Reexecution in Abstract Interpretation of Prolog

Baudouin Le Charlier[1]
Pascal Van Hentenryck[2]

Technical Report No. CS-92-12
March 1992

[1]University of Namur, 21 rue Grandgagnage, B-5000 Namur Belgium
[2]Brown University, Department of Computer Science Box 1910, Providence, RI 02912 USA

Reexecution in Abstract Interpretation of Prolog

Baudouin Le Charlier
University of Namur,
21 rue Grandgagnage, B-5000 Namur (Belgium)
Email: ble@info.fundp.ac.be

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

Abstract

Logic programming, because of referential transparency, enjoys the property that a goal may be reexecuted arbitrarily often without affecting the meaning of the program. This property, although not interesting computationally in general, can be exploited in abstract interpretation to improve the accuracy of the analysis, as noted by Bruynooghe in [3].

In this paper, we study reexecution from its theoretical foundations to its experimental evaluation. We define a new abstract semantics for Prolog, which incorporates the notion of reexecution, and we study its correctness and precision properties. A fixpoint algorithm to compute (parts of) the abstract semantics is then presented. The accuracy and efficiency of the algorithm is evaluated experimentally on two abstract domains, a simple domain and an elaborate domain, and compared with conventional approaches.

The experimental results indicate that (1) reexecution can provide significant gain in accuracy at a very reasonable computation cost and that (2) reexecution on a simple domain is a versatile alternative to the standard approach on a more sophisticated domain.

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 (e.g. [21, 28, 19, 20, 3, 2, 7, 1]), the algorithms (e.g. [24, 2, 6, 16, 14]), the abstract domains (e.g. [23, 4, 13]) and the implementation (e.g. [27, 12, 18, 11]). Recent results indicate that abstract interpretation can be competitive with specialized data flow algorithms and could be integrated in industrial compilers.

Traditionally, an application of abstract interpretation for Prolog requires the definition of an abstract domain which captures, through abstract substitutions, the information relevant to the application. A generic algorithm (e.g. [2, 16, 7]) can then be instantiated to this abstract domain. If the level of precision provided by the resulting algorithm is

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

Figure 1: The Normalized Version of Append

not satisfactory, another, more sophisticated, abstract domain is defined to overcome the limitations.

In this paper, we investigate another (complementary) approach. The approach exploits the referential transparency of logic programming which guarantees that a goal may be reexecuted arbitrarily often without affecting the program. Although not interesting computationally in general, this property can be used in abstract interpretation to improve the precision of the analysis as noted by Bruynooghe as early as 1987 [3] (See also [2]).

As a motivating example, consider abstract interpretation using a simple domain, containing modes, sharing, and same value components on the append program depicted in Figure 1. Assume also that the input substitution includes the modes (Any,Var,Ground) for X_1,X_2,X_3. Ideally, the output modes (Ground,Ground,Ground) for the same variables should be contained in the output substitution. However, the simple domain returns an output substitution with modes (Any,Ground,Ground). Figure 2 depicts the second iteration of the algorithm and illustrates why the analysis loses precision. On return of the recursive call, the algorithm needs to propagate the fact that variable X_5 is Ground. Since X_4 is ground as well, X_1 should now be Ground. However, the type information relating X_1 to X_4 and X_5 has been lost and the mode of X_1 cannot be improved upon. The traditional way of overcoming the problem amounts to preserving more information such as type information on the variables, e.g. remembering that X_1 is bound to $[X_4|X_5]$ as done in the elaborate domain used later in the paper.[1] However, a similar effect can be achieved (for the append program) simply by reexecuting the first goal of the clause. The reexecution automatically assigns the mode Ground to X_1.

The research described in this paper originates in an attempt to study theoretically and experimentally an algorithm that would automatically reexecute goals to achieve more precision. We propose a new abstract semantics which generalizes the traditional basic abstract semantics and incorporates the idea of reexecution. The reexecution semantics makes use of a new operation REFINE characterized axiomatically and whose properties are studied. The abstract semantics is proven safe wrt the operational semantics and improves (or at least preserves) the precision of the basic semantics. A fixpoint algorithm computing the abstract semantics is then presented and its implementation choices discussed. The algorithm is then evaluated experimentally on two domains: a simple domain and a type

[1] Other approaches have been explored including covering and the Prop domain.

```
TRY CLAUSE 1
   EXIT EXTC **CACHED** (Any,Var,Ground)  SV: [{1}{2}{3}]  PS: [{1}{2}{3}]
   CALL UNIF-FUN  (Any,Var,Ground)  SV: [{1}{2}{3}]  PS: [{1}{2}{3}]
   EXIT UNIF-FUN **CACHED **  (Ground,Var,Ground)  SV: [{1}{2}{3}]  PS:
 [{1}{2}{3}]
   CALL UNIF-VAR  (Ground,Var,Ground)  SV: [{1}{2}{3}]  PS: [{1}{2}{3}]
   EXIT UNIF-VAR  ** CACHED **  (Ground,Ground,Ground)  SV: [{1}{2,3}]  PS:
 [{1}{2}{3}]
   EXIT RESTRC  ** CACHED **  (Ground,Ground,Ground)  SV: [{1}{2,3}]  PS:
 [{1}{2}{3}]
   EXIT UNION  ** CACHED **  (Ground,Ground,Ground)  SV: [{1}{2,3}]  PS:
 [{1}{2}{3}]
EXIT CLAUSE 1
TRY CLAUSE 2
   EXIT EXTC  ** CACHED **  (Any,Var,Ground,Var,Var,Var)
               SV: [{1}{2}{3}{4}{5}{6}]  PS: [{1}{2}{3}{4}{5}{6}]
   CALL UNIF-FUN  (Any,Var,Ground,Var,Var,Var)
               SV: [{1}{2}{3}{4}{5}{6}]  PS: [{1}{2}{3}{4}{5}{6}]
   EXIT UNIF-FUN  ** CACHED **  (Any,Var,Ground,Any,Any,Var)
               SV: [{1}{2}{3}{4}{5}{6}]  PS: [{1,4,5}{2}{3}{6}]
   CALL UNIF-FUN  (Any,Var,Ground,Any,Any,Var)
               SV: [{1}{2}{3}{4}{5}{6}]  PS: [{1,4,5}{2}{3}{6}]
   EXIT UNIF-FUN  ** CACHED **  (Any,Var,Ground,Ground,Any,Ground)
               SV: [{1}{2}{3}{4}{5}{6}]  PS: [{1,5}{2}{3}{4}{6}]
   CALL PRO-GOAL  append(Any,Var,Ground)  SV: [{1}{2}{3}]  PS: [{1}{2}{3}]
   EXIT PRO-GOAL  append(Ground,Ground,Ground)  SV: [{1}{2,3}]  PS: [{1}{2}{3}]
   EXIT EXTG  (Any,Ground,Ground,Ground,Ground,Ground)
               SV: [{1}{2,6}{3}{4}{5}]  PS: [{1}{2}{3}{4}{5}{6}]
   EXIT RESTRC  (Any,Ground,Ground)  SV: [{1}{2}{3}]  PS: [{1}{2}{3}]
   EXIT UNION  (Any,Ground,Ground)  SV: [{1}{2}{3}]  PS: [{1}{2}{3}]
EXIT CLAUSE 2
```

Figure 2: The Second Iteration on the Simple Domain

domain. Results comparing the standard algorithm with the reexecution algorithm are given both in terms of accuracy and efficiency of the analysis. We also compare the reexecution algorithm on the simple domain with the standard algorithm on the elaborate domain to demonstrate that reexecution on a simple domain provide a complementary and versatile alternative to the standard algorithm on a more sophisticated domain. Experimental results evaluating the impact of implementation choices are also discussed.

The rest of the paper is organized as follows. The next section introduces the operational semantics which is used in the safeness results. Section 3 recalls the basic abstract semantics which serves as a basis for the new abstract semantics introduced in Section 4. Section 5 presents the fixpoint algorithm for the reexecution semantics while Section 6 reports the experimental results. The last section contains the conclusion of the paper.

2 The Operational Semantics

2.1 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 $x_1, x_2, \ldots, x_i, \ldots$.

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 $x_1, \ldots, x_n$.

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 SS the set of *ss*.

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 $\sigma \in SS$ 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 SS hereafter. Finally, we note $var(S)$ the set of variables appearing in any syntactic object S.

2.2 Abstract Syntax

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

$$
\begin{array}{lcl}
P & \in & Programs \\
pr & \in & Procedures \\
c & \in & Clauses \\
sg & \in & Goals \\
g & \in & Atoms\ (or\ Subgoals) \\
f & \in & Functors \\
p & \in & ProcedureNames \\
x_i & \in & ProgramVariables
\end{array}
$$

$$
\begin{array}{lcl}
P & ::= & \{pr_1\ ,\ \ldots\ ,\ pr_n\} \\
pr & ::= & <> \mid pr.c \\
c & ::= & p(x_1\ ,\ \ldots\ ,\ x_n) \leftarrow sg \\
sg & ::= & <> \mid sg.g \\
g & ::= & x_i = x_j \mid x_{i_1} = f(x_{i_2}\ ,\ \ldots\ ,\ x_{i_n}) \mid p(x_{i_1}\ ,\ \ldots\ ,\ x_{i_n})
\end{array}
$$

We note $head(c)$ (resp. $head(p)$) the head of a clause c (resp. a procedure p).

