithm `OLDT-Local`

Motivation Although it uses dependencies on goals, the naive algorithm contains a significant amount of redundant computation. In particular, two successive iterations of the `repeat` loop of procedure `solve_clause` entail much redundancy, since the second iteration systematically redoes all the work performed during the first iteration. Two main problems need to be solved to avoid redundant computations. On the one hand, it is necessary to identify quickly all partial computation paths which may lead to new solutions because some tuples have been added to the results of one of their goals. On the other hand, it is important to consider only new results for a procedural call whose result has been updated. In the optimized version both problems are solved using the same technique: suffix dependencies. The key idea is to consider nodes in the computation paths and to set up dependencies on the nodes. A node in a computation path is a tuple $(\beta, \langle \beta_{in}, body, c\rangle)$, where $\langle body, c\rangle$ identifies uniquely a clause suffix, β is a substitution on the variables of the clause, and β_{in} is the substitution of the input pair (β_{in}, p) that led to the node. Procedure `solve_suffix` sets up dependencies between each procedure call encountered and the nodes of the clause which lead to the procedure call. When `solve_suffix` is called, it first tests if the node to consider is stable, in which case it returns immediately, since a new execution will not produce any new result. Otherwise, it extends the dependency graph with the node and executes the procedure as in the naive algorithm, the only difference being the addition of node dependencies. If any procedure call in a clause receives new results, all nodes on the computation paths leading to the call are removed from *stable* and will be reconsidered during the next iteration.

The use of suffix dependencies automatically guides the computation to interesting computation paths since the non-relevant ones are discarded immediately. In addition, when taking the results in a set $sat(\beta, p)$, only the relevant elements are reconsidered, since those which are redundant are discovered immediately at the entry of the next call to procedure `solve_suffix` thanks to the dependencies. This comes from the fact that the node will be stable, except of course if some elements it depends upon have also been updated.

The suffix optimization also enjoys the nice property that the execution is shared when a pair (β, p) leads to the same node in a clause using different computation paths, i.e. the suffix is executed only once. This optimization is referred to as *local optimization* in [39] and is generally performed using subsumption tests. In our algorithm, *local optimization* is a consequence of the suffix dependencies. The optimization is local, since different input pairs cannot share a suffix. This comes from the inclusion of β_{in} in the node and simplifies the algorithm as no work needs to be performed when a node is stable. In Section 3.5, we show how to generalize this optimization.

Combining suffix dependencies with caching of operations [14] also guarantees that the cost of following the paths is almost negligible, as we will see in our experimental results. More advanced techniques have also been investigated to jump directly at the right place (using computation paths) but the improvement is negligible.

Formalization To formalize the algorithm, we need to generalize the dependency graph.

Definition 12 A dependency graph is a partial function: $dp : UDP \not\to \mathcal{P}(UDP)$, where UDP is the set of elements of the form (β_{in}, p) or $(\beta_{in}, \langle \beta, body, c\rangle)$, where p is a predicate symbol, c is a clause, and $body$ is a suffix of clause c.

All definitions concerning dependency graphs can now be generalized easily to accommodate the modification. The algorithm also makes use of a procedure `ADD_DP_PRE` which can be defined as follows:[5]

procedure `ADD_DP_PRE`$(\beta_{in}, p, suffixes, de, dp)$
begin
  `ADD_DP`$((\beta_{in}, p), de, dp)$;
  for all $ds \in suffixes$ **do**

[5]There are more economical ways to state the dependencies but this formulation simplifies the presentation.

ADD_DP(ds, de, dp)
end;

We are now ready to present the optimized version of the algorithm. Since the only changes are in procedure `solve_clause` and `solve_suffix`, only these procedures are depicted in Figure 4.

Procedure `solve_suffix` has one more argument, a set *suffixes*, which contains all the suffixes encountered so far in the execution of a clause for the given input abstract substitution. This set will be used to set up the dependencies and is initialized with an empty set. Procedure `solve_suffix` first tests if the suffix considered needs to be reconsidered. It returns immediately if the suffix is in the dependency graph, avoiding entire suffix or clause execution when this is the case. Otherwise, the dependency graph is extended with the considered suffix and execution proceeds as in the naive version. The only other change is procedure `ADD_DP_PRE` which replaces procedure `ADD_DP`. This new procedure uses the new set *suffixes* and sets up the dependencies not only wrt the goal (β, p) but also wrt all suffixes encountered so far. This handling of dependencies makes sure that all relevant suffixes are reconsidered when an element is added to $sat(\beta_{aux}, q)$ thanks to procedure `REMOVE_DP`.

3.4 An Example

We now show a trace of the algorithm with the suffix optimization on naive `reverse` for a simple domain, containing mode, same-value, and sharing components. The implementation is slightly different from the algorithm presented above in that it only inserts dependencies on procedure calls and not on goals dealing with unification. This allows for a reduction in memory space without inducing any significant overhead since the unification operations are cached. In the trace, we show the results of most abstract operations, mark with `*` the cached operations, and only display the mode component for clarity. Also, the parts of the trace concerning `append` are removed to improve clarity.

Figure 5 depicts the normalized version of the naive reverse program while Figures 6 and 7 depict the iterations of the algorithm. The first iteration simply produces a result for the first clause and stops after the recursive call, since no result are available for this call. The second iteration skips the first clause thanks to the dependencies. The recursive call in the second clause returns a single result and the suffix of the clause following the recursive call is executed with this result. It leads to a procedure call to `append` which produces two results. These two results are considered in turn for the last suffix (which contains only the `RESTRC` operation) and two results, one of which is new, are produced for the clause. Note that the first two unifications of the clause are cached. The third iteration is the most interesting. It is similar to the second iteration up to the recursive call which now produces two results. The first result should not be reconsidered and this is discovered by the dependencies which aborts the execution of that suffix immediately after the next sequence of built-ins (which are all cached). The second result of the recursive call is finally considered and leads to two new results, none of which are new. This concludes the analysis.

3.5 The Fixpoint Algorithm `OLDT-GL`

Motivation We mentioned previously that algorithm `OLDT-Local` does not allow the sharing of suffixes between independent calls. In this section, we indicate how these global redundant computations can be avoided. The key insight is to recognize that the suffix of a clause is essentially similar to a goal whose parameters are all the clause variables and which is defined by a clause whose

procedure `solve_clause`(**in** β_{in}, $p, c, suspended$; **out** $SetOut$; **inout** $sat, dp, stable$)
begin
  β_{ext} := EXTC(c,β_{in});
  $SetOut$:= $\emptyset$;
  solve_suffix(β_{ext},β_{in},BODY(c),p,c,$suspended$,$\emptyset$,$SetOut$,sat,dp,$stable$)
end;

procedure `solve_suffix`(**in** β,β_{in},$body, p, c, suspended, suffixes$; **inout** $SetOut, sat, dp, stable$)
begin
  if $(\beta, \langle\beta_{in}, body, c\rangle) \notin stable$ **then**
  begin
    EXT_DP(($\beta, \langle\beta_{in}, body, c\rangle$),$dp$,$stable$);
    if $body = nil$ **then**
      $SetOut$:= $SetOut \cup \{$ RESTRC(c,β) $\}$
    else begin
      g := HEAD($body$);
      $tail$:= TAIL($body$);
      $newsuffixes$:= $suffixes \cup (\beta, \langle\beta_{in}, body, c\rangle)$;
      β_{aux} := RESTRG(β,g);
      switch (g) **of**
      case X_j = X_k:
        β_{int} := AI_VAR(β_{aux});
        β_{ext} := EXTG(g,β,β_{int});
        solve_suffix(β_{ext},β_{in},$tail$,p,c,$suspended$, $newsuffixes$,$SetOut$,sat,dp,$stable$)
      case X_j = $f(\ldots)$:
        β_{int} := AI_FUNC(β_{aux},f)
        β_{ext} := EXTG(g,β,β_{int});
        solve_suffix(β_{ext},β_{in},$tail$,p,c,$suspended$, $newsuffixes$,$SetOut$,sat,dp,$stable$)
      case $q(\ldots)$:
        solve_call(β_{aux},q,$suspended$,sat,dp);
        ADD_DP_PRE(β_{in},p,$newsuffixes$,(β_{aux},q),dp);
        for all $\beta_{out} \in sat(\beta_{aux}, q)$ **do**
        begin
          β_{ext} := EXTG(g,β,β_{out});
          solve_suffix(β_{ext},β_{in},$tail$,p,c,$suspended$,$newsuffixes$,$SetOut$,sat,dp,$stable$)
        end
      end
    end
  end
end

Figure 4: The Fixpoint Algorithm `OLDT-Local`

```
reverse(X1 , X2 ) :-
   X1 = [],
   X2 = [].
reverse(X1 , X2 ) :-
   X1 = [ X3 | X4 ] ,
   reverse( X4 , X5 ),
   X7 = [],
   X6 = [ X4 | X7 ] ,
   append( X5 , X6 , X2 ).
```

Figure 5: Naive Reverse Normalized

body is the suffix considered. Hence it is natural to suspend execution on an already encountered suffix and to collect simply its result. In essence, this idea is a generalization of Chomsky normal form for context-free grammars [5]. The nodes in the new algorithm will then be tuples of the form $(\beta, \langle body, c\rangle)$ and it is necessary to associate and update results with these nodes as if they were goals.

