Patrick Cousot Moreno Falaschi
Gilberto Filè Antoine Rauzy (Eds.)

Static Analysis

Third International Workshop, WSA '93
Padova, Italy, September 22-24, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Volume Editors

Patrick Cousot
DMI, Ecole Normale Superieur
45 rue d'Ulm, F-75230 Paris Cedex 05, France

Moreno Falaschi
Dipartimento di Elettronica e Informatica, University of Padova
Via Gradenigo 6/A, I-35131 Padova, Italy

Gilberto Filè
Dipartimento di Matematica Pura e Applicata, University of Padova
Via Balzoni 7, I-35131 Padova, Italy

Antoine Rauzy
Departement Informatique, IUT "A", Université Bordeaux I
F-33405 Talence, France

CR Subject Classification (1991): D.1, D.2.8, D.3.2-3, F.3.1-2, F.4.2

ISBN 3-540-57264-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57264-3Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Typesetting: Camera-ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Foreword

This volume contains the proceedings of the Third Workshop on Static Analysis (WSA'93), held in Padova (Italy) September 22-24, 1993. The previous workshops in this series, JTASPEFL and WSA'92, took place in Bordeaux (France). The aim of WSA'93 is to illustrate the use of static analysis in different programming paradigms. WSA'93 is a step towards improving contacts and promoting cross-fertilization among the numerous researchers in this area. The program committee has selected 20 papers out of the 68 submitted. These papers contribute to the following topics:

- generic algorithm for fixpoint computation
- program transformation
- strictness analysis
- static analysis techniques for logic, functional, concurrent and parallel languages and for term rewriting systems.

The workshop also includes system demonstrations and three invited lectures delivered by Pascal Van Hentenryk, Peter Van Roy and Paul Hudak. The abstracts or papers of these lectures are included in this volume.

We thank all members of the program committee and all the referees for their care in reviewing the submitted papers.

The organization of WSA'93 was supported by:

- Consiglio Nazionale delle Ricerche
- Department of Pure and Applied Mathematics, University of Padova
- University of Padova.

Finally, we express our gratitude to the members of the Organizing Committee for their enthusiastic contribution to the success of WSA'93.

July 1993 Patrick Cousot, Moreno Falaschi, Gilberto Filè, Antoine Rauzy
Co-chairpersons

Program Committee

Charles Consel (OGI)
Patrick Cousot (ENS; Chair)
Radhia Cousot (Polytechnique)
Olivier Danvy (CMU)
Bart Demoen (KUL)
Gilberto File (Padova; Co-chair)
Pascal Van Hentenryck (Brown)
Manuel Hermenegildo (UPM)
Neil Jones (DIKU)
Pierre Jouvelot (ENSMP)
Baudouin Le Charlier (Namur)
Giorgio Levi (Pisa)
Kim Marriott (Monash)
Alan Mycroft (Cambridge)
Antoine Rauzy (Bordeaux; Co-chair)
Helmut Simonis (Cosytec)

Organizing Committee

Annalisa Bossi (Padova), Michele Bugliesi (Padova), Moreno Falaschi (Padova; Chair), Giuseppe Nardiello (Padova), Sabina Rossi (Padova), Kanindа Musumbu (Bordeaux), Michel Billaud (Bordeaux), Pierre Casteran (Bordeaux), Marc-Michel Corsini (Bordeaux).