2.3 States

The concrete semantics operates on five different computation states:

$\langle \theta, p \rangle$	where $dom(\theta) = var(head(p))$	(p is called with input θ)
$\langle \theta, c \rangle$	where $dom(\theta) = var(head(c))$	(c is called with input θ)
$\langle \theta, sg \rangle$	where $var(sg) \subseteq dom(\theta)$	(resolution of $sg\theta$ is initiated)
$\langle \theta, g \rangle$	where $var(g) \subseteq dom(\theta)$	(atom $g\theta$ is selected)
θ		(success substitution)

2.4 Concrete Primitive Operations

The operational semantics uses five primitive operations[2]. In the definitions, we make the assumptions that $var(c) = \{x_1, \ldots, x_m\}$, $var(head(c)) = \{x_1, \ldots, x_n\}$, and $x_{i_1}, \ldots, x_{i_n}$ is the sequence of variables occurring in g (from left to right).

$$
\begin{array}{rcl}
\mathrm{EXTC}(c, \theta) & = & \{\ \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n, \ldots, x_{n+1} \leftarrow y_1, \ldots, x_m \leftarrow y_{m-n}\} :\ t_i = x_i\theta\ (1 \leq i \leq n), \\
 & & \qquad y_1, \ldots, y_{m-n} \in SV,\ y_1, \ldots, y_{m-n} \text{ are all distinct, and} \\
 & & \qquad var(t_1, \ldots, t_n) \cap \{y_1, \ldots, y_{m-n}\} = \emptyset\ \}
\end{array}
$$

$$
\mathrm{RESTRC}(c, \theta) \ = \ \{\theta_{/\{x_1, \ldots, x_n\}}\ \}
$$

[2]Operation **RENAME** is in fact used implicitly but it is important for the rest of the paper.

$$\texttt{RESTRG}(g,\theta) \;=\; \{\{x_1 \leftarrow x_{i_1}\theta, \ldots, x_n \leftarrow x_{i_n}\theta\}\}$$

$$\texttt{EXTG}(g,\theta_1,\theta_2) = \{\; \theta_1\sigma \;: \exists\theta_3 \text{ such that } \theta_3 = \texttt{RESTRG}(g,\theta_1) \text{ and } \theta_2 = \theta_3\sigma \text{ where } dom(\sigma) \subseteq codom(\theta_3) \text{ and } (codom(\theta_1) - codom(\theta_3)) \cap codom(\sigma) = \emptyset \;\}$$

$$\texttt{RENAME}(g,\theta) \;= \{\{x_{i_1} \leftarrow x_1\theta, \ldots, x_{i_n} \leftarrow x_n\theta\}\}$$

Note that the five operations returns a *set* of program substitutions although the result is conceptually a single substitution in general. This is formally convenient because all operations can then be treated the same way in all cases. (EXTC returns an infinite set of program substitutions equivalent up to renaming. EXTG returns sometimes an empty set.)

2.5 Transition Rules

Procedure Call

$$\frac{\begin{array}{l} c \;::=\; p(x_1,\ldots,x_n) \;\leftarrow\; sg \\ \langle\theta, c\rangle \longmapsto \theta' \end{array}}{\langle\theta, p\rangle \longmapsto \theta'}$$

Clause Call

$$\frac{\begin{array}{l} c \;::=\; p(x_1,\ldots,x_n) \;\leftarrow\; sg \\ \theta_1 \in \texttt{EXTC}(c,\theta) \\ \langle\theta_1, sg\rangle \longmapsto \theta_2 \\ \theta' \in \texttt{RESTRC}(c,\theta_2) \end{array}}{\langle\theta, c\rangle \longmapsto \theta'}$$

Execution of the Empty Goal

$$\frac{sg \;::=\; <>}{\langle\theta, sg\rangle \longmapsto \theta}$$

Execution of a Non Empty Goal

$$\frac{\begin{array}{l} sg \;::=\; sg'.g \\ \langle\theta, sg'\rangle \longmapsto \theta_1 \\ \langle\theta_1, g\rangle \longmapsto \theta' \end{array}}{\langle\theta, sg\rangle \longmapsto \theta'}$$

Selection of an Atom

$$\frac{\begin{array}{l} g ::= x_{i_1} = x_{i_2} \\ \theta_1 \in \texttt{RESTRG}(g, \theta) \\ \sigma \in mgu(x_1\theta_1, x_2\theta_1) \\ \theta' \in \texttt{EXTG}(g, \theta, \theta_1\sigma) \end{array}}{\langle \theta, g \rangle \longmapsto \theta'}$$

$$\frac{\begin{array}{l} g ::= x_{i_1} = f(x_{i_2}, \ldots, x_{i_n}) \\ \theta_1 \in \texttt{RESTRG}(g, \theta) \\ \sigma \in mgu(x_1\theta_1, f(x_2, \ldots, x_n)\theta_1) \\ \theta' \in \texttt{EXTG}(g, \theta, \theta_1\sigma) \end{array}}{\langle \theta, g \rangle \longmapsto \theta'}$$

$$\frac{\begin{array}{l} g ::= p(x_{i_1}, \ldots, x_{i_n}) \\ \theta_1 \in \texttt{RESTRG}(g, \theta) \\ \langle \theta_1, p \rangle \longmapsto \theta_2 \\ \theta' \in \texttt{EXTG}(g, \theta, \theta_2) \end{array}}{\langle \theta, g \rangle \longmapsto \theta'}$$

where $mgu(t_1, t_2)$ is the set of all most general unifiers of t_1 and t_2 (using standard variables only).

2.6 Properties of the Operational Semantics

The following theorems are consequences of some well-known results in logic programming.

Theorem 2 [Equivalence wrt SLD-resolution]
Let p be a procedure name of arity n and θ be a program substitution on variables $x_1, \ldots, x_n$. Then $\langle \theta, p \rangle \longmapsto \theta'$ if and only if there exists a correct answer substitution σ for the query $p(x_1, \ldots, x_n)\theta$ such that $\theta\sigma = \theta'$. ∇

Theorem 3 [Reexecution theorem] Let *cons* be a procedure name, a clause, a goal or an atom, θ be a program substitution such that $\langle \theta, cons \rangle$ is a computation state and σ be a standard substitution. Then $\langle \theta'\sigma, cons \rangle \longmapsto \theta'\sigma$ whenever $\langle \theta, cons \rangle \longmapsto \theta'$. ∇

3 The Basic Abstract Semantics

In this section, we recall the abstract semantics proposed in [22] and used subsequently in [15, 16, 18, 11] to derive abstract interpretation algorithms which were comparable in accuracy and efficiency to the specialized tools [27, 12]. The new semantics to be presented in the next section is a generalization of the basic semantics.

For each finite set D of program variables we assume the existence of a *cpo* AS_D whose elements are called abstract substitutions on domain D and denoted by β. Let CS_D be the set of program substitutions having domain D. The meaning of each abstract substitution

is given through the concretization function: $Cc : AS_D \rightarrow \mathcal{P}(CS_D)$. We assume in the following that Cc is monotone[3].

The basic abstract semantics uses seven (families of) abstract primitive operations EXTC, RESTRG, EXTG, RESTRC, UNION, AI_VAR and AI_FUNC. The first four operations are safe abstractions of the corresponding "concrete" operations used by the operational semantics. Recall that an (abstract) operation

$$o_a : AS_{D_1} \times \ldots \times AS_{D_n} \rightarrow AS_D$$

is said to be a safe abstraction of a (concrete) operation

$$o_c : CS_{D_1} \times \ldots \times CS_{D_n} \rightarrow \mathcal{P}(CS_D)$$

if and only if

$$(\forall i : 1 \leq i \leq n : \theta_i \in Cc(\beta_i)) \Rightarrow o_c(\theta_1, \ldots, \theta_n) \subseteq Cc(o_a(\beta_1, \ldots, \beta_n))$$

for all $\theta_i \in AS_{D_i}$ and $\beta_i \in CS_{D_i}$.

The last three operations are safe in the following sense (assuming $D = \{x_1, \ldots, x_n\}$):

$$\forall \theta \in CS_D : (\theta \in Cc(\beta_1) \text{ or } \theta \in Cc(\beta_2)) \Rightarrow \theta \in Cc(\texttt{UNION}(\beta_1, \beta_2))$$

$$\left.\begin{array}{ll} \forall \theta \in CS_{\{x_1,x_2\}} & \sigma \in mgu(x_1\theta, x_2\theta) \\ \forall \beta \in AS_{\{x_1,x_2\}} : & \theta \in Cc(\beta) \end{array}\right\} \Rightarrow \theta\sigma \in Cc(\texttt{AI\_VAR}(\beta))$$

$$\left.\begin{array}{ll} \forall \theta \in CS_D & \sigma \in mgu(x_1\theta, f(x_2, \ldots, x_n)\theta) \\ \forall \beta \in AS_D : & \theta \in Cc(\beta) \end{array}\right\} \Rightarrow \theta\sigma \in Cc(\texttt{AI\_FUNC}(\beta))$$

The abstract semantics of a program P is defined as a set of abstract tuples $(\beta_{in}, p, \beta_{out})$ where p is a predicate symbol of arity n occurring in P, $D = \{x_1, \ldots, x_n\}$ and $\beta_{in}, \beta_{out} \in AS_D$. For convenience, we assume in the following an underlying program P. The *underlying domain* UD of the program is the set of all (β_{in}, p) such that $\beta_{in} \in AS_D$ and p occurs in P. We denote $SATT$ the set of all functional, total, and monotone sets of abstract tuples sat, i.e. the sets sat which satisfy

1. $\forall\, (\beta_{in}, p) \in UD\ \exists!\ \beta_{out} \in AS_D\ :\ (\beta_{in}, p, \beta_{out}) \in sat$;
2. $\forall (\beta_1, p), (\beta_2, p) \in UD :\ \beta_1 \leq \beta_2\ \Rightarrow\ sat(\beta_1, p) \leq sat(\beta_2, p)$.

$SATT$ is endowed with the following ordering:

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

We note $\beta_{out} = sat(\beta_{in}, p)$ iff $(\beta_{in}, p, \beta_{out}) \in sat$.

$TSAT(sat) = \{(\beta, p, \beta') : (\beta, p) \in UD \text{ and } \beta' = T_p(\beta, p, sat)\}$.

$T_p(\beta, p, sat) = \texttt{UNION}(\beta_1, \ldots, \beta_n)$
where $\beta_i = T_c(\beta, c_i, sat)$,
$c_1, \ldots, c_n$ are the clauses of p.

$T_c(\beta, c, sat) = \texttt{RESTRC}(c, \beta')$
where $\beta' = T_b(\texttt{EXTC}(c, \beta), b, sat)$,
b is the body of c.

$T_b(\beta, <>, sat) = \beta$.
$T_b(\beta, g.gs, sat) = T_b(\beta_3, gs, sat)$
where $\beta_3 = \texttt{EXTG}(g, \beta, \beta_2)$,

$$\beta_2 = \begin{cases} sat(\beta_1, p) & \text{if } g \text{ is } p(\ldots) \\ \texttt{AI\_VAR}(\beta_1) & \text{if } g \text{ is } x_i = x_j \\ \texttt{AI\_FUNC}(\beta_1, f) & \text{if } g \text{ is } x_i = f(\ldots), \end{cases}$$

$\beta_1 = \texttt{RESTRG}(g, \beta)$.

Figure 3: The Basic Abstract Semantics

We are now in position to define the abstract semantics which is depicted in Figure 3. The semantics of P is, by definition, the least fixpoint of the transformation $TSAT$, denoted $lfp(TSAT)$. It is guaranteed to exist provided that the primitive operations be monotone.[4]

3.1 Safeness of the Basic Abstract Semantics

Theorem 4 Let p be a predicate symbol of arity n in P, $D = \{x_1, \ldots, x_n\}$, $\theta, \theta' \in CS_D$, $\beta \in AS_D$ and $sat = lfp(TSAT)$. The following property holds.

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

Proof See [22]. □

As we will see later on, the reexecution semantics preserves this safeness property. In addition, reexecution makes sure that its results are at least as precise as the basic semantics.