Formalization To formalize this idea, we first need to modify the dependency graph to deal with the new kind of nodes. This modification is straightforward and is omitted here. We also need to generalize the set of abstract tuple *sat* to deal with tuples $(\beta, suffix, Set)$ in addition to the traditional (β, p, Set). The operations `EXTEND` and `ADJUST` are modified accordingly in a straightforward manner.

The new implementation of procedures `solve_clause` and `solve_suffix` are depicted in Figure 8. To understand the algorithm, it is helpful to consider the pairs (or nodes) as goals; procedure `solve_suffix` is then modelled after procedure `solve_goal`. Procedure `solve_suffix` starts by extending *sat* if necessary. If the pair is stable, the procedure simply picks up the results in *sat* and returns them together with the pair. Otherwise, procedure `solve_suffix` enters a loop until the pair is stable. The body of the loop is essentially similar to the previous version except for the dependencies and the update of *sat* to include the results of the pair. When a suffix execution returns, the procedure sets up a dependency for the current pair on the pair returned by the suffix. Also, after a call to procedure `solve_call`, the procedure sets up a dependency for the current pair on the input pair. The loop also contains, as its last two instructions, the update of *sat* for the current pair as well as the update of the dependency graph if necessary. The last two instructions of the procedure simply make sure to return the set of results for the suffix and the current pair. Note also the additional instruction in procedure `solve_clause` to set up a dependency which guarantees that (β_{in}, p) is reconsidered as soon as its first pair is not stable.

It is interesting to mention that this new version is conceptually very different from the local optimization. In the local optimization, the nodes are mainly used to guide the recomputation towards the interesting suffixes. In this version, the nodes behave like goals in that they are associated with sets of solutions. The main difference between goals and nodes is that nodes are not considered for suspension and widening.

The correctness of algorithm `OLDT-GL` can be proved by noticing that it is an instance of our universal algorithm for an instrumental semantics based on the first abstract semantics. The

```
CALL SOLVE-CLAUSE 1
   EXTC  (Gro,Var)
   UNIF-FUN  (Gro,Var)
   UNIF-FUN  (Gro,Gro)
   RESTRC  (Gro,Gro)
EXIT SOLVE-CLAUSE 1
CALL SOLVE-CLAUSE 2
   EXTC  (Gro,Var,Var,Var,Var,Var,Var)
   UNIF-FUN  (Gro,Var,Gro,Gro,Var,Var,Var)
   CALL SOLVE-CALL  reverse(Gro,Var)
   EXIT SOLVE-CALL  reverse
   RESULTS TO EXPLORE:
EXIT SOLVE-CLAUSE 2
RESULTS reverse(Gro,Var): (Gro,Gro)

CALL SOLVE-CLAUSE 1
EXIT SOLVE-CLAUSE 1
CALL SOLVE-CLAUSE 2
   *EXTC (Gro,Var,Var,Var,Var,Var,Var)
   *UNIF-FUN  (Gro,Var,Gro,Gro,Var,Var,Var)
   CALL SOLVE-CALL  reverse(Gro,Var)
   EXIT SOLVE-CALL  reverse
   RESULTS TO EXPLORE: (Gro,Gro)

   SUFFIX-0 AFTER GOAL reverse: (Gro,Var,Gro,Gro,Gro,Var,Var)
      UNIF-FUN  (Gro,Var,Gro,Gro,Gro,Var,Gro)
      UNIF-FUN  (Gro,Var,Gro,Gro,Gro,Gro,Gro)
      CALL SOLVE-CALL  append(Gro,Gro,Var)
      ...
      EXIT SOLVE-CALL  append
      RESULTS TO EXPLORE: (Gro,Gro,Gro)
                          (Gro,Gro,Any)

      SUFFIX-0 AFTER GOAL append: (Gro,Gro,Gro,Gro,Gro,Gro,Gro)
         RESTRC  (Gro,Gro)
      END-SUFFIX-0

      SUFFIX-1 AFTER GOAL append: (Gro,Any,Gro,Gro,Gro,Gro,Gro)
         RESTRC  (Gro,Any)
      END-SUFFIX-1
   END-SUFFIX-0
EXIT SOLVE-CLAUSE 2
RESULTS reverse(Gro,Var): (Gro,Gro)
                          (Gro,Any)
```

Figure 6: Naive Reverse: First Two Iterations of OLDT-Local

```
CALL SOLVE-CLAUSE 1
EXIT SOLVE-CLAUSE 1
CALL SOLVE-CLAUSE 2
   *EXTC (Gro,Var,Var,Var,Var,Var,Var)
   *UNIF-FUN (Gro,Var,Gro,Gro,Var,Var,Var)
   CALL SOLVE-CALL  reverse(Gro,Var)
   EXIT SOLVE-CALL  reverse
   RESULTS TO EXPLORE: (Gro,Gro)
                       (Gro,Any)

   SUFFIX-0 AFTER GOAL reverse: *(Gro,Var,Gro,Gro,Gro,Var,Var)
      *UNIF-FUN  (Gro,Var,Gro,Gro,Gro,Var,Gro)
      *UNIF-FUN  (Gro,Var,Gro,Gro,Gro,Gro,Gro)
   END-SUFFIX-0

   SUFFIX-1 AFTER GOAL reverse: (Gro,Var,Gro,Gro,Any,Var,Var)
      UNIF-FUN  (Gro,Var,Gro,Gro,Any,Var,Gro)
      UNIF-FUN  (Gro,Var,Gro,Gro,Any,Gro,Gro)
      CALL SOLVE-CALL  append(Any,Gro,Var)
      ...
      EXIT SOLVE-CALL  append
      RESULTS TO EXPLORE: (Gro,Gro,Gro)
                          (Any,Gro,Any)

      SUFFIX-0 AFTER GOAL append: *(Gro,Gro,Gro,Gro,Gro,Gro,Gro)
         *RESTRC (Gro,Gro)
      END-SUFFIX-0

      SUFFIX-1 AFTER GOAL append:
         RESTRC  (Gro,Any)
      END-SUFFIX-1
   END-SUFFIX-1
EXIT SOLVE-CLAUSE 2
RESULTS reverse(Gro,Var): (Gro,Gro)
                          (Gro,Any)
```

Figure 7: Naive Reverse: Third Iteration of `OLDT-Local`

instrumental semantics must simply be defined over an extended set of tuples including tuples of the form $\langle \beta, \mathit{suffix}, Set\rangle$ where *suffix* is a suffix of clause. Alternatively, the correctness can be proved by rewriting the program into an equivalent binary program (using the idea underlying Chomsky normal form) and simply using the first abstract semantics.

3.6 The Fixpoint Algorithm OLDT-GL-SU

The final algorithm integrates the idea of output subsumption in OLDT-GL. It can thus be viewed as an instance of the universal algorithm for an instrumental semantics based on the second abstract semantics. The use of output subsumption can dramatically improve the performance of the algorithm and the gain will be quantified later in the paper.

OLDT-GL-SU systematically applies this optimization by redefining operation ADJUST as

$$\begin{array}{l}\texttt{ADJUST}((\beta,p),Set,sat)\ \texttt{=}\\ \quad \textbf{let}\ S = sat(\beta,p)\ \textbf{in}\\ \qquad \langle sat\ \setminus\ \{(\beta,p,S)\}\ \cup\ \{(\beta,p,\texttt{REMOVE\_SUB}(S\cup Set))\}\ ,\ \texttt{REMOVE\_SUB}(S\cup Set)\neq S\rangle\,.\end{array}$$

where

$$\texttt{REMOVE\_SUB}(Set)\ \texttt{=}\ \{\ \beta\in Set\ \mid\ \neg\exists\gamma\in Set:\gamma>\beta\ \}$$

3.7 The Example Revisited

We now reconsider the example with the use of subsumption. Figures 9 and 10 depict the trace of the final algorithm. The first iteration is similar to the previous trace. The second iteration is similar until execution reaches the call to append. Procedure append returns one result only, since the other result (Gro,Gro,Gro) is subsumed by (Gro,Gro,Any). As a consequence, only one suffix is explored leading to the production of (Gro,Any). Now, since (Gro,Any) subsumes (Gro,Gro), only this last result is kept as a result for reverse(Gro,Var), as shown at the end of the second iteration. The third iteration does not generate any new result and the analysis is completed. There are several interesting points to notice on this example. First, the execution of the OLDT algorithm essentially mimics the execution of an algorithm collecting a single abstract substitution by taking the LUB of the clause results. The effect of subsumption is similar to the use of LUB in this example. Although this is not true in general, it is likely to be the case for many recursive algorithms for domains not containing a pattern component. Second, the use of subsumption precludes the need for the suffix optimization in this example, since there is only one abstract result per procedure.