List of Referees

J.R. Abrial,
T. Amtoft,
P. H. Andersen,
N. Andersen,
L.O. Andersen,
R. Bagnara,
R. Barbuti,
P. Bazet,
M. Bellia,
L. Birkedal,
A. Bondorf,
A. Bossi,
D. Boulanger,
F. Bourdoncle,
S. Brookes,
M. Bruynooghe,
F. Bueno,
D. Cabeza,
M. Carro,
M. Chiara Meo,
M. Codish,
C. Codognet,
P. Codognet,
L. Colussi,
M. M. Corsini,
A. Cortesi,
R. Cridlig,
P. L. Curien,
D. De Schreye,
B. Demoen,
M. Denecker,
P. Deransart,
A. Deutsch,
A. Dovier,
J. C. Fernandez,
P. Ferragina,
G. Filé,
A. Filinski,
M. García de la Banda,
R. Giacobazzi,
R. Glück,
E. Goubault,
P. Granger,
K. Havelund,
T. Hospel,
J. Joergensen,
N. D. Jones,
P. Jouvelot,
P. López García,
K. Lackner Solberg,
G. Levi,
G. Longo,
P. Mancarella,
A. Marien,
T. Marlowe,
B. Martens,
F. Masdupuy,
B. Monsuez,
J. J. Moreno-Navarro,
A. Mulkers,
A. Mycroft,
F. Nielson,
C. Palamidessi,
J. Palsberg,
A. Pettorossi,
C. Queinnec,
B. Ryder,
B. Salvy,
D. Sands,
H. Søndergaard,
M. Welinder,
P. Zimmermann

Contents

Reasoning About Programs

Types

Poster Session

The Impact of Granularity in Abstract Interpretation of Prolog

Pascal Van Hentenryck[1], Olivier Degimbe[2],
Baudouin Le Charlier[2], Laurent Michel[2]

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

Abstract. Abstract interpretation of Prolog has received much attention in recent years leading to the development of many frameworks and algorithms. One reason for this proliferation comes from the fact that program analyses can be defined at various granularities, achieving a different trade-off between efficiency and precision. The purpose of this paper is to study this tradeoff experimentally. We review the most frequently proposed granularities which can be expressed as a two dimensional space parametrized by the form of the inputs and outputs. The resulting algorithms are evaluated on three abstract domains with very different functionalities, `Mode`, `Prop`, and `Pattern` to assess the impact of granularity on efficiency and accuracy. This is, to our knowledge, the first study of granularity at the algorithm level and some of the results are particularly surprising.

1 Introduction

Abstract interpretation of Prolog has attracted many researchers in recent years. This effort is motivated by the need for 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. [1, 4, 2, 7, 18, 21, 22, 25, 37]), the algorithms (e.g. [2, 5, 11, 16, 17, 18, 30]), the abstract domains (e.g. [3, 14, 27]) and the implementation (e.g. [11, 13, 18, 36]). Recent results indicate that abstract interpretation can be competitive with specialized data flow algorithms and could be integrated in industrial compilers.

As can be seen from the above references, abstract interpretation of Prolog has led to the development of many frameworks and algorithms. One of the reasons for this proliferation is the fact that program analysis can be defined ar various granularities achieving specific tradeoffs between accuracy and efficiency.[3] The granularity of an algorithm is influenced by numerous parameters, including the choice of program points and the form of the results (e.g. how many output abstract substitutions are related to each program point). In fact, combinations of these two parameters cover most existing algorithms.

The first parameter, `program point`, concerns the number of abstract objects considered per procedure. At least, three possibilities (`single, call, dynamic`)

[3] Note that the tradeoff between efficiency and accuracy can be studied at the abstract domain level as well, as for instance in [10].

have been investigated and they differ in the way different call patterns for a procedure are dealt with. `single` associates a unique abstract object with each procedure as, for instance, in the algorithm of Taylor [31, 32]. As a consequence, different call patterns are merged together within this granularity. Mellish [25] also associates a unique abstract object with each procedure. Contrary to Taylor however, the abstract object is a set of abstract substitutions and not a single substitution. `call` associates an abstract object with each procedure call in the *text of the program*, as in the framework of Nilsson [28, 29].[4] Different call patterns for a procedure are not merged but rather distributed among the procedure calls. Of course, different call patterns are merged inside each program point. `Dynamic` associates an abstract object with each pair (β, p) in the program, where β is an abstract substitution and p is a predicate symbol. This granularity is adopted in many frameworks (e.g. [2, 4, 11, 13, 17, 18, 22, 23, 37, 38]) and keeps different call patterns separate. It is interesting to note that, for the first two granularities, it is possible to generate a priori a finite set of equations whose variables represent the abstract substitutions adorning the program points. This is not possible for the third granularity whose semantics defines a functional equation. However, this equation can be approximated by a dynamic set of finite equations. As a consequence, it is more difficult to produce an algorithm for `dynamic` since the static analyzer must combine the fixpoint computation with the generation of the equations.