[3] This condition is not always required for the basic semantics as mentioned in [16] and fully explained in [22]. It is however a requirement for the reexecution semantics.

[4] In fact, monotonicity of the primitive operations is not a strict requirement (see [22]). It simplifies the presentation however because it ensures that the least fixpoint exists and is the best abstract characterization of the program meaning.

4 The Reexecution Abstract Semantics

In this section, we present the new abstract semantics. A new abstract operation, REFINE, is first introduced. The reexecution semantics is defined and its properties studied. We conclude the section by a study of the properties of REFINE.

4.1 The REFINE Operation

4.1.1 Motivation

The abstract interpretation of a clause c of the form $p(x_1, \ldots, x_n) \leftarrow g_1, \ldots, g_m$ for an input abstract substitution β_{in} can be viewed as the annotation of c with abstract substitutions $\beta_{in}, \beta_0, \ldots, \beta_m$, say

$$\beta_{in}\ p(x_1, \ldots, x_n) \leftarrow \beta_0\ g_1\ \beta_1\ , \ldots, \beta_{m-1}\ g_m\ \beta_m.$$

satisfying $\theta_i \in Cc(\beta_i)$ for all $\theta_0, \theta_1, \ldots, \theta_m$ such that

- $\theta_{in} \in Cc(\beta_{in})$;
- $\theta_0 \in \mathtt{EXTC}(c, \theta_{in})$;
- $\langle \theta_{i-1}, g_i \rangle \longmapsto \theta_i \ \ (1 \leq i \leq m)$.

θ_i is more instantiated than θ_j when $(i > j)$. Hence $\langle \theta_i, g_j \rangle \rightarrow \theta_i$ by the reexecution theorem. The above property allows the reexecution of subgoal g_j on abstract substitution β_i and guarantees that the resulting abstract substitution β_i' safely approximates θ_i (i.e. $\theta_i \in Cc(\beta_i')$). Moreover, the property holds for any subgoal g_j $(1 \leq j \leq i)$ and allows for multiple reexecutions.

Note however that naive reexecution is not guaranteed to improve the accuracy of β_i. In fact, it may even decrease its accuracy.[5] The purpose of the REFINE operation is to make sure that reexecution improves (or at least preserves) accuracy.

4.1.2 Axiomatic Characterization

For some set of variables $D = \{x_1, ..., x_n\}$, let us note $Aseq_D$ the set of finite sequences $(\beta_1, \ldots, \beta_q)$ such that $q \geq 0$ and $dom(\beta_i) \subseteq D$ $(\forall i : 1 \leq i \leq q)$.

Definition 5 A *refinement* operation on D is any operation

$$\mathtt{REFINE} : AS_D \times Aseq_D \rightarrow AS_D$$

satisfying

1. $\forall \beta \in AS_D\ \forall (\beta_1, \ldots, \beta_q) \in Aseq_D : \mathtt{REFINE}(\beta, \beta_1, \ldots, \beta_q) \leq \beta$.

[5] Sufficient properties to improve accuracy are discussed in Section 4.4.

2. For all $D_1, \ldots, D_q \subseteq D$, $\beta \in AS_D$, $(\beta_1, \ldots, \beta_q) \in AS_{D_1} \times \ldots \times AS_{D_q}$ and $\theta \in Cc(\beta)$:

$$\theta_{/D_1} \in Cc(\beta_1) \;\&\; \ldots \;\&\; \theta_{/D_q} \in Cc(\beta_q) \;\Rightarrow\; \theta \in Cc(\texttt{REFINE}(\beta, \beta_1, \ldots, \beta_q)).$$

3. Let $\beta \in AS_D$ and $seq_1, \ldots, seq_i, \ldots$ be an infinite sequence of elements from $Aseq_D$. The following properties hold

   (a) the sequence $\beta_0, \ldots, \beta_i, \ldots$ has a greatest lower bound denoted $\bigsqcap_{i=1}^{\infty} \beta_i$
   (b) $Cc(\bigsqcap_{i=1}^{\infty} \beta_i) = \bigcap_{i=1}^{\infty} Cc(\beta_i)$;

   for all infinite decreasing sequence $\beta_0 \geq \ldots \geq \beta_i \geq \ldots$ defined by

$$\begin{aligned} \beta_0 &= \beta, \\ \beta_i &= \texttt{REFINE}(\beta_{i-1}, seq_i) \quad (i \geq 1). \end{aligned}$$ ∇

Note first that the existence of the greatest lower bound is only enforced on certain infinite sequences (and not all of them). Note also that existence of refinement operations for practical applications is rather obvious.

Example 6 The simplest REFINE operation is defined by

$$\texttt{REFINE}(\beta, seq) = \beta \quad (\forall \beta \; \forall seq)$$

The reexecution semantics with this operation reduces to the basic semantics. ∇

Example 7 On some domains, REFINE can be defined in terms of an *exact join* operation, say $\bowtie$. For all $\beta_1 \in AS_{D_1}$, $\beta_2 \in AS_{D_2}$, the exact join satisfies

$$Cc(\beta_1 \bowtie \beta_2) = \{\, \theta : dom(\theta) = D_1 \cup D_2 \;\&\; \theta_{/D_1} \in Cc(\beta_1) \;\&\; \theta_{/D_2} \in Cc(\beta_2) \,\}$$

Note that $dom(\beta_1 \bowtie \beta_2) = dom(\beta_1) \cup dom(\beta_2)$. With the exact join, REFINE is then defined as

$$\texttt{REFINE}(\beta, \beta_1, \ldots, \beta_n) = \beta \bowtie \beta_1 \bowtie \ldots \bowtie \beta_n. \;\nabla$$

4.2 The Reexecution Semantics

We are now in position to define the reexecution semantics. The reexecution semantics is the same as the basic abstract semantics except for the function T_b which is depicted in Figure 4. The new function T_b makes use of two other semantic functions T_g and T_r corresponding respectively to a normal goal execution and to a goal reexecution. It also makes use of the greatest lower bound and the operation REFINE. Note also that the semantic function T_r makes use of the renaming operation RENAME which is a safe abstraction of the corresponding concrete operation.

The abstract semantics of a program is defined as the least fixpoint of $TSAT$. It is guaranteed to exist if the primitive operations are monotone. In practice, this may be too strong a requirement, especially for an operation such as REFINE. Fortunately, prefixpoints of $TSAT$ (i.e. the sets sat such that $sat \geq TSAT(sat)$) are always guaranteed to exist which is sufficient for the purpose of abstract interpretation [22, 9]. The reexecution algorithm is of interest provided that they compute a prefixpoint of the reexecution semantics which is more accurate than the least fixpoint of the basic semantics.

$T_b(\beta, <>, sat) = \beta.$

$T_b(\beta, (g_1, \ldots, g_k), sat) = \sqcap_{i=1}^{\infty} \beta_i \quad (k \geq 1)$
where $\beta_0 = T_b(\beta, (g_1, \ldots, g_{k-1}), sat)$
$\beta_1 = T_g(\beta_0, g_k, sat)$
$\beta_{i+1} = \texttt{REFINE}(\beta_i, T_r(\beta_i, g_1, sat), \ldots, T_r(\beta_i, g_k, sat)) \quad (i \geq 1)$

$T_g(\beta, g, sat) = \texttt{EXTG}(g, \beta, \beta_2)$
where $\beta_2 = \begin{cases} sat(\beta_1, p) & \text{if } g \text{ is } p(\ldots) \\ \texttt{AI\_VAR}(\beta_1) & \text{if } g \text{ is } x_i = x_j \\ \texttt{AI\_FUNC}(\beta_1, f) & \text{if } g \text{ is } x_i = f(\ldots), \end{cases}$
$\beta_1 = \texttt{RESTRG}(g, \beta).$

$T_r(\beta, g, sat) = \texttt{RENAME}(g, \beta_2)$
where $\beta_2 = \begin{cases} sat(\beta_1, p) & \text{if } g \text{ is } p(\ldots) \\ \texttt{AI\_VAR}(\beta_1) & \text{if } g \text{ is } x_i = x_j \\ \texttt{AI\_FUNC}(\beta_1, f) & \text{if } g \text{ is } x_i = f(\ldots), \end{cases}$
$\beta_1 = \texttt{RESTRG}(g, \beta).$

Figure 4: The Reexecution Semantics: The Function T_b

4.3 Properties of the Reexecution Semantics

In this section, we study various properties of the reexecution semantics.

4.3.1 Safeness

We first establish the safeness of the reexecution semantics. To simplify notations, we denote by sg_k the sequence of goals $g_1, \ldots, g_k$.

Theorem 8 Let sat be any prefixpoint of $TSAT$. Let D, θ, β be such that $var(sg_k) \subseteq D = var(\theta) = var(\beta)$. Then

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

Proof The proof is by induction on k. The base case $k = 0$ is trivial. For the induction step, note that $\langle \theta, sg_k \rangle \longmapsto \theta'$ implies $\langle \theta', sg_j \rangle \longmapsto \theta' \quad (1 \leq j \leq k)$ by the reexecution theorem. We first prove (1)(2) that $\theta' \in Cc(\beta_i)$ $(i \geq 1)$ with the β_i defined as in the new semantic function T_b, i.e.

$$\begin{array}{l} \beta_0 = T_b(\beta, sg_{k-1}, sat) \\ \beta_1 = T_g(\beta_0, g_k, sat) \\ \beta_{i+1} = \texttt{REFINE}(\beta_i, T_r(\beta_i, g_1, sat), \ldots, T_r(\beta_i, g_k, sat)) \quad (i \geq 1). \end{array}$$

We then prove (3) that $\theta' \in Cc(\beta')$ where $\beta' = \sqcap_{i=1}^{\infty} \beta_i$.

1. $\theta' \in Cc(\beta_1)$: $\langle \theta, sg_k \rangle \longmapsto \theta'$ implies the existence of θ_{inter} such that

   $$\langle \theta, sg_{k-1} \rangle \longmapsto \theta_{inter} \text{ and } \langle \theta_{inter}, g_k \rangle \longmapsto \theta'.$$

   By induction on the length of the transition sequences, we have

   $$\begin{array}{llllll} \theta_{inter} & \in & Cc(\beta_0) & \text{where} & \beta_0 & = & T_b(\beta, sg_{k-1}, sat); \\ \theta' & \in & Cc(\beta_1) & \text{where} & \beta_1 & = & T_g(\beta_0, g_k, sat). \end{array}$$

2. $\theta' \in Cc(\beta_i)$ for all $i \geq 1$: The proof is by induction on i and the base case $i = 1$ has just been proven. If $i > 1$, then

   $$\begin{array}{l} \theta' \in Cc(\beta_{i-1}) \quad \text{(by induction)} \\ \theta' \in Cc(T_r(\beta_{i-1}, g_j, sat)) \quad (1 \leq j \leq k) \text{ as } \langle \theta', g_j \rangle \longmapsto \theta' \text{ and } \theta' \in Cc(\beta_{i-1}). \end{array}$$

   Hence, since $\beta_i = \texttt{REFINE}(\beta_{i-1}, T_r(\beta_{i-1}, g_1, sat), \ldots, T_r(\beta_{i-1}, g_k, sat))$, we have $\theta' \in Cc(\beta_i)$ by property 2 of REFINE.