3.8 Widening

In the case of infinite abstract domains, widening [10] is used to ensure termination. Moreover, widening techniques are used at two levels in the algorithms:

input widening: widening is used at the goal level to make sure that only finitely many input pairs are considered during the analysis;

output widening: widening is used at the output level to ensure that only finitely many outputs are stored in an output set.

```
procedure solve_clause(in β_in, p, c, suspended; out SetOut; inout sat, dp, stable)
begin
   SetOut := ∅;
   solve_suffix(EXTC(c, β_in), BODY(c), c, suspended, SetOut, sat, dp, stable, node);
   ADD_DP((β_in, p), node, dp)
end;

procedure solve_suffix(in β, body, c, suspended; inout SetOut, sat, dp, stable; out node)
begin
   if (β, ⟨body, c⟩) ∉ sat then sat := EXTEND((β, ⟨body, c⟩), sat);
   while (β, ⟨body, c⟩) ∉ stable do
   begin
      EXT_DP((β, ⟨body, c⟩), dp, stable);
      if body = nil then SetAux := { RESTRC(c, β) }
      else begin
         g := HEAD(body);
         tail := TAIL(body);
         SetAux := sat(β, ⟨body, c⟩);
         β_aux := RESTRG(β, g);
         switch (g) of
         case X_j = X_k:
            β_int := AI_VAR(β_aux);
            β_ext := EXTG(g, β, β_int);
            solve_suffix(β_ext, tail, c, suspended, SetAux, sat, dp, stable, nodeAux);
            ADD_DP((β, ⟨body, c⟩), nodeAux, dp)
         case X_j = f(...):
            β_int := AI_FUNC(β_aux, f)
            β_ext := EXTG(g, β, β_int);
            solve_suffix(β_ext, tail, c, suspended, SetAux, sat, dp, stable, nodeAux);
            ADD_DP((β, ⟨body, c⟩), nodeAux, dp)
         case q(...):
            solve_call(β_aux, q, suspended, sat, dp);
            ADD_DP((β, ⟨body, c⟩), (β_aux, q), dp);
            for all β_out ∈ sat(β_aux, q) do
            begin
               β_ext := EXTG(g, β, β_out);
               solve_suffix(β_ext, tail, c, suspended, SetAux, sat, dp, stable, nodeAux);
               ADD_DP((β, ⟨body, c⟩), nodeAux, dp)
            end
         end
      end;
      (sat, modified) := ADJUST((β, ⟨body, c⟩), SetAux);
      if modified then REMOVE_DP((β, ⟨body, c⟩), dp, stable);
   end
   SetOut := SetOut ∪ sat(β, ⟨body, c⟩);
   node := (β, ⟨body, c⟩)
end;
```

Figure 8: The Fixpoint Algorithm OLDT-GL

```
CALL SOLVE-CLAUSE 1
   EXTC  (Gro,Var)
   UNIF-FUN  (Gro,Var)
   UNIF-FUN  (Gro,Gro)
   RESTRC  (Gro,Gro)
EXIT SOLVE-CLAUSE 1
CALL SOLVE-CLAUSE 2
   EXTC  (Gro,Var,Var,Var,Var,Var,Var)
   UNIF-FUN  (Gro,Var,Gro,Gro,Var,Var,Var)
   CALL SOLVE-CALL  reverse(Gro,Var)
   EXIT SOLVE-CALL  reverse
   RESULTS TO EXPLORE:
EXIT SOLVE-CLAUSE 2
RESULTS reverse(Gro,Var): (Gro,Gro)

CALL SOLVE-CLAUSE 1
EXIT SOLVE-CLAUSE 1
CALL SOLVE-CLAUSE 2
   *EXTC (Gro,Var,Var,Var,Var,Var,Var)
   *UNIF-FUN (Gro,Var,Gro,Gro,Var,Var,Var)
   CALL SOLVE-CALL  reverse(Gro,Var)
   EXIT SOLVE-CALL  reverse
   RESULTS TO EXPLORE: (Gro,Gro)

   SUFFIX-0 AFTER GOAL reverse: (Gro,Var,Gro,Gro,Gro,Var,Var)
      UNIF-FUN  (Gro,Var,Gro,Gro,Gro,Var,Gro)
      UNIF-FUN  (Gro,Var,Gro,Gro,Gro,Gro,Gro)
      CALL SOLVE-CALL  append(Gro,Gro,Var)
      ...
      EXIT SOLVE-CALL  append
      RESULTS TO EXPLORE: (Gro,Gro,Any)

      SUFFIX-0 AFTER GOAL append: (Gro,Any,Gro,Gro,Gro,Gro,Gro)
          RESTRC  (Gro,Any)
      END-SUFFIX-0
   END-SUFFIX-0
EXIT SOLVE-CLAUSE 2
RESULTS reverse(Gro,Var): (Gro,Any)
```

Figure 9: Naive Reverse: First Two Iterations OLDT-GL-SU

```
CALL SOLVE-CLAUSE 1
EXIT SOLVE-CLAUSE 1
CALL SOLVE-CLAUSE 2
   *EXTC (Gro,Var,Var,Var,Var,Var,Var)
   *UNIF-FUN (Gro,Var,Gro,Gro,Var,Var,Var)
   CALL SOLVE-CALL  reverse(Gro,Var)
   EXIT SOLVE-CALL  reverse
   RESULTS TO EXPLORE: (Gro,Any)

   SUFFIX-0 AFTER GOAL reverse: (Gro,Var,Gro,Gro,Any,Var,Var)
      UNIF-FUN  (Gro,Var,Gro,Gro,Any,Var,Gro)
      UNIF-FUN  (Gro,Var,Gro,Gro,Any,Gro,Gro)
      CALL SOLVE-CALL  append(Any,Gro,Var)
      ...
      EXIT SOLVE-CALL  append
      RESULTS TO EXPLORE: (Any,Gro,Any)

      SUFFIX-0 AFTER GOAL reverse: (Gro,Any,Gro,Gro,Any,Gro,Gro)
          RESTRC  (Gro,Any)
      END-SUFFIX-0
   END-SUFFIX-0

EXIT SOLVE-CLAUSE 2
RESULTS reverse(Gro,Var): (Gro,Any)
```

Figure 10: Naive Reverse: Third Iteration of OLDT-GL-SU

The widening at the input level is similar to our previous algorithms and amounts to generalizing goals at the beginning of procedure `solve_goal` in case of imbricated recursive calls. See [21] for more discussion on this topic.

The widening at the output level is novel and is used when a new output, say β, is about to be inserted in an output set, say S. Instead of inserting β, the algorithm inserts

$$\beta \nabla S$$

for a given widening operator ∇.

There are a variety of possible widening operators, some of them being domain-dependent and others being domain-independent. In our experiments, we mainly use two operators, ∇_d and ∇_c. ∇_d is domain-dependent, is defined on the domain `Pattern` to be discussed later, and relates to the depth-k abstraction sometimes used in abstract interpretation. Informally speaking, ∇_d widens the new substitution by taking its lub with all the substitutions having the same outermost functors (depth-1). Since there are finitely many function symbols in a program, the output set is guaranteed to be finite. ∇_c is domain-independent and amounts to preserving at most a single substitution per clause defining the predicate. Since each predicate has only a finite number of clauses, the finiteness of the output sets is trivially guaranteed.

Most of the experimental results are performed with the widening operator ∇_d. An experimental comparison between ∇_d and ∇_c is given in Section 5.1.

4 Experimental Results

In this section, we report experimental results regarding the accuracy and efficiency of the algorithm. The experimental results were obtained on a variety of benchmarks (see Section 4.1) and on two abstract domains (see Section 4.2). Algorithm `OLDT-GL-SU` is used as the basic algorithm for `OLDT`. `OLDT-GL-SU` is also compared to a more traditional algorithm [19, 21, 14] storing tuples of the forms $(\beta_{in}, p, \beta_{out})$. This last algorithm is referred to as `AIDISO` (for Abstract Interpretation with Dynamic Input and Single Output) in the following.

4.1 The Programs Tested

The programs we use are hopefully representative of "pure" logic programs (i.e. without the use of dynamic predicates such as `assert` and `retract`). They are taken from a number of authors and used for various purposes from compiler writing to equation-solvers, combinatorial problems, and theorem-proving. Hence they should be representative of a large class of programs. In order to accommodate the many built-ins provided in Prolog implementations and not supported in our current implementation, some programs have been extended with some clauses achieving the effect of the built-ins. Examples are the predicates to achieve input/output, meta-predicates such as `setof`, `bagof`, `arg`, and `functor`. The clauses containing `assert` and `retract` have been dropped in the one program containing them (i.e. Syntax error handling in the reader program).