The second parameter, `abstract result`, concerns the form of the result stored at each program point. At least two possibilities (`single`, `multiple`) have been proposed and differ in the way they handle the results of the clauses to produce the result of a procedure. `single` stores a single result per program by using, an upper bound operation on the clause results. This granularity is used in many frameworks and algorithms (see all the above references). `multiple` stores a set of results per program point by collecting the results of all clauses and possibly applying a filter (e.g. a subsumption test). This granularity is used in the frameworks based on `OLDT`-resolution (e.g. [5, 10, 12, 16, 33, 35]).

The two parameters, when combined, produce a two-dimensional design space depicted in Figure 1. Other granularities exist. For instance, the `single` and `call` entries can be doubled by allowing set of abstract objects for the forms of the inputs. These granularities are related to `OLDT`-based abstract interpretation but are not studied here.

The purpose of this paper is to study experimentally this two dimensional space. The experimental results are given for a variety of benchmarks and for three abstract domains: `mode`, a domain containing same-value, sharing, and mode components [26], `pattern`, a domain containing same-value, sharing, mode, and pattern components [26, 18], and `Prop`, a domain using Boolean formulas to compute groundness information [8, 20, 24].

The rest of this paper is organized in the following way. Section 2 reviews informally the various granularities considered in this paper. Section 3 presents the experimental results. Section 4 contains the conclusion of this research. Most of the

[4] In the presentation of Nilsson, program points are associated with clause entry, clause exit, and any point between the literals in the clause. As discussed later in the paper, this is equivalent to adorning each procedure call in the text of the program with an input and an output substitution.

input / output	single	call	dynamic
single	AISISO	AICISO	AIDYSO
multiple	AISIMO	AICIMO	AIDYMO

Table 1. The Design Space of Granularities

results given here are described in detail in two technical reports [34, 33].

2 The Granularities

In this section, we give an informal overview of the various granularities. We assume that the frameworks use abstract substitutions to represent sets of concrete substitutions and that *Abs* is an abstract domain of this type (e.g. a pointed cpo with an upper bound operation). We use *Pred* and *Call* to denote the set of predicate symbols and the set of procedure calls in the text of the program. Abstract substitutions are denoted by β (generally subscripted), predicates by the letter p, and procedure calls by the letter C.

2.1 Dynamic/Single

This granularity is probably the most popular in the logic programming community and corresponds to what is called a *polyvariant* analysis in the functional programming community. It is used for instance in [4, 2, 13, 17, 18, 11, 22, 23, 38, 37].

The key idea is to associate with each predicate symbol p multiple abstract tuples of the form $(\beta_{in}, p, \beta_{out})$. More precisely, the result of the analysis is a partial function of signature $Pred \rightarrow Abs \rightarrow Abs$ which, given a predicate symbol p and an input abstract substitution β_{in}, returns a result β_{out} satisfying the following informal condition:

> "the execution of $p(x_1, \ldots, x_n)\theta$, where θ is a substitution satisfying the property expressed by β_{in}, produces substitutions $\theta_1, \ldots, \theta_n$, all of which satisfy the property expressed by β_{out}."

The main features of this granularity are as follows:

- The abstract semantics at this granularity define a functional and cannot be reduced to a finite set of equations. As a consequence, the fixpoint algorithm needs to interleave the generation of the equations and their solving.
- Since the semantics preserve multiple input patterns, it can be used to implement advanced program transformations such as multiple specializations [37] which associates multiple versions to each procedure (possibly one for each input patterns).

GAIA [18], which is the basis of the experimental work described later on, is a top-down algorithm working at this granularity. It can be viewed as an instance of a general fixpoint algorithm [19] or, alternatively, as an implementation of Bruynooghe's

framework [2]. The algorithm is query-directed, providing an algorithmic counterpart to the notion of minimal function graph [15]. It also includes many optimizations such as caching of the operations [11] and a dependency graph to avoid redundant computations. Finally, in the case of infinite domains, the algorithm uses a widening operator to ensure the finiteness of the analysis for domains satisfying the ascending chain condition. Another closely related algorithm is PLAI [13]. The algorithm at this granularity is referred to as AIDYSO in the following.