3. Points (1) and (2), in conjunction with property 3 of REFINE, implies $\theta' \in Cc(\sqcap_{i=1}^{\infty} \beta_i)$. This completes the proof.

□

4.3.2 Accuracy

The following theorem establishes that the reexecution semantics is at least as precise as the basic semantics.

Theorem 9 Let T_b^r and T_b^b denote the reexecution and basic versions of the T_b function. Then, for any β, sg_k and sat, $T_b^r(\beta, sg_k, sat) \leq T_b^b(\beta, sg_k, sat)$.

Proof Consequence of Property 1 of REFINE. □

4.4 Notes on Operation REFINE

4.4.1 Practical Computation

The semantic functions should not be seen as suggesting any particular practical implementation. In fact, the expression

$$\texttt{REFINE}(\beta_{i-1}, T_r(\beta_{i-1}, g_1, sat), \ldots, T_r(\beta_{i-1}, g_k, sat))$$

is best viewed as an operation which automatically chooses which arguments to reevaluate using, say, *call by name* instead of *call by value*.[6] The choices taken in our implementation are discussed in a subsequent section and amounts to applying REFINE on a single goal.

4.4.2 Exact Join

In presence of an exact join, $\texttt{REFINE}(\beta_{i-1}, \ldots)$ may be computed as $\beta_{i-1} \bowtie T_r(\beta_{i-1}, g_j, sat)$ for a *selected goal* g_j.

The existence of an exact join is rather natural in the context of abstract interpretation and has empirical side and theoretical justifications. On the empirical, the two domains used in our experiments enjoy an exact join although they were originally designed for the basic semantics. On the theoretical side, the exact join operation is a generalization of an abstract intersection operation verifying

$$\theta \in Cc(\beta_1) \;\&\; \theta \in Cc(\beta_2) \;\Rightarrow\; \theta \in Cc(\beta_1 \cap \beta_2).$$

Since it is natural to assume $\beta_1 \cap \beta_2 \leq \beta_1, \beta_2$, $Cc(\beta_1 \cap \beta_2) \subseteq Cc(\beta_1) \cap Cc(\beta_2)$ follows by monotonicity of Cc. Hence $Cc(\beta_1 \cap \beta_2) = Cc(\beta_1) \cap Cc(\beta_2)$.

Note also that, in this case, abstract intersection coincides with the glb which is not always a safe abstraction of intersection.

4.4.3 General Case

In the absence of a join operation, several possibilities exist for REFINE. One of them amounts to using EXTG as follows.

[6] Strictly speaking, it would be more precise to add one more argument to the operation to provide the relevant information. This would however complicate the exposition.

If there exists at least one $\beta'_j = \texttt{EXTG}(\beta, g_j, \beta_j)$ $(1 \leq j \leq q)$ such that $\beta'_j \leq \beta$, say β'_i, define

$$\texttt{REFINE}(\beta, \beta_1, \ldots, \beta_n) = \beta'_i.$$

Otherwise, define

$$\texttt{REFINE}(\beta, \beta_1, \ldots, \beta_n) = \beta.$$

This brute force method can be improved upon when the abstract domain consists of several components, say

$$\beta = (\beta^1, \ldots, \beta^r)$$

In general, the ordering on β is closely related to (component) orderings on $\beta^1, \ldots, \beta^r$. Hence $\beta^k_j \leq \beta^k$ may hold for some components. The new substitution may consist of the decreasing components of β_j and the remaining components from β.[7]

REFINE is also closely connected to the notions *downwards* and *upwards closed* abstract domains (e.g. [5]). AS_D is downwards closed if, for all $\beta \in AS_D$,

$$\theta \in Cc(\beta) \;\Rightarrow\; \theta\sigma \in Cc(\beta).$$

It is upwards closed if, for all $\beta \in AS_D$,

$$\theta\sigma \in Cc(\beta) \;\Rightarrow\; \theta \in Cc(\beta).$$

Reexecution can only instantiate further the arguments. Hence, if the domain is downwards closed, we have

$$\langle \theta, g \rangle \longmapsto \theta' \quad \Rightarrow \quad \theta' \in Cc(\beta).$$

It is thus likely that the abstract reexecution of g produces β' such that $\beta' \leq \beta$. If the domain is upwards closed however, reexecution is likely to produce a greater (less precise) result. Typical downwards closed domains are domains for types, covering and aliasing. Typical upwards closed domains are domains for (possible) sharing of variables between terms.[8]

Practical abstract domains incorporate several components, some being downwards closed and others upwards closed. As a consequence, reexecution can be performed by collecting the information from the downwards closed components only. Of course more specific treatments are often possible.

4.4.4 Relation to Cousot's Narrowing

REFINE relates also to the *narrowing* operation introduced by the Cousots in [8]. The main difference is that REFINE is more related to semantic issues while narrowing is more related to approximate fixpoint computations. REFINE allows a more accurate semantics while narrowing allows a better approximation of a fixpoint. This seems to be two sides of the same coin although more theoretical work is needed to state the equivalence of both notions in a

[7] Practical domains are often more complicated because of constraints relating the components.

[8] Domains for modes are neither downwards nor upwards closed because **ground** is downwards closed while **var** is upwards closed. **any** is both.

formal way. The main difference seems to be that REFINE must provide a safe abstraction of a concrete join operation on substitutions while narrowing is not concerned with safeness. However narrowing is only used to compute upper approximations of a fixpoint which is safe by hypothesis.

5 The Algorithm

We now present a fixpoint algorithm to compute the abstract semantics. The fixpoint algorithm is an instance of a universal top-down fixpoint algorithm [17] whose correctness has been proven. It is also a generalization of the standard algorithm designed for the basic abstract semantics [16, 18]. The standard algorithm was shown to be practical and competitive with specific tools [18]. We first recall a number of notions from the standard algorithm, then present the reexecution algorithm, and finally discuss the implementation issues.

5.1 Preliminaries

5.1.1 Manipulation of the Set of Abstract Tuples

The abstract semantics needs no operation on sets of abstract tuples besides the ability to look up a value. Our generic algorithm however needs to construct a subset of the fixpoint containing the value of the query. Hence it needs a number of operations on sets of abstract tuples.

There are mainly two new operations that need to be defined: EXTEND and ADJUST. Informally speaking, the first operation is intended to extend a set of tuples with a new element while ADJUST is intended to update the result of a pair (β, p).

The two operations can be specified as follows, assuming that the domain of a set of abstract tuples sat, denoted $dom(sat)$, is the set of pairs (β, p) for which there exists a β' such that $(\beta, p, \beta') \in sat$.

- EXTEND(β, p, sat), given an abstract substitution β, a predicate symbol p, and a set of abstract tuples sat which does not contain (β, p) in its domain, returns a set of abstract tuples sat' containing (β, p) in its domain. Moreover the value $sat'(\beta, p)$ is defined as the *lub* (i.e. the least upper bound) of all $sat(\beta', p)$ such that $\beta' \leq \beta$.

- ADJUST(β, p, β', sat), where β' represents a new result computed for the pair (β, p), returns a sat' which is sat updated with this new result. More precisely, the value of $sat'(\beta'', p)$ for all $\beta'' \geq \beta$ is equal to $lub\{\beta', sat(\beta'', p)\}$ and all other values are left unchanged. In the algorithm, we use a slightly more general version of ADJUST which, in addition to the new set of abstract tuples, returns the set of pairs (β, p) whose values have been updated.

As should be clear, operation EXTEND computes the best possible approximation for the new pairs by taking the *lub* of all values. Operation ADJUST adjusts not only the pair included as an argument but possibly all the pairs above in the abstract domain, since it is known that they have to be at least the value of the modified element.

5.1.2 Goal Dependencies

We now have all operations necessary to design the algorithm. However, one of our main concerns in the design of the algorithm has been the detection of redundant computations. Our algorithm includes specific data structures to maintain goal dependencies. We only introduce the basic notions here.

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.

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

- REMOVE_DP$(modified, dp)$, where $modified$ is a list of pairs $(\alpha_1, p_1), \ldots, (\alpha_n, p_n)$ and dp is a dependency graph, removes from the dependency graph all elements $\langle(\alpha, q), lt\rangle$ for which there is a $(\alpha_i, q_i))$ $\in$ $trans\_dp(\alpha, q, dp)$.
- EXT_DP(β, p, dp) inserts an element $\langle(\beta, p), \emptyset\rangle$ in dp;
- ADD_DP$(\beta, p, \alpha, q, dp)$ simply updates dp to include the dependency of (β, p) wrt (α, q) (after its execution (α, q) $\in$ $dp(\beta, p)$.

The algorithm makes sure that the elements (β, p) that need to be reconsidered are such that $(\beta, p) \notin dom(dp)$.