The program `kalah` is a program which plays the game of kalah. It is taken from [34] and implements an alpha-beta search procedure. The program `press` is an equation-solver program taken from [34] as well. We use two versions of this program, `press1` and `press2`, the difference being that `press2` has a procedure call repeated in the body of a procedure. The program `cs` is a cutting-stock program taken from [36]. It is a program used to generate a number of configurations representing various ways of cutting a wood board into small shelves. The program uses, in various ways, the nondeterminism of Prolog. We use two versions of the program; one of them (i.e. `cs1`) assumes that the data are ground while the other one (i.e. `cs`) assumes that the data are ground lists. The program `disj` is taken from [12] and is the generate and test equivalent of a constraint program used to solve a disjunctive scheduling problem. This is also a program using the nondeterminism of Prolog. Once again, we use two versions of the program with the same distinction as for the cutting stock example. The program `read` is the tokeniser and reader written by R. O'keefe and D.H.D. Warren for Prolog. It is mainly a deterministic program, with mutually recursive procedures. The program `pg` is a program written by W. Older to solve a specific mathematical problem. The program `gabriel` is the `Browse` program taken from Gabriel benchmarks. The program `plan` is a planning program taken from Sterling & Shapiro. The program `queens` is a simple program to solve the n-queens problem. `peep` is a program written by S.Debray to carry out the *peephole* optimization in the SB-Prolog compiler. It is a deterministic program. We also use the traditional concatenation and quicksort programs, say `append` (with input modes `(var,var,ground)`) and `qsort` (difference lists).

4.2 Overview of the Abstract Domains

4.2.1 The Domain `Pattern`

The abstract domain `Pattern` contains patterns (i.e. for each subterm, the main functor and a reference to its arguments are stored), sharing, same-value, and mode components. The domain is more sophisticated than the sharing domains of, for instance, [17, 30] and than the mode domains of, for instance, [41, 16]. It is best viewed as an abstraction of the domain of Bruynooghe and Janssens [3] where a pattern component has been added. The domain is fully described in [29] which contains also the proofs of monotonicity and consistency. Note also that the presentation in [29] is generic and can be instantiated to a variety of applications. The presentation here is an instantiation to modes.

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 with each subterm the following information:

- its *mode* (e.g. Gro, Var, Ngv (i.e. neither ground nor variable));
- its *pattern* which specifies the main functor as well as the subterms which are its arguments;
- its possible *sharing* with other subterms.

Note that the *pattern* is optional. If it is omitted, the pattern is said to be undefined.
In addition to the above information, each variable in the domain of the substitution is associated to one of the subterms. Note that this information enables to express that two arguments have the same value (and hence that two variables are bound together). To identify the subterms in an unambiguous way, an index is associated to each of them. If there are n subterms, we make use of indices $1, \ldots, n$. For instance, the substitution

$$\{X_1 \leftarrow t * v \ , \ X_2 \leftarrow v \ , \ X_3 \leftarrow Y_1 \setminus [\,]\}$$

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

$$\{(1, t * v), (2, t), (3, v), (4, v), (5, Y_1 \setminus [\,]), (6, Y_1), (7, [\,])\}.$$

As mentioned previously, each index is associated with a mode taken from

$$\{\perp\ ,\ Gro\ ,\ Var\ ,\ Ngv\ ,\ Novar\ ,\ Gv\ ,\ Nogro\ ,\ Any\}.$$

In the above example, we have the following associations

$$\{(1, Gro)\ ,\ (2, Gro)\ ,\ (3, Gro)\ ,\ (4, Gro)\ ,\ (5, Ngv)\ ,\ (6, Var)\ ,\ (7, Gro)\}.$$

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

$$\{(1, 2 * 3), (2, t), (3, v), (4, v), (5, 6 \setminus 7), (7, [\,])\}.$$

Finally the sharing component specifies which indices, not associated with a pattern, may possibly share variables. We only restrict our attention to indices with no pattern since the other patterns already express some sharing information and we do not want to introduce inconsistencies between the components. The actual sharing relation can be derived from these two components. In our particular example, the only sharing is the couple $(6, 6)$ which expresses that variable Y_1 shares a variable with itself.

The widening operator can be defined more precisely now.

Definition 13 Let β_1, β_2 be substitutions on domain $\{x_1, \ldots, x_n\}$. We say that β_1 and β_2 are depth-1 equivalent, denoted $\mathit{DE1}(\beta_1, \beta_2)$, if β_1 associates a functor f to x_i whenever f is associated with x_i in β_2. In addition, we denote by $\mathit{DES1}(\beta, S)$ the set

$$\{\beta' \mid \beta' \in S \text{ and } \mathit{DE1}(\beta, \beta')\}$$

for a substitution β and a set of substitutions S on the same domain.

Definition 14 [Widening operator ∇_d] Let β and S be a substitution and a set of substitutions defined on the same domain.

$$\beta\ \nabla_d\ S = \texttt{LUB}\{\beta\ \cup\ \mathit{DES1}(\beta, S)\}.$$

4.2.2 The Abstract Domain `Mode`

The domain of [29] is a reformulation of the domain of [2]. The domain could be viewed as a simplification of the elaborate domain where the pattern information has been omitted and the sharing has been simplified to an equivalence relation. Only three modes are considered: `ground`, `var and any`. Equality constraints can only hold between program variables (and not between subterms of the term bound to them). The same restriction applies to sharing constraints. Moreover algorithms for primitive operations are significantly different. They are much simpler and the loss of accuracy is significant.

4.3 Results on The Domain `Mode`

We now turn to the experimental results on the domain `Mode`.

	Possible	Actual	%Actual/Possible
append	4	2	50.00
cs	205	88	42.93
cs1	203	107	52.71
disj	165	131	79.71
disj1	163	133	81.60
gabriel	125	71	56.80
kalah	268	186	69.40
peep	543	393	72.38
pg	57	32	56.14
plan	32	14	43.75
press1	436	189	43.35
press2	455	202	44.40
qsort	12	2	16.67
queens	18	7	38.89
read	402	261	64.93
Mean			54.22
Max			81.60
Min			16.67

Table 1: Unification Specializations for the Domain `Mode`

Accuracy The modes inferred by `OLDT-GL-SU` and `AIDISO` are exactly the same, both for the input and the output modes. To try distinguishing between the two algorithms, we also measure how many unifications (i.e. executions of $x_i = x_j$ and $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$) can be specialized. We assume that $x_i = x_j$ can be specialized when one of its arguments is either ground or variable and that $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$ can be specialized when its first argument is either ground or a variable. Table 1 depicts the results for `OLDT-GL-SU`, which are also similar to the results of `AIDISO`. For this domain, the additional granularity did not bring any noticeable improvement for the two measures.

Efficiency Table 2 depicts the main efficiency results for `OLDT-GL-SU` and compare them with `AIDISO`. We give the computation times of both algorithms on a Sun SS10-30. The results indicate that, in the average, `OLDT-GL-SU` is very close to `AIDISO`, i.e. about 15% slower than `AIDISO`. This result is mainly due to the subsumption mechanism on the output which dramatically reduces the number of outputs. This is especially the case with this domain which loses much information and leads to very general substitutions.

Table 3 reports some results on the number of input pairs (β, p) considered by both algorithms. The results indicate that `OLDT-GL-SU` and `AIDISO` have essentially a similar behaviour. `OLDT-GL-SU` explores more elements on `peep`, `plan`, `qsort` and `queens` but the increase is rather small.

Table 4 reports some statistics on the size of the output sets in `OLDT-GL-SU`. The results indicate that, on the average, the size of the output sets is 1.03. The maximal size for this domain is 3 and is reached on program `read`. These last two tables explain why `OLDT-GL-SU` is so close to `AIDISO` on this domain, as the number of outputs is essentially 1. This also validates some of our hypothesis on the small number of elements manipulated by the algorithm.

	OLDT-GL-SU	AIDISO	OLDT-GL-SU/AIDISO
cs	1.51	1.50	1.01
cs1	1.27	1.23	1.03
disj	0.77	0.77	1.00
disj1	0.81	0.80	1.01
gabriel	0.35	0.34	1.03
kalah	1.20	1.18	1.02
peep	1.97	1.15	1.71
pg	0.15	1.14	1.07
plan	0.16	0.11	1.45
press1	1.61	1.50	1.07
press2	1.62	1.52	1.07
qsort	0.14	0.07	2.00
queens	0.06	0.04	1.50
read	1.80	1.37	1.31
Mean	0.96	0.84	1.15
Max	1.97	1.52	2.00

Table 2: Efficiency Results on the Domain `Mode`

	OLDT-GL-SU		AIDISO	
	max	mean	max	mean
append	2	2.00	2	2.00
cs	4	1.65	4	1.65
cs1	3	1.36	3	1.36
disj	2	1.30	2	1.30
disj1	3	1.31	3	1.31
gabriel	10	2.33	10	2.33
kalah	6	2.18	6	2.18
peep	11	2.95	9	2.42
pg	4	2.00	4	2.00
plan	7	2.79	7	2.64
press1	8	2.64	8	2.64
press2	8	2.64	8	2.64
qsort	12	6.00	8	4.67
queens	7	3.20	7	2.80
read	4	1.68	4	1.68
Mean		2.40		2.24
Max	12	6.00	10	4.67

Table 3: Statistics on the Input Pairs for the Domain `Mode`

	min	max	mean
append	1	1	1.00
cs	1	2	1.02
cs1	1	1	1.00
disj	1	2	1.03
disj1	1	2	1.03
gabriel	1	1	1.00
kalah	1	2	1.01
peep	1	2	1.07
pg	1	1	1.00
plan	1	2	1.05
press1	1	1	1.00
press2	1	1	1.00
qsort	1	2	1.06
queens	1	2	1.06
read	1	3	1.14
Min			1.00
Mean			1.03
Max	1	3	1.14

Table 4: Statistics on the Sizes of the Output Sets for Domain `Mode`

4.4 Results on The Domain `Pattern`

We now turn to the experimental results for the domain `Pattern`.