2.2 Single/Single

Single/Single is the coarsest granularity studied in this paper and corresponds to what is called a *univariant* analysis in the functional programming community. Taylor's algorithm [31, 32] is an example of analyzer working at this granularity.

The key idea here is to associate with each predicate in the program a unique pair $\langle \beta_{in}, \beta_{out} \rangle$, where β_{in} (resp. β_{out}) is an abstract substitution representing the properties of the concrete input (resp. output) substitutions of p. More precisely, the result of the analysis is a partial function of signature $Pred \rightarrow Abs \times Abs$. The result $\langle \beta_{in}, \beta_{out} \rangle$ of the analysis for a predicate symbol p can be read informally as follows:

> "p is executed in the analyzed program with input substitutions satisfying β_{in} and produces answer substitutions satisfying β_{out}".

The loss of efficiency compared to AIDYSO occurs because input patterns from different procedure calls may be merged together resulting in a less precise input pattern for analyzing the procedure.

The main features of this granularity are as follows:

- The granularity collapses all the input patterns into a single input substitution. As a consequence, it produces the coarsest granularity studied in this paper. We expected this granularity to give rise to the fastest algorithm.
- The granularity precludes certain types of program transformations such as multiple specializations.
- The abstract semantics defined at this granularity can be expressed as a finite set of equations and the fixpoint algorithm does not need widening operators when the abstract domain satisfies the ascending chain property.

The fixpoint algorithm AISISO for this granularity can be deduced from AIDYSO by computing before the execution of a procedure an upper bound on the memoized input abstract substitution to be refined and a new input abstract substitution under consideration. The upper bound is used both as the new memoized abstract substitution and to continue the analysis.

2.3 Call/Single

The granularity Call/Single was proposed by Ulf Nilsson [28, 29] and is intermediary between the previous two granularities.

Its key idea is to associate with each *procedure call* a pair of abstract substitutions $\langle \beta_{in}, \beta_{out} \rangle$. More precisely, the fixpoint algorithm computes a partial function $Call \rightarrow Abs \times Abs$ which, given a procedure call C, returns a pair $\langle \beta_{in}, \beta_{out} \rangle$ whose informal semantics is described as follows:

> "during the program execution, the substitutions encountered before the execution of a procedure call C satisfy the property expressed by β_{in} while the substitutions encountered after the execution of the call satisfies the property expressed by β_{out}."

Although it seems to be fundamentally different from the previous two, this granularity can be reexpressed in the same framework by considering simply that the function computed is of signature $Pred \rightarrow Call \rightarrow Abs \times Abs$. Viewing it this way, it becomes clear that the granularity is intermediary between `Single/Single` and `Dynamic/Single`. Instead of collapsing all input patterns into a single input, `Call/Single` distributes them among a finite number of procedure calls. The gain in precision compared to `Single/Single` comes from the fact that different procedure calls do not interfere with each other. The loss of precision compared to `Dynamic/Single` comes from the merging of abstract substitutions for a given procedure call.

The key features of this granularity are as follows:

- The granularity is coarser than `Dynamic/Single` and finer than `Single/Single`. We expected the algorithm to be faster than `AIDYSO` and slower than `AISISO`.
- The granularity allows for multiple specializations although their full potential may not be realized because of the merging.
- The semantics defined at this level can be reduced to a finite set of equations.

Once again, the algorithm for this granularity `AICISO` can be obtained from `AIDYSO` by computing upper bound operations appropriately. The key insight, mentioned earlier, is to associate with each predicate symbol p as many pairs as there are program points corresponding to procedure calls to p.

It is also interesting to note that a finer granularity can be obtained from *Dynamic/Single* and *Call/Single* by associating multiple pairs $\langle \beta_{in}, \beta_{out} \rangle$ to a procedure call. This results in an analysis returning a partial function of signature $Call \rightarrow Abs \rightarrow Abs$. This granularity is not explored here for reasons that will appear clearly in the experimental results.