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

procedure solve_call(in β_in, p, suspended; inout sat, dp)
begin
   if (β_in, p) ∉ (dom(dp) ∪ suspended) then
   begin
      if (β_in, p) ∉ dom(sat) then
         sat := EXTEND(β_in, p, sat);
      repeat
         β_out := ⊥;
         EXT_DP(β_in, p, dp);
         for i := 1 to m with c_1, ..., c_m clauses-of p
         do begin
            solve_clause(β_in, p, c_i, suspended ∪ {(β_in, p)}, β_aux, sat, dp);
            β_out := UNION(β_out, β_aux)
         end;
         (sat, modified) := ADJUST(β_in, p, β_out, sat);
         dp := dp \ REMOVE_DP(modified, dp)
      until (β_in, p) ∈ dom(dp)
   end
end
```

Figure 5: The Algorithm: Part I

```
procedure solve_clause(in β_in, p, c, suspended; out β_out; inout sat, dp)
begin
   Let g_1,...,g_m body-of c in
      β_ext := EXTC(c, β_in);
      ready := {g_1};
      queue := {g_1, ..., g_m};
      while queue ≠ ∅ do
      begin
         select g_i from ready ∩ queue;
         β_aux := RESTRG(g_i, β_ext);
         switch (g_i) of
         case X_j = X_k:
            β_int := AI_VAR(β_aux)
         case X_j = f(...):
            β_int := AI_FUNC(β_aux, f)
         case q(...):
            solve_call(β_aux, q, suspended, sat, dp);
            β_int := sat(β_aux, q);
            if (β_in, p) ∈ dom(dp) then
               ADD_DP(β_in, p, β_aux, q, dp)
         end;
         if firstExecutionOf(g_i) then
            β_next := EXTG(g_i, β_ext, β_int)
         else
            β_next := REFINE(β_ext, RENAME(g_i, β_int))
         ready := ready ∪ {g_{i+1}};
         queue := queue \ {g_i} ∪ RECONSIDER({g_1, ..., g_m}, β_ext, β_next);
         β_ext := β_next;
      end;
      β_out := RESTRC(c, β_ext)
end
```

Figure 6: The Algorithm: Part II

5.2 The Reexecution Fixpoint Algorithm

The algorithm makes use of all the operations and data structures of the original algorithm. In addition, it makes use of the REFINE operation on two substitutions, of operation RENAME, and of a new operation RECONSIDER.[9] The last operation is generic but should satisfy

$$\texttt{RECONSIDER}(set, \beta_{old}, \beta_{new}) \subseteq \{\, g \in set \mid \texttt{RESTRG}(\beta_{old}, g) \neq \texttt{RESTRG}(\beta_{new}, g) \,\}.$$

The algorithm is depicted in Figures 5 and 6. Compared to the standard algorithm, the novelties are to be found in Procedure `solve_clause`. This procedure makes use of two sets: the `ready` set, which contains all atoms that can be executed (or reexecuted) and the `queue` set which contains the atoms to execute or which are thought to be interesting to reexecute. An atom to be executed needs to be present in both sets. Operation EXTG is used after the first execution while operation REFINE is used after reexecution. The `ready` set is updated to include the next goal which can be executed while the `queue` set is updated using the RECONSIDER operation.

5.3 Implementation Choices

The basic algorithm leaves open a number of decisions such as the choice of the next goal to execute, the implementation of REFINE, and the implementation of RECONSIDER. In this section, we review the choices taken in our implementation.

5.3.1 Goal Selection

Our implementation always gives priority to reexecution of goals. The underlying motivation is to provide the best possible substitution for the first execution of a goal, yielding the best possible input modes. It may also avoid calling a goal recursively with a greater (i.e. less precise) substitution. As a consequence, it tends to produce substitution preserving programs which is an important property for complexity reasons [16]. Of course, particular examples may always be found where this strategy leads to unnecessary reexecution. The experimental results however indicate the effectiveness of this heuristics.

5.3.2 REFINE

The domains used in the experimental evaluation both enjoy an exact join operation. Hence it was natural to use it as a basis for REFINE. However, other choices have also been explored to evaluate the impact of that choice.

5.3.3 RECONSIDER

Our basic implementation always reconsiders all possible goals, i.e.

$$\texttt{RECONSIDER}(set, \beta_o, \beta_n) = \{\, g \in set \mid \texttt{RESTRG}(\beta_o, g) \neq \texttt{RESTRG}(\beta_n, g) \,\}.$$

[9] RECONSIDER was considered part of the REFINE operation in the reexecution semantics.

This gives maximal precision at the cost of additional complexity. Once again, the experimental results also study the trade-off between precision and efficiency in the context of the RECONSIDER operation.

5.3.4 Widening and Termination

In the case of infinite domains, widening techniques need to be used to guarantee termination (e.g. to avoid infinite decreasing sequences). The algorithm for the basic semantics uses a simple form of widening, reported in [18], at the goal level.

In addition to the above widening technique, it is necessary to avoid, in the reexecution algorithm, an infinite sequence of reexecutions for a given clause activation (i.e. a pair (clause,substitution)). Our algorithm applies a termination test before reexecuting a goal. For each clause activation, a stack is associated with each procedure call in the clause, which contains the substitutions encountered for the procedure call during the clause activation. The termination test uses the stacks to determine whether to reexecute a goal by comparing components (e.g. the pattern component in the elaborate domain) of the goal substitution wrt the substitutions in the stack.

Note also that an infinite sequence of reexecution for a clause activation can only occur if the analysed program loops. In other words, the termination test at the clause level is not necessary for terminating programs. Hence, to increase accuracy in practice, the reexecution termination test can be omitted until there is some evidence (e.g. running out of memory) that the original program may loop.

5.4 Caching

Our algorithm uses as well the caching optimization [11] which memoize the results of all abstract operations in order to reuse them subsequently. This optimization is simpler to use than the prefix optimization (also proposed in [11]) in the reexecution context. We also believe that it is more efficient although no experimental results are available at this point.[10] Note also that the worst-case complexities for the reexecution and standard algorithms are the same with the caching optimization.

6 Experimental Evaluation

This section reports the experimental evaluation of the reexecution algorithm. Section 6.1 and 6.2 give an overview of the programs tested and of the abstract domains. Sections 6.3 and 6.4 compare the standard and reexecution algorithms on the simple and the elaborate domains respectively. Section 6.5 compares reexecution on the simple domain with standard execution on the elaborate domain. Sections 6.7 and 6.8 evaluate the impact of some of the design decisions.

[10] In worst-case complexity, caching is always better but there is a large discrepancy between the worst-case results and the average behaviour.

6.1 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 [25] and implements an alpha-beta search procedure. The program `press` is an equation-solver program taken from [25] as well. The program `cs` is a cutting-stock program taken from [26]. 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. The program `Disj` is taken from [10] 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. 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 benchmark. The program `Plan` (PL for short) 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).

6.2 Overview of the Abstract Domains

6.2.1 The Elaborate Domain

The elaborate abstract domain 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, [13, 23] and than the mode domains of, for instance, [27, 12]. It is best viewed as an abstraction of the domain of Bruynooghe and Janssens [4] where a pattern component has been added. The domain is fully described in [22] which contains also the proofs of monotonicity and consistency. Note also that the presentation in [22] 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

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

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.

6.2.2 The Simple Abstract Domain

The mode domain of [22] is a reformulation of the domain of [2]. The domain could be viewed as a simplification of the elaborate domain discussed 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.

	meq	mgt	% meq	% mgt	peq	pgt	% peq	% pgt
Append	2	1	66.67	33.33	0	1	0.00	100.00
Kalah	57	66	46.35	53.66	8	36	18.18	81.82
Queens	2	9	18.18	81.82	0	5	0.00	100.00
Press	130	13	90.91	9.09	44	8	84.62	15.38
Peep	38	25	60.32	39.68	6	13	31.58	68.42
CS	28	66	29.79	70.21	4	30	11.76	88.24
Disj	24	36	40.00	60.00	10	20	33.33	66.67
PG	8	23	25.81	74.19	0	10	0.00	100.00
Read	116	5	95.87	4.13	36	5	87.80	12.20
Plan	8	24	25.00	75.00	3	10	23.08	76.92
QSort	3	6	33.33	66.67	0	3	0.00	100.00
Mean			48.38	51.62			26.39	73.61

Table 1: Standard Versus Reexecution: Output Modes for the Simple Domain.

6.3 Simple Standard Versus Simple Reexecution

In this section, we compare the standard and reexecution algorithm on the simple domain. The simple domain may lose much precision since no information on types (e.g. the form of the terms bound to variables) is preserved. The reexecution algorithm recovers part of this information by repeating the execution of built-ins and procedure calls. It may thus improve the accuracy substantially. The purpose of this section is to quantify experimentally this gain in accuracy and its cost in performance on the simple domain.

6.3.1 Accuracy

Input and Output Modes To compare the accuracy of the standard and the reexecution algorithm, we first collect the input and output modes for each procedure in the program. Only one input and output mode is associated to each procedure (hence no program specialization is performed). Table 1 reports the results for the output modes. The first column, meq, indicates the number of modes for which the two strategies produce a similar result. The second column, mgt, indicates the number of modes for which the standard algorithm produces a greater (and thus less precise) result than the reexecution algorithm. The next two columns report the same results in percentage of the number of modes inferred. The columns peq, pgt report how many procedures have similar or greater modes in the two algorithms. In other words, peq indicates the number of procedures for which both strategies produce a similar result while pgt reports the number of procedures for which the standard algorithm loses information compared to the reexecution algorithm. The last two columns report the same measures in percentage of the number of procedures.