4.4.1 Accuracy

Tables 5 and 6 compare respectively the input and output modes of `OLDT-GL-SU` and `AIDISO`. In both tables, the first column gives the number of arguments, the second column the number of arguments whose mode is inferred more precisely by `OLDT-GL-SU`, and the last column is the same measure in percentage. The input results indicate that `OLDT-GL-SU` brings some improvement over `AIDISO` on four programs, although the differences are not very important. Some of the improvements are significant for program analysis as on program `peep` for which `OLDT-GL-SU` infers three times *Var* instead of *Any* for `AIDISO`. Other are less important (e.g. inferring *Gv* instead of *Any*). The most important improvement is on `plan`. The output results indicate that `OLDT-GL-SU` brings some small improvements for three programs.

Table 7 reports the specialization results for `OLDT-GL-SU`; those results are almost similar to those of `AIDISO`. `OLDT-GL-SU` produces a better result on `peep`, where it handles a list in two cases: an empty list and a non-empty list. The finer granularity does not bring much improvement on the domain for the benchmarks considered but it shows that `OLDT-GL-SU` can be more precise than `AIDISO` on practical programs.

	Arguments	MorePrecise	%MorePrecise
cs	94	0	0.00
cs1	93	0	0.00
disj	60	0	0.00
disj1	59	0	0.00
gabriel	60	2	3.33
kalah	124	0	0.00
peep	63	3	4.76
pg	30	0	0.00
plan	33	5	15.15
press1	144	1	0.69
press2	144	0	0.00
qsort	9	0	0.00
queens	11	0	0.00
read	123	0	0.00

Table 5: Comparison of Input Modes for the Domain `Pattern`

	Arguments	MorePrecise	%MorePrecise
cs	94	0	0.00
cs1	93	0	0.00
disj	60	0	0.00
disj1	59	0	0.00
gabriel	60	1	1.66
kalah	124	0	0.00
peep	63	4	6.34
pg	30	0	0.00
plan	33	0	0.00
press1	144	0	0.00
press2	144	0	0.00
qsort	9	0	0.00
queens	11	0	0.00
read	123	1	0.81

Table 6: Comparison of Output Modes for the Domain `Pattern`

	Possible	OLDT-GL-SU	OLDT-GL-SU/Possible	AIDISO	AIDISO/Possible
append	4	4	100.00	4	100.00
cs	205	205	100.00	205	100.00
cs1	203	203	100.00	203	100.00
disj	165	164	99.39	164	99.39
disj1	163	162	99.39	162	99.39
gabriel	114	72	63.16	72	63.16
kalah	268	263	98.13	263	98.13
peep	543	538	99.07	527	97.05
pg	56	51	91.07	51	91.07
plan	32	29	90.63	29	90.63
press1	434	258	59.45	258	59.45
press2	435	420	96.55	420	96.55
qsort	12	11	91.67	11	91.67
queens	18	18	100.00	18	100.00
read	405	274	67.65	274	67.65
Mean			90.36		89.58
Max			100.00		100.00
Min			59.45		59.45

Table 7: Unification Specializations for the Domain `Pattern`

4.4.2 Efficiency

Table 8 reports the efficiency results of `OLDT-GL-SU` and `AIDISO` on the domain `pattern`. They indicate that `OLDT-GL-SU` is about 17 times slower than `AIDISO` in the average. For some large programs (e.g. program `disj_b1`), `OLDT-GL-SU` is about 45 times slower than `AIDISO`. The analysis time is in the worst case (i.e. program `press1`) about 126 seconds. As can be seen easily, the performance of `OLDT-GL-SU` on this domain is not as good as on domain `Mode`. The main reason is the fact that domain `Pattern` is both more precise and richer than domain `Mode`. In particular, the pattern component forces `OLDT-GL-SU` to maintain outputs whose modes are similar but which have different functors as is typical in analyzing recursive programs. This can lead to additional precision as we have seen but it also entails some duplicated effort as indicated by the efficiency results.

Table 9 reports the experimental results on the number of input pairs (β_{in}, p) for both algorithms. The maximum number of input pairs in `OLDT-GL-SU` is 104 (on program `press1`), while it is 31 for `AIDISO` (on program `read`). The average number of input pairs is about 3.5 for `OLDT-GL-SU`, while it is about 2 for `AIDISO`. The results indicate that `OLDT-GL-SU` explores more elements compared to `AIDISO`, although the increase is reasonable.

Table 10 reports the experimental results on the size of the output sets for `OLDT-GL-SU`. The results indicate that, on the average, the size of the output sets is about 3. This number is rather low and is mainly due to the power of subsumption on the outputs. The maximal size for this domain is 108 and is reached on program `peep`.

Table 11 reports the number of abstract operations executed in both algorithms. The ratios indicate that `OLDT-GL-SU` performs about 22, 11, 17, and 45 times more executions of operations

	OLDT-GL-SU	AIDISO	OLDT-GL-SU/AIDISO
cs	15.42	2.15	7.17
cs1	15.00	2.16	6.94
disj	54.70	1.21	45.21
disj1	53.85	1.18	54.64
gabriel	3.00	0.76	3.95
kalah	9.34	1.98	4.72
peep	70.12	2.42	28.98
pg	0.61	0.29	2.10
plan	0.48	0.24	2.00
press1	126.31	9.42	13.41
press2	31.14	3.00	10.38
qsort	0.56	0.01	56.00
queens	0.12	0.01	12.00
read	38.27	5.58	6.86
Mean	29.92	2.17	17.52
Max	126.31	9.42	56.00

Table 8: Efficiency Results on the Domain Pattern

	OLDT-GL-SU		AIDISO	
	max	mean	max	mean
cs	19	3.00	8	1.38
cs1	19	2.73	8	1.39
disj	6	1.93	3	1.23
disj1	6	1.90	3	1.24
gabriel	7	2.76	7	2.24
kalah	9	3.38	7	1.67
peep	13	3.42	13	3.00
pg	5	2.50	4	2.20
plan	5	2.00	5	1.86
press1	104	12.00	31	6.32
press2	28	3.45	13	2.40
qsort	4	2.00	4	2.00
queens	3	1.80	2	1.40
read	92	5.84	32	3.16
Mean	22.86	3.48	9.40	2.25
Max	104	12.00	3	6.32

Table 9: Statistics on the Input Pairs for the Domain Pattern

	min	max	mean
cs	1	4	1.28
cs1	1	4	1.27
disj	1	3	1.48
disj1	1	3	1.47
gabriel	1	4	2.05
kalah	1	6	1.58
peep	1	108	18.82
pg	1	3	1.52
plan	1	2	1.32
press1	1	13	1.99
press2	1	13	2.06
qsort	2	4	2.33
queens	1	2	1.78
read	1	30	2.81
Min	1	2	1.27
Mean			2.98
Max	2	108	18.82

Table 10: Statistics on the Sizes of the Output Sets for Domain `Pattern`

`AI_IS`, `EXTG`, `SMALLER`, and `COMPARE`. The high figure on `COMPARE` is due to the use of subsumption. Recall also that the more costly operations are `EXTG`, `AI_FUNC`, and `LUB` for `AIDISO`. The results on the number of operations correspond well to the efficiency results.

Finally, we report in Tables 12 and 13 some results on the time spent in the abstract operations. As a consequence, it also gives an idea of the time spent in the control part of the algorithm, i.e. the dependency graph, the handling of abstract tuples, the suspended set, and the various loops in the procedures. Several of these costs are in fact not compressible. The results in Table 12 establish that our algorithm is an efficient implementation of the abstract semantics and spends, in the average, about 90 to 94% of its time in the abstract operations depending upon the inclusion of the caching overhead in the time. `time` is the time purely spent in the operations while `time_c` includes the overhead of hashing and copying the substitutions. This indicates that little room is left for improvement in our algorithm, since the control time is about 10% in the average.

Table 13 indicates how the time is distributed between the various operations. Once again, it appears that `EXTG` is the most costly operation, followed by `AI_FUNC` and `LUB`. This was also the case in `AIDISO`.