2.4 Dynamic/Multiple

`Dynamic/Multiple` is another popular granularity in the logic programming community. It was used for instance in [5, 10, 12, 16, 33, 35]. The main reason is that the algorithm for this granularity can be obtained automatically by applying `OLDT`-resolution to an abstract version of the program as shown in [6, 35]. This is due to the interesting termination properties of `OLDT`-resolution.

The key idea here is to associate with each predicate symbol p in the program multiple abstract tuples of the form $\langle \beta_{in}, S_{out} \rangle$, where S_{out} is a set of abstract substitutions (i.e. $S_{out} \in 2^{Abs}$). More precisely, the result of the analysis is a partial function of signature $Pred \rightarrow Abs \rightarrow 2^{Abs}$ which, given a predicate symbol p and an input abstract substitution β_{in}, returns a set S_{out} whose informal semantics is given by:

> "the execution of $p(x_1, \ldots, x_n)\theta$, where θ is a substitution satisfying the property expressed by β_{in}, will produce substitutions $\theta_1, \ldots, \theta_n$, all of which satisfy the property expressed by some β_{out} in S_{out}"

In general, for efficiency reasons, it is important to add some more structure on 2^{Abs} to eliminate redundant elements from the output sets (i.e. the elements β' such there exists another element β satisfying $\beta' \leq \beta$). The relational powerdomain (i.e. Hoare powerdomain) can be used instead of the powerset for that purpose.

This granularity is the most precise studied in this paper. The gain in accuracy compared to `Dynamic/Single` comes from the multiple outputs which give rise to more precise input patterns, especially when the abstract domain maintains structural information.

The key features of this granularity are as follows:

- It is the finest granularity defined in this paper and is obviously appropriate for multiple specializations [37].
- The abstract semantics at this granularity define a functional transformation.

The algorithm `GAIA` can be generalized to work at this granularity but the task is non-trivial, since each procedure call gives rise to multiple clause suffix and special care should be given to avoid redundant work. In [33], we report how optimizations such as the suffix optimization, caching, and output subsumption are important to achieve a reasonable efficiency. With this optimization, the resulting algorithm `AIDYMO` spends over 90% of its time in the abstract operations.

Another point to stress is that a new widening operator is necessary to make sure that an output cannot be refined infinitely often in case of infinite abstract domains. This new widening is used when a new output, say β, is about to be inserted in an output set, say S. Instead of inserting β, the algorithm inserts $\beta \nabla S$ for a given widening operator ∇. There are a variety of possible widening operators, some of them being domain-dependent and others being domain-independent. In our experiments, we use the operators ∇_d. The operator is domain-dependent, is defined on the domain `Pattern` to be discussed later, and relates to the depth-k abstraction sometimes used in abstract interpretation. Informally speaking, ∇_d widens the new substitution by taking its lub with all the substitutions having the same outermost functors (depth-1). Since there are finitely many function symbols in a program, the output set is guaranteed to be finite.

2.5 Single/Multiple

`Single/Multiple` is an hybrid between `Single/Single` and `Dynamic/Multiple`. It is close to the early proposal of Mellish [25], the only difference being that the single input in Mellish is also a set of abstract substitution. This granularity will thus give us an idea on how appropriate this early proposal was.

The key idea is to associate with each predicate symbol p in the program a single abstract tuple (β_{in}, S_{out}).

The key features of this granularity are as follows:

- The abstract semantics can be reduced to a finite set of equations.
- The granularity is coarser than `Dynamic/Multiple` and finer than `Single/Single`. It is difficult to compare to the other granularities proposed earlier. The granularity is not appropriate for multiple specialization.

The algorithm `AIDYMO` can be specialized to produce an algorithm `AISIMO` for this granularity, once again by taking appropriate upper bound operations.

3 Experimental Results

We now turn to the experimental results. We start with a brief description of the abstract domains before considering the experimental results for efficiency and accuracy. The Prolog programs used in the experiments are described in previous papers (e.g. [11]) and are available by anonymous ftp from Brown University.