The experimental results show substantial gains on the modes for the reexecution algorithm on most programs. On the average, the gain is over 50%. Reexecution also improves Queens, CS, PG, and Plan by more than 70% and produces more than 30% improvement on 9 programs. Press exhibits only a 9% improvement as reexecution is not sufficient to achieve

	meq	mgt	% meq	% mgt	peq	pgt	% peq	% pgt
Append	3	0	100.00	0.00	1	0	100.00	0.00
Kalah	78	45	63.41	36.59	8	36	18.18	81.82
Queens	5	6	45.45	54.55	1	4	20.00	80.00
Press	137	6	95.80	4.20	48	4	92.31	7.69
Peep	46	17	73.02	26.98	10	9	52.63	47.37
CS	57	37	60.64	39.36	14	20	41.18	58.82
Disj	33	27	55.00	45.00	13	17	43.33	56.67
PG	18	13	58.06	41.94	4	6	40.00	60.00
Read	122	0	100	0.00	41	0	100.00	0.00
Plan	16	16	50.00	50.00	4	9	30.77	69.23
QSort	4	5	44.44	55.56	1	2	33.33	66.67
Mean			67.80	32.20			51.98	49.02

Table 2: Standard Versus Reexecution: Input Modes for the Simple Domain.

maximal precision on this program.[11] The read programs show little improvement since the results are in fact close to optimality.[12] The results at the procedure granularity are in fact higher and factor out the fact that ground input modes give ground output modes that cannot be improved upon. The average improvement here is over 70%. Reexecution also produces an improvement of over 75% for 7 programs, 4 of which show a 100% improvement.

Table 2 reports the results for the input modes. The improvements are somewhat lower than for the output modes which is intrinsic to the reexecution framework. The average improvement at the mode granularity is over 30%. It is over 49% at the procedure granularity. Note also that 7 programs are improved by more than 30% at the mode granularity while 8 programs exhibit a gain over 45% at the procedure granularity. Peaks of 81% and 80% are achieved on Kalah and Queens.

It is interesting to point out as well that the modes inferred are now optimal or close to optimal for many programs. This will be discussed in more detail in a subsequent section.

Unification Specializations To provide a more complete picture of the gain in accuracy, we also indicate how many unifications (i.e. executions of $x_i = x_j$ et $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$) can be specialized using the modes inferred by both strategies. 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. For both strategies, we report the number of static specializations (which requires a unique procedure for all uses) and the number of dynamic specializations (which assume that a procedure can be specialized in several distinct procedures). In the static case, a program uses the results of abstract interpretation to associate a substitution to each program point corresponding to the unification operation. This substitution is then used to decide whether the unification can be specialized or not. In the dynamic case, a program associates several

[11] Note that the elaborate domain with the standard algorithm does not produce optimal results either.

[12] An improvement can only come by using a domain where lists are not considered as usual functors, using for example recursive types.

	dps	ds	% ds	sps	ss	% ss
Append	8	6	75.00	4	2	50.00
Kalah	418	269	64.35	283	186	65.72
Queens	51	21	41.18	17	6	35.29
Press	1220	669	54.84	432	185	42.82
Peep	1123	828	73.73	543	393	72.38
CS	837	426	50.90	394	159	40.36
Disj	239	180	75.31	183	138	75.41
PG	116	60	51.72	53	28	52.83
Read	941	640	68.01	454	297	65.42
Plan	106	61	57.55	36	12	33.33
QSort	68	31	45.54	12	2	16.67
Mean			59.83			55.82

Table 3: Standard Algorithm: Unification Specializations for the Simple Domain

new versions with each procedure and a substitution is also attached to each program point corresponding to a unification operation. Once again, this substitution is then used to decide whether the unification can be specialized or not.

dps represents the number of possible dynamic specializations (i.e. the total number of unification operations), ds the number of specializations inferred by the analysis, %ds the percentage of dynamic specializations performed, sps the number of possible static specializations, ss the number of static specializations performed and %ss the percentage of static specializations performed.

The results are reported in Tables 3,4, and 5. In the last table, subscripts are used to differentiate the strategies. The results indicate that reexecution produces an average improvement of over 23% for the dynamic specializations and of over 25% for the static specializations. The programs with the least improvements are Press (since reexecution still loses much information) and Read (since not much information is to be gained). They are also the programs with the smallest number of specializations. The measures also show that reexecution produces 80% of dynamic specializations and over 75% of static specializations in the average, illustrating the potential for specializations in the programs.

Finally, note that in general the more accurate is the analysis, the smaller is the total number of built-ins candidate to a (dynamic) specialization. However it is not always the case as illustrated by the Press program. Reexecution on the simple domain is still not precise enough and hence it will explore new call patterns in addition to all those explored by the standard algorithm. This is a typical worst case for a reexecution algorithm.

6.3.2 Efficiency

We now investigate the computational cost of the reexecution algorithm. We first report the computation times on a SPARC-1 (Sun 4/60), the number of executions of Procedure Solve_Goal, and the number of executions of Procedure Solve_Clause. They are denoted respectively Time, Gexec, and Cexec. The results are depicted in Tables 6, 7, and 8.

The results indicate that reexecution is in the average 8% slower than the standard

	dps	ds	% ds	sps	ss	% ss
Append	8	6	75.00	4	2	50.00
Kalah	363	329	90.63	283	249	87.99
Queens	25	23	92.00	17	15	88.24
Press	1641	1059	64.53	432	203	46.99
Peep	1043	965	92.52	543	477	87.85
CS	625	514	82.24	394	289	73.35
Disj	209	189	90.43	183	163	89.07
PG	61	51	83.61	53	43	81.13
Read	941	640	68.01	454	297	65.42
Plan	56	44	78.57	36	24	66.67
QSort	24	23	95.83	12	11	91.67
Mean			83.03			75.03

Table 4: Reexecution Algorithm: Unification Specializations for the Simple Domain

	$\%ds_r - \%ds_s$	$\%ss_r - \%ss_s$
Append	0.00	0.00
Kalah	26.28	22.27
Queens	50.82	52.95
Press	9.69	4.17
Peep	18.79	15.47
CS	31.34	32.99
Disj	15.12	13.66
PG	31.89	28.30
Read	0.00	0.00
Plan	21.02	33.34
QSort	50.29	75
Mean	23.20	25.28

Table 5: Standard Versus Reexecution Algorithm: Specializations for the Mode Domain

	Time	Gexec	Cexec
Append	0.01	4	8
Kalah	1.67	91	186
Queens	0.14	23	45
Press	3.74	238	797
Peep	2.71	75	483
CS	6.43	81	284
Disj	1.89	62	122
PG	0.35	34	73
Read	3.44	118	513
Plan	0.30	46	99
QSort	0.23	26	61

Table 6: Standard Algorithm: Computation Results for the Simple Domain

	Time	Gexec	Cexec
Append	0.01	4	8
Kalah	1.88	80	160
Queens	0.07	11	21
Press	7.73	381	1229
Peep	2.92	59	438
CS	7.58	64	216
Disj	2.42	53	105
PG	0.27	20	40
Read	5.77	146	659
Plan	0.21	29	61
QSort	0.11	12	25

Table 7: Reexecution Algorithm: Computation Results for the Simple Domain

algorithm. The highest increase in computation time occurs for the Press program which is about twice as slow with reexecution. Three programs are faster using reexecution thanks to the additional precision: Queens, Plan, and PG. The number of executions at the goal and procedure level is always lower using reexecution but for the Press program (for the abovementioned reasons) and the Read program where little information can be gained.

Table 9 reports some information about the reexecution profile. It reports the percentage of pairs (goal,substitution) reexecuted on the total number of pairs called (i.e. % Goal), the percentage of atoms (i.e. built-in or procedure calls) reexecuted at least once (wrt the number of atoms that can possibly be reexecuted) (i.e. %atom) and the percentage of clauses and procedures that contain reexecuted atoms (i.e. %clause and %proc). Finally, it gives the maximum number of times an atom is reexecuted on the whole execution (i.e. maxStatic) and the maximum number of times an atom is reexecuted for a single clause activation (i.e. maxDynamic).

The results indicate that the maxDynamic is atmost 4 and in the average 2.36, indicating that few reexecutions are in fact needed. On the whole execution, an atom is atmost

	$Time_r/Time_s$	$Gexec_r/Gexec_s$	$Cexec_r/Cexec_s$
Append	1.00	1.00	1.00
Kalah	1.13	0.88	0.86
Queens	0.50	0.48	0.47
Press	2.06	1.60	1.54
Peep	1.08	0.79	0.96
CS	1.18	0.85	0.76
Disj	1.28	0.85	0.86
PG	0.77	0.59	0.55
Read	1.68	1.24	1.29
Plan	0.70	0.63	0.62
QSort	0.47	0.46	0.41
Mean	1.08	0.85	0.84

Table 8: Standard Versus Reexecution Algorithm: Computation Results for the Mode Domain

	%Goal	%proc	%clause	%atom	maxStatic	maxDynamic
Append	30.43	100.00	50.00	25.00	5	2
Kalah	10.31	33.33	26.79	6.11	4	4
Queens	10.61	57.14	30.77	11.11	3	2
Press	17.64	47.25	33.33	19.17	9	3
Peep	16.67	58.33	34.35	11.18	6	2
CS	16.67	58.33	34.71	11.18	6	2
Disj	26.78	38.24	19.4	22.18	4	3
PG	15.69	72.73	40.00	16.22	3	2
Read	12.90	51.76	36.63	14.50	4	2
Plan	17.61	47.83	33.33	17.54	5	2
QSort	17.98	83.33	50.00	22.16	5	2
Mean	17.57	58.93	35.39	16.03	4.90	2.36

Table 9: Reexecution Algorithm: Repetition Results for the Simple Domain

reexecuted 9 times and the average on all programs is close to 5. %atom also indicates that reexecution is concentrated in only about 16% of all atoms which can be reexecuted, indicating that only a small portion of the program needs reexecution.

6.3.3 Summary

Reexecution on the simple domain produces significant gains in accuracy (over 50%, 30%, 20%, 20% for the output modes, input modes, and dynamic and static specializations) at an almost negligible computational cost (8% in the average).

6.4 Elaborate Standard Versus Elaborate Reexecution

We now consider the elaborate domain. This domain produces optimal results or near-optimal results for many programs. The main exceptions are Press and Qsort where the UNION operation loses some dependencies between variables in a difference list, leading to a significant loss in accuracy. Peep has a small loss in accuracy for the same reason. Finally, Read has some imprecision due to the lack of a recursive data type for lists in the present domain (there is no hope that reexecution can overcome the problem and a simple extension of the modes considered would solve the problem [18].)

6.4.1 Accuracy

Table 10 reports the improvements for the output modes. Improvements are produced on 3 programs. Press exhibits improvements of 69.93% and 90.38% at the mode and procedure granularity, Peep produces improvements of 3.17% and 10.53%, and Qsort obtains 11.11% and 33.33%. Table 11 reports the improvement for the inputs where Press only exhibits some improvement.

Tables 12, 13 and 14 present the specialization results for both standard execution and reexecution on the elaborate domain. The only difference occurs once again on the Press program where 25% and 38% improvements are obtained.