4.4.3 Memory Consumption

Table 14 reports statistics on the memory consumption of the algorithm. To understand the measures, it is worthwhile mentioning that our algorithm uses a garbage collector which recovers (some of) the caching tables when memory consumption becomes too high. All the experimental results reported so far have been gathered with a version of the algorithm where the garbage collection is activated when memory requirements reach 10 megabytes. Only `press1` requires the use of the garbage collector. The table reports in column `total` the total memory consumption

	OLDT-GL-SU	AIDISO	OLDT-GL-SU/AIDISO
AI_VAR	1564	496	3.15
AI_FUNC	23456	7316	3.21
AI_IS	7038	321	21.93
AI_TEST	1374	376	3.65
RESTRG	10145	2536	4.00
EXTG	37863	3478	10.89
RESTRC	18964	2657	7.14
EXTC	5165	3074	1.68
LUB	16682	4083	4.09
EXTEND	1729	1025	1.69
ADJUST	32648	1199	27.23
COMPARE	54745	1213	45.13
SMALLER	133743	7775	17.20
Mean			11.61

Table 11: Statistics on the Abstract Operations for Domain Pattern

	time	time_c
cs	96.05	98.50
cs1	96.73	98.91
disj	98.17	99.16
disj1	98.75	99.17
gabriel	88.54	93.70
kalah	92.75	96.37
peep	84.08	90.17
pg	89.09	93.88
plan	85.41	89.61
press1	85.33	91.99
press2	90.17	94.96
qsort	85.09	91.35
queens	82.28	92.12
read	88.01	93.49
Mean	90.03	94.53

Table 12: Time Spent in the Abstract Operations for Domain Pattern

	time	time_c
EXTG	42.82	43.43
EXTC	0.90	1.08
RESTRG	1.15	1.53
RESTRC	2.36	2.75
AI_VAR	1.59	1.62
AI_FUNC	31.89	32.50
AI_IS	2.96	3.13
AI_TEST	0.73	0.84
LUB	5.24	6.27
SMALLER	0.39	1.38

Table 13: Time Repartition in the Abstract Operations for Domain `Pattern`

	total	min
cs	2,642,902	1,023,667
cs1	2,575,794	991,986
disj	8,059,318	2,877,008
disj1	8,053,103	2,872,606
gabriel	580,709	273,408
kalah	1,644,798	781,400
peep	9,717,050	2,594,221
pg	194,963	127,696
plan	146,992	111,570
press1	8,657,704*	4,939,254
press2	5,252,580	1,573,642
qsort	111,485	74,096
queens	69,976	56,117
read	5,938,061	2,364,052

Table 14: Memory Consumption for Domain `Pattern`

	A:OLDT-GL-SU(∇_c)	B:AIDISO	A/B	C:OLDT-GL-SU	A/C
cs	15.79	2.15	7.34	15.42	1.15
cs1	15.49	2.16	7.17	15.00	1.03
disj	56.96	1.21	47.07	54.70	1.04
disj1	55.80	1.18	47.29	53.85	1.04
gabriel	2.95	0.76	3.88	3.00	0.98
kalah	9.33	1.98	4.71	9.34	1.00
peep	6.68	2.42	2.76	70.12	0.10
pg	0.94	0.29	3.24	0.61	1.54
plan	0.57	0.24	2.38	0.48	1.19
press1	169.26	9.42	17.97	126.31	1.34
press2	24.63	3.00	8.21	31.14	0.79
qsort	0.39	0.01	39.00	0.56	0.70
queens	0.11	0.01	11.00	0.12	0.92
read	20.00	5.58	3.58	38.27	0.52
Mean	27.06	2.17	14.75	29.92	0.95
Max	169.26	9.42	47.29	126.31	1.54

Table 15: Efficiency of Results of OLDT-GL-SU(∇_c) on Domain Pattern

in use at the end of the computation and in column min the amount of memory strictly needed for the program to execute the benchmark. As can be seen, the largest programs require about 3 Mbytes of memory with a peek at about 5 Mbytes on press1. Half of the programs require more than 1 Mbyte. The memory consumption is significantly higher when the caches are never garbage collected as is the case of all programs but press1. Several programs reach peaks of 8 to 9 megabytes. Of course, press1 requires more than 10 megabytes, since it uses the garbage-collector.

5 Choices and Variations for the Algorithm

In this section, we review the importance of some of the design choices in our algorithm and consider several possible variations. Section 5.1 discusses the impact of various widening techniques and Section 5.2 studies the importance of output subsumption. Sections 5.3 and 5.4 consider respectively the impact of caching and of the global optimization. Section 5.5 considers an algorithm using generalization, as is typical in OLDT-resolution. Section 5.6 considers a coarser granularity and studies its impact.

5.1 The Impact of Widening

In this section, we study the impact of the widening and consider OLDT-GL-SU with the widening operator ∇_c. This algorithm is referred to as OLDT-GL-SU(∇_c) in the following. The accuracy of both widening techniques is exactly the same on the domain and benchmarks considered. The efficiency results of OLDT-GL-SU(∇_c) are given in Table 15. Most of the results are consistent with those of OLDT-GL-SU and, in general, OLDT-GL-SU(∇_c) is slightly faster than OLDT-GL-SU. In the average, it is about 14 times slower than AIDISO. The main difference occurs on program peep, where OLDT-GL-SU(∇_c) is much faster than OLDT-GL-SU (6.68 sec. versus 70.12 sec.). The difference comes from the fact that this program has a predicate giving about 100 different answers

	OLDT-GL	AIDISO	OLDT-GL/OLDT-GL-SU
cs	18.92	1.51	12.53
cs1	11.23	1.27	8.84
disj	2.00	0.77	2.60
disj1	1.92	0.81	2.37
gabriel	1.80	0.35	5.14
kalah	1.83	1.20	1.53
peep	9.00	1.97	4.57
pg	0.34	0.15	2.27
plan	0.22	0.16	1.38
press1	8.33	1.61	5.17
press2	8.74	1.62	5.40
qsort	0.43	0.14	3.07
queens	0.01	0.06	0.17
read	83.00	1.80	46.11
Mean	10.56	0.96	7.22
Max	83.00	1.97	46.11

Table 16: Efficiency Results on the Domain Mode without Subsumption

in both algorithms; however, most of these answers are clustered by the widening operator ∇_c at the next level up which has only three clauses, while the operator ∇_d propagates them upwards.

5.2 The Impact of Output Subsumption

We now report some experimental results to indicate the importance of subsumption on the results. As mentioned previously, subsumption on results does not decrease accuracy but may potentially remove many redundant computations. The purpose of this section is to quantify this gain. The algorithm without subsumption on the outputs is denoted OLDT-GL.

5.2.1 The Domain Mode

Table 16 depicts the main efficiency results for OLDT-GL and compare them with OLDT-GL-SU. The results indicate that, in the average, OLDT-GL is about 7 times slower than AIDISO, proving the importance of subsumption. OLDT-GL is more than 46 times slower on read. In general, the difference between OLDT-GL and OLDT-GL-SU is more significant for the large programs.

Table 17 reports some results on the number of input pairs (β, p) considered by both algorithms. The results indicate that OLDT-GL has an average of about 3.6 input pairs per predicate while OLDT-GL-SU has an average of 2.4. The maximum number for OLDT-GL is 26 while it is 10 for OLDT-GL-SU.

Table 18 reports some statistics on the size of the output sets in OLDT-GL. The results indicate that, on the average, the size of the output sets is about 2.7. The maximal size for this domain is 45 and is reached on program read. The last two tables indicate clearly the benefit of subsumption for OLDT-GL-SU on domain Mode. Subsumption reduces the size of the outputs by almost a factor 3 in addition to eliminating many input calls.

	OLDT-GL		OLDT-GL-SU	
	max	mean	max	mean
append	2	2.00	2	2.00
cs	25	4.79	4	1.65
cs1	24	2.94	3	1.36
disj	5	2.20	2	1.30
disj1	5	2.10	3	1.31
gabriel	11	4.24	10	2.33
kalah	8	2.47	6	2.18
peep	26	5.26	11	2.95
pg	6	2.30	4	2.00
plan	7	3.36	7	2.79
press1	13	4.72	8	2.64
press2	13	4.83	8	2.64
qsort	12	6.00	12	6.00
queens	8	3.60	7	3.20
read	24	3.50	4	1.68
Mean		3.62		2.40
Max	26	6.00	12	6.00

Table 17: Statistics on the Input Pairs for the Domain `Mode` without Subsumption

	min	max	mean
append	2	2	2.00
cs	1	7	2.11
cs1	1	7	2.55
disj	1	4	1.59
disj1	1	4	1.56
gabriel	1	10	2.44
kalah	1	4	1.71
peep	1	19	4.40
pg	1	5	2.17
plan	1	6	1.87
press1	1	16	3.18
press2	1	16	3.14
qsort	2	4	2.78
queens	1	2	1.56
read	1	45	7.23
Min	1	2	1.56
Mean	1.13	10.07	2.69
Max	2	45	7.23