3.1 Abstract Domains

The Domain `Mode`*:* The domain `Mode` of [26] is a reformulation of the domain of [2]. The domain could be viewed as a simplification of the domain `Pattern` described below, 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 terms 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.

The Domain `Prop`*:* In `Prop` [24, 8, 20], a set of concrete substitutions over $D = \{x_1, \ldots, x_n\}$ is represented by a Boolean function using variables from D, that is an element of $(D \rightarrow Bool) \rightarrow Bool$, where $Bool = \{false, true\}$. `Prop` only considers Boolean functions that can be represented by propositional formulas using variables from D, the truth values, and the logical connectives $\vee, \wedge, \Leftrightarrow$. $x_1 \wedge x_2$ and $x_1 \Leftrightarrow x_2 \wedge x_3$ are such formulas. The basic intuition behind the domain `Prop` is that a substitution θ is abstracted by a Boolean function f over D iff, for all instances θ' of θ, the truth assignment I defined by $I(x_i) =$ true iff θ' grounds x_i $(1 \leq i \leq n)$ satisfies f.

The Domain `Pattern`*:* The abstract domain `Pattern` contains patterns (i.e. for each subterm, the main functor and a reference to its arguments are stored), sharing, same-value, and mode components. It should be related to the ***depth-k abstraction*** of [16], but no bound is imposed a priori to the terms depth. Since the domain is infinite, widening operations must be used by many of the algorithms. The domain is fully described in [26, 18] and reference [26] contains also the proofs of monotonicity and safeness. This is an infinite domain and the experimental results are reported with a simple widening technique which applies an upper bound operation on each recursive call.

3.2 Efficiency

This section reports our experimental efficiency results on a variety of domains. For lack of space, we only report a summary of the results, the full tables being available in the technical reports associated with this paper.

Table 2 reports the efficiency results for the algorithms on the domain `Mode`. We give the ratios between the cpu times of the algorithms wrt `AIDYSO` and the absolute time in seconds of `AIDYSO` on a Sun Sparc 10/30. There are two important results in this table:

1. The first and more surprising result is that `AICISO` is in fact 13% slower than `AIDYSO`, indicating that a coarser granularity does not imply necessarily a better

	Ratio on AIDYSO			Time
	AISISO	AICISO	AIDYMO	AIDYSO
cs	0.89	1.03	1.01	1.44
cs1	0.85	1.01	1.03	1.18
disj	0.99	1.14	1.00	0.74
disj1	0.93	1.05	1.01	0.81
gabriel	0.71	0.89	1.03	0.35
kalah	0.74	0.83	1.02	1.21
peep	0.90	1.14	1.71	1.11
pg	0.76	0.82	1.07	0.17
plan	1.00	1.09	1.45	0.11
press1	0.63	1.14	1.07	1.53
press2	0.65	1.14	1.07	1.55
qsort	1.00	1.00	2.00	0.01
queens	1.00	1.00	1.50	0.01
read	0.71	2.51	1.31	1.40
Mean	0.84	1.13	1.15	

Table 2. Ratios on the Efficiency Results on Domain Mode

efficiency. This negative result can be attributed to the fact that some redundant computations occur because the same results are stored twice in different program points. This forces AICISO to perform many more iterations and, although most of the redundancy is removed by the caching optimization, the loss in efficiency is still important.

2. The second result is that the algorithms are really close. AISISO gains about 16% over AIDYSO on this domain, while AIDYSO is 1.15 times faster than AIDYMO.

Table 3 depicts the efficiency results for the domain Pattern. The table also contains some interesting results.