6.4.2 Efficiency

Tables 15, 16 and 17 depicts the execution results on the elaborate domain. It is interesting to note that reexecution is, in the average, 2.31 times slower than the standard algorithm. Reexecution is two times faster than the standard algorithm for the Press program and, is the worst case, less than 5.3 times slower than the standard algorithm. The worst case occurs for program CS which builds up large terms that requires reexecution although no information can be gained. We will discuss later how this inefficiency can be removed in a natural way. Except for Press, reexecution makes always more iterations than the standard algorithm which is easily explained since not much precision is gained that could compensate for reexecution.

Table 18 presents the reexecution profile on the elaborate domain. Once again, it is clear that reexecution is focused on a small subpart of the program (10.58% of the possible atoms

	meq	mgt	% meq	% mgt	peq	pgt	% peq	% pgt
Append	3	0	100.00	0.00	1	0	100.00	0.00
Kalah	123	0	100.00	0.00	44	0	100.00	0.00
Queens	11	0	100.00	0.00	5	0	100.00	0.00
Press	43	100	30.07	69.93	5	47	9.62	90.38
Peep	61	2	96.83	3.17	17	2	89.47	10.53
CS	94	0	100.00	0.00	34	0	100.00	0.00
Disj	60	0	100.00	0.00	30	0	100.00	0.00
PG	31	0	100.00	0.00	10	0	100.00	0.00
Read	122	0	100.00	0.00	41	0	100.00	0.00
Plan	32	0	100.00	0.00	13	0	100.00	0.00
QSort	8	1	88.89	11.11	2	1	66.67	33.33
Mean			92.34	7.66			87.79	12.21

Table 10: Standard Versus Reexecution: Output Modes for the Elaborate Domain

	meq	mgt	% meq	% mgt	peq	pgt	% peq	% pgt
Append	3	0	100.00	0.00	1	0	100.00	0.00
Kalah	123	0	100.00	0.00	44	0	100.00	0.00
Queens	11	0	100.00	0.00	5	0	100.00	0.00
Press	61	82	42.66	57.34	4	40	7.69	92.31
Peep	63	0	100.00	0.00	19	0	100.00	0.00
CS	94	0	100.00	0.00	94	0	100.00	0.00
Disj	60	0	100.00	0.00	0	0	100.00	0.00
PG	31	0	100.00	0.00	31	0	100.00	0.00
Read	122	0	100.00	0.00	41	0	100.00	0.00
Plan	32	0	100.00	0.00	13	0	100.00	0.00
QSort	9	0	100.00	0.00	3	1	100.00	0.00
Mean			94.78	5.22			91.60	8.40

Table 11: Standard Versus Reexecution: Input Modes for the Elaborate Domain

	dps	ds	% ds	sps	ss	% ss
Append	4	4	100.00	4	4	100.00
Kalah	403	396	98.26	283	278	98.23
Queens	26	26	100.00	17	17	100.00
Press	2757	1808	65.58	431	249	57.64
Peep	1267	1241	97.95	543	526	96.87
CS	256	256	100.00	203	203	100.00
Disj	217	217	100.00	183	183	100.00
PG	113	100.00	88.50	52	47	90.38
Read	2701	1921	71.12	442	298	67.42
Plan	70	62	88.57	36	28	77.78
QSort	24	23	95.83	12	11	91.67
Mean			91.43			89.09

Table 12: Standard Algorithm: Unification Specializations for the Elaborate Domain

	dps	ds	% ds	sps	ss	% ss
Append	4	4	100.00	4	4	100.00
Kalah	403	396	98.26	283	278	98.23
Queens	26	26	100.00	17	17	100.00
Press	982	897	91.34	431	416	96.52
Peep	1267	1241	97.95	543	526	96.87
CS	500	500	100.00	203	203	100.00
Disj	217	217	100.00	183	183	100.00
PG	113	100.00	88.50	52	47	90.38
Read	2701	1921	71.12	442	298	67.42
Plan	70	62	88.57	36	28	77.78
QSort	24	23	95.83	12	11	91.67
Mean			93.72			92.62

Table 13: Reexecution Algorithm: Unification Specializations for the Elaborate Domain

	$\%ds_r - \%ds_s$	$\%ss_r - \%ss_s$
Press	25.76	38.88

Table 14: Standard Versus Reexecution Algorithm: Specializations for the Elaborate Domain

	Time	Gexec	Cexec
Append	0.01	3	6
Kalah	5.41	117	236
Queens	0.16	15	29
Press	24.64	552	1835
Peep	6.31	94	530
CS	5.82	85	168
Disj	3.19	68	134
PG	0.77	38	80
Read	25.08	273	1184
Plan	0.58	36	78
QSort	0.09	13	28

Table 15: Standard Algorithm: Computation Results for the Elaborate Domain

	Time	Gexec	Cexec
Append	0.01	3	6
Kalah	8.65	153	318
Queens	0.18	17	33
Press	12.12	312	1048
Peep	9.75	124	725
CS	30.83	152	554
Disj	7.43	115	230
PG	0.99	48	102
Read	123.69	819	3660
Plan	0.72	44	93
QSort	0.41	25	58

Table 16: Reexecution Algorithm: Computation Results for the Elaborate Domain

	$Time_r/Time_s$	$Gexec_r/Gexec_s$	$Cexec_r/Cexec_s$
Append	1.00	1.00	1.00
Kalah	1.60	1.30	1.35
Queens	1.12	1.13	1.27
Press	0.49	0.57	0.57
Peep	1.55	1.32	1.37
CS	5.30	1.79	3.29
Disj	2.33	1.69	1.71
PG	1.29	1.26	1.275
Read	4.93	3.00	3.09
Plan	1.24	1.20	1.19
QSort	4.56	1.92	2.07
Mean	2.31	1.47	1.65

Table 17: Standard Versus Reexecution Algorithm: Computation Results for the Elaborate Domain

	% Goal	% proc	% clause	% atom	maxStatic	maxDynamic
Append	0.00	0.00	0.00	0.00	0	0
Kalah	1.29	2.94	1.45	1.61	2	1
Queens	8.33	22.22	11.76	20.00	1	1
Press	4.27	6.45	2.34	4.51	6	2
Peep	4.50	9.76	2.51	4.31	3	1
CS	7.07	14.77	8.97	9.20	3	1
Disj	3.81	3.45	1.72	9.17	1	1
PG	2.60	7.14	3.57	4.26	1	1
Read	11.86	25.17	12.81	12.53	6	2
Plan	2.25	6.67	3.28	2.99	1	1
QSort	28.21	54.55	24.00	47.83	4	2
Mean	6.74	13.92	6.58	10.58	2.54	1.18

Table 18: Reexecution Algorithm: Repetition Results for the Elaborate Domain

	meq	mgt	mlt	% meq	% mgt	% mlt	peq	pgt	plt	% peq	% pgt	% plt
Append	3	0	0	100.00	0.00	0.00	1	0	0	100.00	0.00	0.00
Kalah	123	0	0	100.00	0.00	0.00	44	0	0	100.00	0.00	0.00
Queens	11	0	0	100.00	0.00	0.00	5	0	0	100.00	0.00	0.00
Press	142	1	0	99.30	0.70	0.00	51	1	0	98.80	1.92	0.00
Peep	61	0	2	96.83	0.00	3.17	17	0	2	89.47	0.00	10.53
CS	94	0	0	100.00	0.00	0.00	34	0	0	100.00	0.00	0.00
Disj	60	0	0	100.00	0.00	0.00	30	0	0	100.00	0.00	0.00
PG	31	0	0	100.00	0.00	0.00	10	0	0	100.00	0.00	0.00
Read	121	0	0	100.00	0.00	0.00	41	0	0	100.00	0.00	0.00
Plan	32	0	0	100.00	0.00	0.00	13	0	0	100.00	0.00	0.00
QSort	8	0	1	88.89	0.00	11.11	2	0	1	66.67	0.00	33.33
Mean				98.63	0.06	1.3				95.90	0.17	3.98

Table 19: Simple Reexecution Versus Elaborate Standard : Output modes

are reexecuted). Note also the very low figure (1.18) for maxDynamic indicating that a goal essentially needs to be reexecuted once in the average.

6.4.3 Summary

Reexecution can produce gain in accuracy and speed even for sophisticated domains as best illustrated by the Press program. Most programs relying heavily on the logical variable may in fact benefit from reexecution. The price to pay is, in the average, a slowdown of 2.31 which is reasonable although not negligible. An interesting topic we are now investigating is how complicated and efficient would be a domain including covering information to achieve the same level of precision as reexecution on the elaborate domain.

6.5 Simple Reexecution Versus Elaborate Standard

As mentioned previously reexecution on a simple domain provides an alternative to standard execution on a more sophisticated domain. In this section, we evaluate experimentally the potential of this idea, both in terms of efficiency and in terms of accuracy.

6.5.1 Accuracy

Table 19 compares the output modes obtained by reexecution on the simple domain with standard execution on the elaborate domain. The measures are the same as previously except that we also include mlt (i.e. the number of modes which are inferred more precisely by reexecution with the simple domain and reexecution than with the elaborate domain and the standard algorithm. We also include the same measure at the procedure level and the percentage versions. It indicates that over 95% of the modes are the same in both strategies. Moreover, reexecution on the simple domain improves upon the standard execution on the elaborate domain in two programs: Peep and qsort. It only loses precision on Press. For output modes, it appears that reexecution on the simple domain compares favourably with standard execution on the elaborate domain.

	meq	mgt	% meq	% mgt	peq	pgt	% peq	% pgt
Append	2	1	66.67	33.33	0	1	0.00	100.00
Kalah	116	7	94.31	5.64	37	7	84.09	15.91
Queens	9	2	81.82	18.18	3	2	60.00	40.00
Press	133	10	93.01	6.99	42	10	80.77	19.23
Peep	57	6	90.48	9.52	13	6	68.42	31.58
CS	79	15	84.04	15.96	21	13	61.76	38.34
Disj	54	6	90.00	10.00	24	6	80.00	20.00
PG	29	2	93.55	6.45	9	1	90.00	10.00
Read	114	7	94.21	5.79	33	7	82.50	17.50
Plan	29	3	90.62	9.38	10	3	76.92	23.08
QSort	9	0	100.00	0.00	3	0	100.00	0.00
Mean	92.18	7.82		71.26	29.74			

Table 20: Simple Reexecution Versus Elaborate Reexecution: Input modes

	$\%ds_{ts} - \%ds_{mr}$	$\%ss_{ts} - \%ss_{mr}$
Append	25.00	50.00
Kalah	7.63	10.24
Queens	8.00	11.76
Press	1.05	10.65
Peep	5.43	9.02
CS	17.76	26.65
Disj	9.57	10.93
PG	4.69	9.25
Read	3.11	2.00
Plan	10.00	11.10
QSort	0.00	0.00
Mean	8.39	13.8

Table 21: Simple Reexecution Versus Elaborate Standard: Unification Specializations

	$\texttt{Time}_{ts}/\texttt{Time}_{mr}$	$\texttt{Gexec}_{ts}/\texttt{Gexec}_{mr}$	$\texttt{Cexec}_{ts}/\texttt{Cexec}_{ms}$
Append	1.00	0.75	0.75
Kalah	2.88	1.46	1.47
Queens	2.29	1.36	1.38
Press	3.19	1.45	1.49
Peep	2.16	1.59	1.21
CS	0.77	1.32	0.78
Disj	1.32	1.28	1.28
PG	2.85	1.90	2.00
Read	4.35	1.87	1.80
Plan	2.76	1.24	1.28
QSort	0.81	1.08	1.12
Mean	2.22	1.39	1.32

Table 22: Simple Reexecution Versus Elaborate Standard: Computation Results

Table 20 compares the input modes inferred by both algorithms. It turns out that 92% of modes are the same in both approaches. Although the result is slightly below those obtained for the output modes, reexecution on the simple domain still compares well with standard execution on the elaborate domain.

Table 21 compares the unification specializations obtained by both approaches. It indicates that reexecution on the mode domain loses about 8% in dynamic specializations and about 14% in static specializations. This is in fact very much in accordance with the input modes and shows that reexecution bridges most of the gap between the mode and the elaborate domains.

6.6 Efficiency

Table 22 reports the comparison of execution times for both approaches. It indicates that reexecution on the simple domain is more than two times faster than the standard execution on the elaborate domain. The improvements are particularly high on the large programs such as Read, Press, and Kalah, reaching 4.35, 3.19, and 2.88. The numbers of executions of goals and procedures are also smaller but with a reduced factor (indicating that the operations on the simpler domain are less expensive).

6.6.1 Summary

Reexecution on the simple domain is certainly a valuable alternative to standard execution on the elaborate domain. As far as precision is concerned, it compares favourably for the output modes and well on the input modes and the unification specializations. Reexecution on the simple domain is also twice as fast as standard execution on the elaborate domain in the average.

	meq	mgt	% meq	% mgt	peq	pgt	% peq	% pgt
Read	37	3	92.50	7.50	115	6	95.04	4.96

Table 23: Simple Reexecution (EXTG versus $\bowtie$): Input modes

	dps	ds	% ds	sps	ss	% ss
Append	8	6	75.00	4	2	50.00
Kalah	363	324	89.25	283	244	86.21
Queens	25	23	92.00	17	15	88.24
Press	1621	1019	62.86	432	190	43.98
Peep	1037	957	92.29	543	476	87.66
CS	625	468	74.88	394	277	70.30
Disj	209	184	88.04	183	163	89.07
PG	61	51	83.61	53	43	81.13
Read	941	622	66.10	454	279	61.45
Plan	56	44	78.57	36	24	66.67
QSort	24	23	95.83	12	11	91.67
Mean			81.67			74.21

Table 24: Reexecution Algorithm With EXTG: Unification Specializations on the Simple Domain

6.7 On the REFINE Operation