Table 18: Statistics on the Sizes of the Output Sets for Domain `Mode`

	OLDT-GL	OLDT-GL-SU	OLDT-GL/OLDT-GL-SU
cs	300.00*	15.42	19.46
cs1	300.00*	15.00	20.00
disj	300.00*	54.70	5.48
disj1	300.00*	53.85	5.57
gabriel	126.57	3.00	42.19
kalah	300.00*	9.34	32.12
peep	300.00*	70.12	4.28
pg	10.50	0.61	17.21
plan	9.22	0.48	200.96
press1	300.00*	126.31	2.38
press2	300.00*	31.14	9.63
qsort	6.40	0.56	11.43
queens	0.16	0.12	1.33
read	300.00*	38.27	7.84
Mean	210.01	29.92	27.13
Max	300.00	126.31	1.13

Table 19: Efficiency Results on the Domain `Pattern` without Subsumption

5.2.2 The Domain `Pattern`

We now turn to the importance of subsumption for domain `Pattern`. Table 19 reports the efficiency results of `OLDT-GL` and `OLDT-GL-SU` on the domain `pattern`. A time limit of 5 minutes was imposed on `OLDT-GL`. The results indicate that subsumption is of primary importance to obtain results on domain `Pattern`. Most large programs cannot terminate in the given time limit without subsumption. On smaller programs, the results indicate substantial differences between `OLDT-GL-SU` and `OLDT-GL`. In particular, `gabriel` is about 42 times slower and `plan` is about 200 times slower. The loss in efficiency without the use of subsumption is more dramatic here, since the domain is richer and more accurate. Note also that most programs would take an unacceptable computation time in `OLDT-GL` if sophisticated dependencies are not used.

5.3 The Impact of Caching

In this section, we assess the importance of caching on the efficiency of the algorithm for the domain `pattern`. The results are depicted in Table 20. They indicate that the caching brings an improvement of about 36% in the average. The best speedup is on program `qsort` which has an improvement of 53%. This improvement is roughly comparable to the speedup of caching on `AIDISO`, which was about 30%.

5.4 The Impact of Dependencies

In this section, we assess the importance of the global optimization on the efficiency of the algorithm for the domain `Pattern`. The results are depicted in Table 21. They indicate that the global optimization brings an improvement of 10% in the average. The best speedup is on program `press1` which has an improvement of 40%. The main reason for the small improvement is due to

	OLDT-NC-GL-SU	OLDT-GL-SU	OLDT-NC-GL-SU/OLDT-GL-SU
cs	22.55	15.42	0.68
cs1	21.79	15.00	0.69
disj	60.41	54.70	0.91
disj1	60.15	53.85	0.90
gabriel	5.68	3.00	0.53
kalah	17.12	9.34	0.55
peep	128.60	70.12	0.55
pg	0.83	0.61	0.73
plan	0.84	0.48	0.57
press1	223.23	126.31	0.57
press2	45.00	31.14	0.69
qsort	1.19	0.56	0.47
queens	0.23	0.12	0.52
read	59.65	38.27	0.64
Mean	46.23	29.92	0.64
Max	223.23	126.31	0.47

Table 20: The Impact of Caching on Domain Pattern

the fact that most of the benefits of this optimization are subsumed by the caching optimization which makes sure that any reexecution never performs any abstract operation.

We also assess the benefits of the optimization in the absence of caching. The results depicted in Table 22 indicate that the improvement is more important in this case, confirming that caching reduces the need for sophisticated dependencies in this domain. Note that the improvement can be even more substantial in domains where the control time of the algorithm may be predominant. In the average, the dependencies produce a 22% gain. On some programs, the gain is more than 50%.

The comparison between the local and global optimizations was also performed. Interestingly, little difference (about 1-2%) shows up between the two optimizations.

5.5 The Impact of Generalization

In OLDT-resolution, it is common, when considering a procedure call G, to determine if G is an instance of a previously encountered procedure call A. If this is the case, the results of A are or will be used to solve G. This technique, sometimes called *generalization*, is complete in the case of OLDT-resolution and may improve efficiency. However, in the context of abstract interpretation, generalization may lead to a loss of precision, since abstract unifications of the procedure call substitution and the result abstract substitutions are not always as precise as the concrete unifications. In addition, the loss of precision may induce a loss in efficiency, since the algorithm may converge more slowly. Note that generalization is somewhat related to widening, although there are important differences. On the one hand, widening only applies to suspended procedure calls, while generalization can apply with respect to any procedure call encountered previously. On the other hand, generalization does not replace widening, since incomparable procedure calls need to be dealt with.

The purpose of this section is to investigate experimentally the use of generalization and its

	OLDT-SU	OLDT-GL-SU	OLDT-GL-SU/OLDT-SU
cs	16.47	15.42	0.94
cs1	16.18	15.00	0.93
disj	55.75	54.70	0.98
disj1	56.95	53.85	0.95
gabriel	3.32	3.00	0.90
kalah	10.80	9.34	0.86
peep	80.95	70.12	0.87
pg	0.71	0.61	0.86
plan	0.49	0.48	0.98
press1	208.86	126.31	0.60
press2	33.93	31.14	0.92
qsort	0.58	0.56	0.97
queens	0.14	0.12	0.86
read	40.17	38.27	0.95
Mean	46.23	29.92	0.90
Min	223.23	126.31	0.60

Table 21: The Impact of Dependencies on Domain Pattern with Caching

	OLDT-NC-SU	OLDT-NC-GL-SU	OLDT-NC-GL-SU/OLDT-NC-SU
cs	33.28	22.55	0.68
cs1	32.22	21.79	0.68
disj	64.64	60.41	0.93
disj1	65.41	60.15	0.92
gabriel	7.00	5.68	0.81
kalah	33.00	17.12	0.52
peep	279.76	128.60	0.46
pg	1.33	0.83	0.62
plan	0.83	0.84	1.01
press1	396.92	223.23	0.56
press2	54.51	45.00	0.83
qsort	1.00	1.19	1.19
queens	0.25	0.23	0.92
read	72.38	59.65	0.82
Mean	74.47	46.23	0.78
Min	396.92	223.23	0.46

Table 22: The Impact of Dependencies on domain Pattern without Caching

	Possible	Actual	Actual/Possible
cs	205	205	100.00
cs1	203	203	100.00
disj	165	164	99.39
disj1	163	162	99.39
gabriel	114	72	63.16
kalah	268	263	98.13
peep	543	538	99.07
pg	56	51	91.07
plan	32	29	90.63
press1	434	258	59.44
press2	435	259	59.54
qsort	12	11	91.67
queens	18	18	100.00
read	405	274	67.65
Mean			87.08
Max			100.00
Min			59.44

Table 23: Unification Specializations with Generalization for the Domain `Pattern`

impact on accuracy and efficiency. Generalization was integrated in the algorithm by introducing new instructions at the beginning of Procedure `solve_call`. The first time a goal (i.e. a pair (β, p)) is encountered, the instructions check if there exists a generalization of (β, p) in sat, in which case the procedure continues with the generalization. Otherwise, the call proceeds as usual. The algorithm also makes sure to assign the same procedure call to the subsequent occurrences of (β, p). The algorithm including the generalization is denoted `OLDT-GL-SU-GE`.

Table 23 depicts the specialization results of `OLDT-GL-SU-GE`. As can be seen, `OLDT-GL-SU-GE` loses precision compared to `OLDT-GL-SU` on program `press2`. This comes from the use of mutually recursive and divide and conquer procedures which often call the same predicate with different call patterns. All other results are similar.

Table 24 reports the efficiency results for `OLDT-GL-SU-GE`. The use of generalization does not really pay off in the context of abstract abstraction, since, in the average, `OLDT-GL-SU-GE`is 2% faster than `OLDT-GL-SU`. It is much faster on programs `press1` but much slower on program `press2` for which it loses precision.

The above results seem to indicate that the use of generalization is not as attractive in the context of abstract interpretation as it is for `OLDT`-resolution.

5.6 The Impact of Granularity

In this section, we report a variation on the granularity of the algorithm. We consider an `OLDT`-based algorithm which stores a single tuple $\langle \beta_{in}, p, S \rangle$ for each procedure p. The algorithm, referred to as `OLDT-GL-SU-SI`, can be obtained from the previous global optimization algorithm through a number of modifications, including the addition of a `LUB` operation at the beginning of `solve_call`, a refinement of the initial test and of the loop test in the same procedure, and a simplification of the

	OLDT-GL-SU-GE	OLDT-GL-SU	OLDT-GL-SU-GE/OLDT-GL-SU
cs	16.00	15.42	1.04
cs1	15.56	15.00	1.04
disj	56.99	54.70	1.04
disj1	57.33	53.85	1.06
gabriel	2.88	3.00	0.96
kalah	6.00	9.34	0.64
peep	61.41	70.12	0.88
pg	0.60	0.61	0.98
plan	0.46	0.48	0.96
press1	71.90	126.31	0.57
press2	76.30	31.14	2.45
qsort	0.47	0.56	0.84
queens	0.11	0.12	0.92
read	12.56	38.27	0.33
Mean	27.04	29.92	0.98
Min	0.11	0.12	0.33
Max	76.30	126.31	2.45

Table 24: Efficiency Results on the Domain `Pattern` with Generalization

dependencies. There is also no need to store the results for each node, since there is a single input pair but it is necessary for correctness to keep the more sophisticated handling of dependencies of the global optimization. The detail of these changes is outside the scope of this paper but are described in [37]. As a consequence, the granularity of the algorithm becomes coarser, since many input calls are now collapsed together, and a loss of accuracy may occur.