1. The most surprising result is the significant loss of efficiency incurred by AIDYMO which is more than 17 times slower than AIDYSO in the average. For some large programs (e.g. program disj1), AIDYMO is about 54 times slower than AIDYSO. The analysis time is in the worst case (i.e. program press1) about 2 minutes. The main reason for this poor result is the fact that domain Pattern is both more precise and richer than domain Mode. In particular, the pattern component forces AIDYMO to maintain outputs whose modes are similar but which have different functors as is typical in analyzing recursive programs. This can lead to additional precision as we will see but it also entails some duplicated effort as indicated by the efficiency results.
2. The second surprising result is that AISIMO is significantly slower than AIDYMO indicating once again that a coarser granularity does not necessarily mean a better efficiency.
3. The remaining results confirm some of the previous results on the domain Mode. They indicate that AISISO brings an improvement of 29% over AIDYSO in the average while AICISO is about twice as slow as AIDYSO, confirming the relatively poor results of AICISO. It is important to stress the impact of the widening

	Ratio on AIDYSO				Time
	AISISO	AICISO	AIDYMO	AISIMO	AIDYSO
cs	0.87	0.98	7.17	8.30	2.13
cs1	0.87	0.96	6.94	7.96	2.18
disj	0.88	1.68	45.21	55.05	1.19
disj1	0.89	1.62	54.64	56.56	1.22
gabriel	0.39	0.81	3.95	3.86	0.90
kalah	0.59	0.75	4.72	5.46	2.97
peep	0.66	1.02	28.98	19.38	2.36
pg	0.34	0.51	2.10	2.31	0.71
plan	0.73	1.23	2.00	2.33	0.22
press1	0.32	1.01	13.41	31.69	8.70
press2	1.08	3.45	10.38	101.81	2.60
qsort	1.00	11.00	56.00	48.00	0.01
queens	1.00	2.00	12.00	12.00	0.01
read	0.29	1.27	6.86	10.42	5.53
Mean	0.71	2.02	17.52	22.14	

Table 3. Ratios on the Efficiency Results on Domain Pattern

techniques on AIDYSO. The fact that widening is implemented through a general upper bound operation explains why AISISO and AIDYSO are rather close. The main difference between the two algorithms on recursive calls is that AIDYSO keeps distinct tuples when it takes the upper bound on two recursive calls while AISISO merges them. The need to update the various tuples should explain the small difference in efficiency and iterations between the two algorithms. On the other hand, keeping distinct versions can lead to important differences in accuracy for non-recursive calls with very different input patterns (i.e. multi-directional procedures).

Table 4 depicts the efficiency results for the domain Prop. The results indicate that AISISO brings an improvement of 5% over AIDYSO in the average while AICISO is about 1.57 as slow as AIDYSO. The gain of AISISO over AIDYSO is rather small in this case. The best improvement occurs for program Press1 (29%) but most programs show little or no improvement. AICISO is the slowest program and is about 5 times slower on Read.

3.3 Accuracy

To evaluate the accuracy of the various algorithms, we use the number of unification specializations made possible by the modes inferred by the algorithms. We consider that $x_i = x_j$ (i.e. AI_VAR) can be specialized when one of its arguments is either ground or variable and that $x_{i_1} = f(x_{i_2}, \ldots, x_{i_n})$ (i.e. AI_FUNC) can be specialized when its first argument is either ground or a variable. Once again, we report the results for all domains. In measuring unification specializations, we assume that there is only one version of each procedure (i.e. no multiple specialization), since AISISO and AICISO do not support several versions. The measure is of course unfair to AIDYSO but helps understanding the tradeoff between efficiency and accuracy.

	Ratio on AIDYSO		Time
	AISISO	AICISO	AIDYSO
cs	1.01	1.14	1.58
cs1	1.00	1.12	1.60
disj	1.01	1.10	1.27
disj1	1.08	1.15	1.21
gabriel	0.95	1.53	0.58
kalah	1.00	1.23	1.00
peep	1.08	1.66	1.43
pg	0.95	1.20	0.20
plan	0.93	1.07	0.14
press1	0.71	1.69	6.76
press2	0.75	1.74	6.69
qsort	1.00	1.00	0.01
queens	1.00	1.00	0.01
read	0.77	5.31	2.12
Mean	0.95	1.57	