As mentioned in the semantics part, there are a variety of ways to define the REFINE operations. Since both domains used in the experiment were endowed with an exact join operation, this was a natural choice for the implementation. Since other domains may not enjoy that property, we investigate its importance in the present section. The reexecution algorithms have been tested on both domains by replacing the exact join operation with EXTG.

Note that, strictly speaking, the resulting algorithm is not an instantiation of the generic algorithm since the first property of REFINE is not satisfied. Hence reexecution can actually lose precision. It is simple to make it an instance of the generic algorithm by using the definition of REFINE in terms of EXTG given at Paragraph 4.1.2. The interesting point illustrated in this section is that it is not even necessary for the domains considered.

Table 23 reports the difference on the input modes for the simple domain. Only the Read program loses some precision with this change. No change occurs for the output modes. Tables 24 and 25 report the results for the unification specializations. It appears that the loss in precision is not really significant being about 1.36% for dynamic specializations and 1.09% for static specializations.

Tables 26 and 27 compare the efficiency of the two implementations on the simple domain. Once again, the difference is not really significant and is about 1% in the average.

On the elaborate domain, there are no difference in accuracy for the measures performed in this paper. Tables 28 and 29 report the difference in efficiency which once again is negligible.

	$\%ds_{mr}$ - $\%ds_{mre}$	$\%ss_{mr}$ - $\%ss_{mre}$
Append	0.00	0.00
Kalah	1.38	1.78
Queens	0.00	0.00
Press	1.67	3.01
Peep	0.23	0.09
CS	7.36	3.05
Disj	2.39	0.00
PG	0.00	0.00
Read	1.91	3.97
Plan	0.00	0.00
QSort	0.00	0.00
Mean	1.36	0.81

Table 25: $\bowtie$ Versus EXTG: Unification Specializations on the Simple Domain

	Time	Gexec	Cexec
Append	0.01	4	8
Kalah	1.95	80	160
Queens	0.06	11	21
Press	8.17	377	1218
Peep	3.04	58	436
CS	7.51	64	216
Disj	2.4	53	105
PG	0.3	20	40
Read	5.83	146	659
Plan	0.21	29	61
QSort	0.12	12	25

Table 26: Reexecution Algorithm With EXTG: Computation Results for the Simple Domain

	$Time_{mre}/Time_{mr}$	$Gexec_{mre}/Gexec_{mr}$	$Cexec_{mre}/Cexec_{mr}$
Append	1.00	1.00	1.00
Kalah	1.03	1.00	1.00
Queens	0.86	1.00	1.00
Press	1.06	0.99	0.99
Peep	1.04	0.90	0.99
CS	0.99	1.00	1.00
Disj	0.99	1.00	1.00
PG	1.11	1.00	1.00
Read	1.01	1.00	1.00
Plan	1.00	1.00	1.00
QSort	1.09	1.00	1.00
Mean	1.02	0.99	0.99

Table 27: $\bowtie$ Versus EXTG: Computation Results on the Simple Domain

	Time	Gexec	Cexec
Append	0.01	3	6
Kalah	8.84	153	318
Queens	0.17	17	33
Press	12.10	311	1045
Peep	9.67	122	717
CS	31.90	152	554
Disj	7.47	115	230
PG	1.05	48	102
Read	128.55	819	3660
Plan	0.71	44	93
QSort	0.42	24	56

Table 28: Reexecution Algorithm With EXTG: Computation Results for the Elaborate Domain

	$\text{Time}_{tre}/\text{Time}_{tr}$	$\text{Gexec}_{tre}/\text{Gexec}_{tr}$	$\text{Cexec}_{tre}/\text{Cexec}_{tr}$
Append	1.00	1.00	1.00
Kalah	1.02	1.00	1.00
Queens	0.99	1.00	1.00
Press	0.99	0.99	0.99
Peep	0.99	0.99	0.99
CS	1.03	1.00	1.00
Disj	1.01	1.00	1.00
PG	1.06	1.00	1.00
Read	1.04	1.00	1.00
Plan	0.99	1.00	1.00
QSort	1.02	0.96	1.00
Mean	1.01	0.99	0.99

Table 29: $\bowtie$ Versus EXTG: Computation Results on the Elaborate Domain

	Time	Gexec	Cexec
Append	0.01	3	6
Kalah	7.93	146	300
Queens	0.15	15	29
Press	8.18	208	677
Peep	8.23	110	631
CS	11.51	85	308
Disj	3.35	68	134
PG	0.87	38	80
Read	123.22	793	3601
Plan	0.70	41	88
QSort	0.29	19	40

Table 30: Reexecution Algorithm With a Simplified Criteria: Computation Results for the Elaborate Domain

6.7.1 Summary

On the domains used in our experiments, it turns out that the implementation of the REFINE operation is not of primary importance. It would be interesting however to evaluate this hypothesis for significantly different domains.

6.8 On the RECONSIDER Operation

All the results reported in the previous sections assume that a goal in a clause is reconsidered each time the new substitution differs from the old one when projected on the goal variables. As mentioned previously, there is no requirement to select all components of a substitution. By selecting a subset of them, it is possible to improve efficiency at the cost of losing accuracy. In this section, we report some experiments carried out on the elaborate domain along that line. The pattern component was systematically ignored when deciding whether a substitution must be reconsidered, i.e. a goal is only reconsidered when information is gained on a component which is different from the patterns.

On the sample programs, this criteria does not lead to any loss in accuracy (it is easy to construct an artificial program where this happens). But it may substantially speed up the computation as reported in Tables 30 and 31. In particular, it produces a 3-fold speed-up on CS and more than a 2-fold speed-up on Disj. In the average, it produces over 20% improvement.

This clearly indicates that reexecution is not a single algorithm but can really be tailored to match the need of an application area.

7 Conclusion

In this paper, we have studied, in a comprehensive manner, the idea of reexecution for the abstract interpretation of Prolog. The approach is based on the recognition that, because of referential transparency, a goal may be repeated arbitrarily often without changing the

	$\texttt{Time}_{trs}/\texttt{Time}_{tr}$	$\texttt{Gexec}_{trs}/\texttt{Gexec}_{tr}$	$\texttt{Cexec}_{trs}/\texttt{Cexec}_{tr}$
Append	1.00	1.00	1.00
Kalah	0.91	0.95	0.94
Queens	0.83	0.88	0.88
Press	0.67	0.67	0.65
Peep	0.84	0.89	0.87
CS	0.37	0.56	0.58
Disj	0.45	0.59	0.58
PG	0.88	0.79	0.78
Read	0.99	0.97	0.98
Plan	0.97	0.93	0.94
QSort	0.76	0.76	0.69
Mean	0.79	0.82	0.8

Table 31: Elaborate Reexecution $\bowtie$ Versus Elaborate Reexecution EXTG: Computation Results

meaning of the program. This fact was mentioned as early as 1987 by Bruynooghe but never studied theoretically or experimentally. In this paper, a new abstract semantics embedding the idea of reexecution has been proposed. The safeness of the reexecution semantics wrt the operational semantics has been established and it has been shown that the semantics is at least as accurate as the basic semantics. The reexecution is based on a new operation REFINE whose properties have been studied and which has been related to the notions of exact join, downwards closed domains and narrowing. A fixpoint algorithm to compute the reexecution semantics has been presented and the implementation choices have been discussed, including the strategy of giving priority to reexecution over first execution. The algorithm has been tested experimentally on two domains: a simple domain (simple modes, simple sharing and same value constraints) and an elaborate domain (patterns, elaborate modes, elaborate sharing and same value constraints) and measures concerning accuracy and efficiency have been reported. They indicate that reexecution produces significant gain in precision at almost no computational cost for the simple domain. On the elaborate domain (which produces optimal or near-optimal results for almost all programs), reexecution produces gains on the non-optimal programs and is about twice as slow as the standard algorithm on the average. It was also demonstrated that reexecution on the simple domain is a valuable alternative to the standard algorithm on the elaborate domain. The accuracy of reexecution on the simple domain improves upon the standard algorithm on the elaborate domain for the output modes and comes close for the input modes and the unification specializations. Reexecution on the simple domain is also twice as fast as standard execution on the elaborate domain.

Further research will investigate the practicality of reexecution for even simpler or more complicated domains. Using reexecution for non pure programs (i.e. with no referential transparency) will also be studied. Finally we will try to define more "intelligent" (i.e. semantically driven) strategies for RECONSIDER, with the hope of achieving almost optimal results with as little overhead as possible compared to the standard algorithms. Examples such as Press (where efficiency is substantially improved) and Read (where the overhead is important) show that some work has still to be done in this respect.

Acknowledgements

This research was partly supported by the Belgian National Incentive-Program for fundamental Research in Artificial Intelligence (Baudouin Le Charlier) and 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 (Pascal Van Hentenryck).

References

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. (To Appear).

[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 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.

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

[5] Codish, M. and Falaschi, M. and Marriott, K. Suspension Analysis for Concurrent Logic Programs. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

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

[8] 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.

[9] P. Cousot and R. Cousot. Inductive Definitions, Semantics, and Abstract Interpretation. Rapport de recherche du liens, 91-14, October 1991. To appear in POPL-92.

[10] 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.

[11] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. Technical Report CS-91-67, CS Department, Brown University, December 1991.

[12] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 1991. (To appear).

[13] 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.

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

[15] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. Efficient and Accurate Algorithms for the Abstract Interpretation of Prolog Programs. Research Paper RP-90/9, University of Namur, August 1990.

[16] 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.

[17] B. Le Charlier and P. Van Hentenryck. A Universal Top-Down Algorithm for Fixpoint Computation and its Correctness Proof. Technical report, 1992. Forthcoming.

[18] 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 Fransisco, CA, April 1992.

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

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

[21] C. Mellish. *Abstract Interpretation of Prolog Programs*, volume Abstract Interpretation of Declarative Languages, pages 181–198. Ellis Horwood, 1987.

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

[23] 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.

[24] 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.

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

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

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

[28] 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.