The accuracy results of `OLDT-GL-SU-SI` are reported in Table 25, which reports the unification specializations for the domain `pattern`. The results are essentially similar to those of `OLDT-GL-SU`, the only difference being on `press2`. On this program, `OLDT-GL-SU-SI` loses much accuracy, since its percentage of specializations now drops to about 58% instead of about 97% for `OLDT-GL-SU`. Interestingly, the results are similar to those obtained using generalization, which can be explained by the fact that this algorithm achieves a form of generalization.

The efficiency results of `OLDT-GL-SU-SI` are depicted in Table 26. `OLDT-GL-SU-SI` is, in the average, 1.8 slower than `OLDT-GL-SU`. `OLDT-GL-SU-SI` is faster on several programs, including `peep` where the gain is significant. It is also slower in many programs, including most of the largest programs. It is about 10 times slower on `press2`. The coarser granularity is not worthwhile for this domain, since it may lose precision and does not result in a better efficiency behaviour.

6 Conclusion

`OLDT`-resolution has been proposed by many authors as a basis for abstract interpretation of logic programs, although, to the best of our knowledge, no experimental results were reported so far. One of the main interests of `OLDT`-resolution is the very fine granularity it offers since sets of abstract substitutions are stored as results.

This paper has presented a systematic study of `OLDT`-resolution for abstract interpretation from

	Possible	Actual	Actual/Possible
cs	205	205	100.00
cs1	203	203	100.00
disj	165	164	99.39
disj1	163	162	99.39
gabriel	114	72	63.16
kalah	268	263	98.13
peep	543	526	96.87
pg	56	51	91.07
plan	32	29	90.63
press1	434	258	59.44
press2	435	259	58.57
qsort	12	11	91.67
queens	18	18	100.00
read	405	274	67.65
Mean			87.08
Max			100.00
Min			58.03

Table 25: Unification Specializations for OLDT-GL-SU-SI on Domain Pattern

	OLDT-GL-SU-SI	OLDT-GL-SU	OLDT-GL-SU-SI/OLDT-GL-SU
cs	17.84	15.42	1.16
cs1	17.21	15.00	1.15
disj	66.62	54.70	1.22
disj1	66.75	53.85	1.24
gabriel	2.94	3.00	0.98
kalah	10.82	9.34	1.16
peep	46.90	70.12	0.67
pg	0.67	0.61	1.10
plan	0.56	0.48	1.17
press1	298.61	126.31	2.36
press2	305.43	31,14	9.81
qsort	0.48	0.56	0.86
queens	0.12	0.12	1.00
read	58.15	38.27	1.52
Mean	63.79	29.92	1.81
Max	305.43	126.31	9.81

Table 26: Granularity: Efficiency Results on the Domain Pattern

the semantic framework to an efficient fixpoint algorithm and its experimental evaluation. The semantic framework demonstrates that pure `OLDT`-resolution is not an appropriate setting for abstract interpretation and proposes an appropriate foundation using the idea of output subsumption. The algorithmic framework presents an efficient fixpoint algorithm by successive refinements. The algorithm includes optimizations such as the caching of abstract operations, output subsumption, and the use of sophisticated dependencies to avoid redundant computations. Comprehensive experimental results about the accuracy and efficiency of the algorithm are presented, using finite and infinite abstract domains. The algorithm is also compared to other algorithms, working at a coarser granularity [19, 21, 14]. Choices and variations of the algorithm are also carefully studied, including the impact of caching, dependencies, output subsumption, widening, and generalization.

The experimental results have been particularly informative. They indicate that OLTD-based abstract interpretation can improve accuracy of the analysis on practical domains, although this was rarely the case in our experiments. The cost of this additional accuracy is high however, since abstract `OLDT`-resolution is about 17 times slower than a more traditional algorithm on the domain `Pattern`. These results is not due to a limitation in our algorithm, since the potential for optimization is rather low (10%). On the smaller domain, both algorithms have the same accuracy and roughly the same efficiency. This indicates that, for the domains considered in this paper, `OLDT`-resolution may work at too fine a granularity. Similar results can be expected on the domain `Prop` [9, 27, 24], since the least upper bound operation in this domain does not lose accuracy. The main open issue now is to find out applications where the tradeoff efficiency/accuracy of `OLDT`-based abstract interpretation is more valuable in practice.

Acknowledgements

Mark Johnson helped clarifying the relationships with computational linguistics. This research was partly supported by the National Science Foundation under grant number CCR-9108032 and the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225.

References

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. (To appear in *ACM Transactions on Programming Languages and Systems*).

[2] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[3] M Bruynooghe and G Janssens. An Instance of Abstract Interpretation: Integrating Type and Mode Inferencing. In *Proc. Fifth International Conference on Logic Programming*, pages 669–683, Seattle, WA, August 1988.

[4] Bruynooghe, M. et al. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc. 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, CA, August 1987.

[5] N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959.

[6] C. Codognet, P. Codognet, and J.M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

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

[8] A. Corsini and G. Filé. A Complete Framework for the Abstract Interpretation of Logic Programs: Theory and Applications. Research report, University of Padova, Italy, 1989.

[9] A. Cortesi, G. Filé, and W. Winsborough. Prop revisited: Propositional formulas as abstract domain for groundness analysis. In *Proc. Sixth Annual IEEE Symposium on Logic in Computer Science (LICS'91)*, pages 322–327, 1991.

[10] 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*, pages 238–252, Los Angeles, CA, 1977.

[11] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, Lecture Notes in Computer Science, Leuven, August 1992. Springer-Verlag. (Invited Paper).

[12] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[13] J. Earley. An Efficient Context-Free Parsing Algorithm. *Comm. ACM*, 13(2):94–102, 1970.

[14] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. In *Fourth International Symposium on Programming Language Implementation and Logic Programming (PLILP-92)*, Leuven (Belgium), August 1992. To Appear in *Software Practice and Experience.*

[15] G Filé and P Sottero. Abstract Interpretation for Type Checking. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP-91)*, Passau (Germany), August 1991.

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

[17] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

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

[19] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and Its Complexity Analysis (Extended Abstract). In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[20] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, CS Department, Brown University, 1992.

[21] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Francisco, CA, April 1992.

[22] B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. In *Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92)*, Washington, DC, November 1992.

[23] B. Le Charlier and P. Van Hentenryck. A Generic Fixpoint Semantics for Logic Programming and its Application for Abstract Interpretation. Technical report, CS Department, Brown University, 1993. Forthcoming.

[24] B. Le Charlier and P. Van Hentenryck. Groundness Analysis for Prolog: Implementation and Evaluation of the Domain `Prop`. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM93)*, Copenhagen, Denmark, June 1993.

[25] K. Marriott and H. Sondergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. North American Conference on Logic Programming, Cleveland, Ohio, 1989.

[26] K. Marriott and H. Sondergaard. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, pages 601–606, San Fransisco, CA, 1989.

[27] K. Marriott and H. Sondergaard. Analysis of Constraint Logic Programs. In *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

[28] C. Mellish. *Abstract Interpretation of Prolog Programs*, pages 181–198. Ellis Horwood, 1987.

[29] K. Musumbu. *Interpretation Abstraite de Programmes Prolog.* PhD thesis, University of Namur (Belgium), September 1990.

[30] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information Through Abstract Interpretation. In *Proceedings of the North American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[31] R.A. O'Keefe. Finite Fixed-Point Problems. In J-L. Lassez, editor, *Fourth International Conference on Logic Programming*, pages 729–743, Melbourne, Australia, 1987.

[32] D. Schmidt. *Denotational Semantics.* Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[33] S.M. Shieber. *Constraint Based Grammar Formalism.* The MIT Press, Cambridge, MA, 1992.

[34] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Ma, 1986.

[35] H Tamaki and T. Sato. OLD-resolution with Tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, July 1986.

[36] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[37] P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The impact of Granularity in Abstract Interpretation of Prolog. Technical report, CS Department, Brown University, 1993. Forthcoming.

[38] L. Vieille. Recursive Axioms in Deductive Databases : the Query/Subquery Approach. In *Proceedings of the First International Conference on Expert Databases Systems*, pages 179–193, Charleston, South Carolina, April 1986.

[39] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69:1–53, 1989.

[40] D.S. Warren. Memoization for Logic Programs. *Communication of the ACM*, 35(3), March 1992.

[41] R. Warren, M. Hermedegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proc. Fifth International Conference on Logic Programming*, pages 684–699, Seattle, WA, August 1988.

[42] W.H. Winsborough. A Minimal Function Graph Semantics for Logic Programs. Technical Report TR-711, Computer Science Department, University of Wisconsin at Madison, August 1987.

Contents