Table 4. Ratios on the Efficiency Results on Domain Prop

		AIDYSO		AISISO		AICISO		AIDYMO		AISIMO	
	P	E	$\%E/P$	E	$\%E/P$	E	$\%E/P$	E	$\%E/P$	E	$\%E/P$
peep	543	527	97.05	527	97.05	527	97.05	538	99.07	526	96.87
press1	434	259	59.68	258	59.45	258	59.45	259	59.68	258	59.45
press2	435	421	96.78	259	59.54	259	59.54	421	96.78	259	58.57
read	405	299	73.83	274	67.65	274	67.65	299	73.83	274	67.65
Mean			90.18		87.07		87.07		90.96		87.07

Table 5. Accuracy Results on Domain Pattern

For the domain Mode, all specialization results are the same for all algorithms.

Table 5 depicts the specialization results for which there is a difference between the programs and the average over all programs. We report the number of possible specializations (P), the number of effective specializations deduced from the analysis (E), and the ratio (E/P) in percentage for the three program. The results for Pattern indicate that AISISO, AICISO, AISIMO lose precision on three programs compared to AIDYSO: press1, press2, and read. These are also the programs for which AISISO produces more significant efficiency improvements. AIDYMO produces a small improvement over AIDYSO on peep but this is rather marginal in the overall accuracy results. The good accuracy of AISISO on the above two domains can be explained by two important features of our algorithms and domains: operation EXTG and the same-value component of domains Mode and Pattern. Operation EXTG performs a form of narrowing [9] at the return of the procedure call. Hence much of the accuracy lost in the upper bound operation of the procedure call is recovered through operation EXTG. The same-value component contributes to the precision recovered by providing EXTG with strong relations on the variables. For instance, it is possible that a call pattern (ground,any) returns (any,any) in AISISO. However, if

the result also concludes that the two arguments are equal thanks to the same-value component, operation EXTG will concludes that both arguments are ground achieving a form of narrowing operation.

For the domain Prop, no difference in accuracy is exhibited by the algorithms. This interesting result can be explained by the fact that the upper bound operation (implemented by applying the LUB operation of the domain) does not lose accuracy in Prop. It is thus equivalent to consider several input patterns or a single one which is the LUB of the encountered patterns. Any domain with this property is probably worth investigating and AISISO is clearly the most appropriate algorithm in this case. Note also that Prop is not particularly appropriate for specialization, since only groundness and not freeness is computed.

In summary, AIDYSO improves the accuracy of AISISO and AICISO on a certain number of programs for the domain Pattern and the domain Mode with reexecution. The improvement occurs for the larger programs in general and correlates well with the programs where AIDYSO spends more time than AISISO. AIDYMO improves slightly over AIDYSO on a single program.

4 Conclusion

Abstract interpretation of Prolog has received much attention in recent years leading to the development of many frameworks and algorithms. One reason for this proliferation comes from the fact that program analyses can be defined at various granularities, achieving a different trade-off between efficiency and precision. The purpose of this paper is to study this tradeoff by considering various granularities for the program points. Three algorithms have been considered and extended with reexecution. The first three algorithms have been evaluated on three abstract domains, Mode, Prop, and Pattern with respect to accuracy and efficiency, while the reexecution algorithms have been studied on the domain Mode.

The experimental results lead to several conclusions.

- AISISO is in general the fastest analyzer but it may lose some precision for programs using the multidirectionality of logic programming. AISISO seems best on domains which enjoy an exact LUB operation, since it seems faster and as accurate as AIDYSO.
- AICISO seems not to be very interesting in practice. It is slower than AIDYSO in the average although it works at a coarser granularity. The difference in efficiency is not dramatic but there is no reason to choose AICISO over AIDYSO.
- The algorithms AIDYMO and AISIMO seems to work at a too fine granularity. They incur a substantial loss without really improving the accuracy.
- The differences in accuracy between these algorithms on our benchmarks were rather small.

It is tempting for us to argue that AIDYSO can be considered as a "best-buy" since

1. its loss in efficiency compared to AISISO is rather small on our domains;
2. it is more accurate than AISISO on arbitrary domains and this difference would show up more clearly on benchmarks exploiting the multi-directionality of logic programs which was not really the case of our benchmark programs;