Experiments in Context-Sensitive Analysis of Modular Programs

Jesús Correas[1], Germán Puebla[2], Manuel V. Hermenegildo[2,3], and Francisco Bueno[2]

[1] School of Computer Science
Complutense University of Madrid (UCM)
[2] School of Computer Science
Technical University of Madrid (UPM)
[3] Depts. of Computer Science and Electrical and Computer Engineering
University of New Mexico (UNM)

jcorreas@fdi.ucm.es
{german,herme,bueno}@fi.upm.es

Abstract. Several models for context-sensitive analysis of modular programs have been proposed, each with different characteristics and representing different trade-offs. The advantage of these context-sensitive analyses is that they provide information which is potentially more accurate than that provided by context-free analyses. Such information can then be applied to validating/debugging the program and/or to specializing the program in order to obtain important performance improvements. Some very preliminary experimental results have also been reported for some of these models which provided initial evidence on their potential. However, further experimentation, which is needed in order to understand the many issues left open and to show that the proposed modes scale and are usable in the context of large, real-life modular programs, was left as future work. The aim of this paper is two-fold. On one hand we provide an empirical comparison of the different models proposed in previous work, as well as experimental data on the different choices left open in those designs. On the other hand we explore the scalability of these models by using larger modular programs as benchmarks. The results have been obtained from a realistic implementation of the models, integrated in a production-quality compiler (CiaoPP/Ciao). Our experimental results shed light on the practical implications of the different design choices and of the models themselves. We also show that context-sensitive analysis of modular programs is indeed feasible in practice, and that in certain critical cases it provides better performance results than those achievable by analyzing the whole program at once, specially in terms of memory consumption and when reanalyzing after making changes to a program, as is often the case during program development.

1 Introduction and Motivation

Global analysis of logic programs has received considerable theoretical and practical attention and as a result it is now possible to infer a wide range of program

properties with a considerable degree of accuracy and for a significant number of programs. Also, tools have been developed which in addition to inferring these properties, allow debugging, validating, and specializing programs, achieving important improvements in both correctness and efficiency. However, most of these techniques were originally designed to be applied to a complete, monolithic program. In contrast, real programs invariably have a more complex structure combining a number of user modules with other modules from system libraries. This is one of the reasons why most global analysis tools are still prototypes and, though numerous experiments demonstrate their effectiveness, they have not yet made their way into existing real-life programming systems.

Performing global analysis on modular programs differs from doing so in a monolithic setting in several interesting ways and poses non-trivial problems which must be solved (see, for example, [5] and its references where the main approaches to separate modular static analysis by abstract interpretation are described). Regarding the analysis of modular LP programs, a preliminary study of the extension of context-sensitive analysis and specialization to the case of modular logic programs was presented in [12]. A full practical proposal for context-sensitive analysis of modular logic programs was presented in [3]. In fact, in [3] a collection of models was proposed, each of them with different characteristics and representing different trade-offs. Some very preliminary experimental data was also reported for an implementation of some of these models in the context of the Ciao system. Also, another implementation of [3] in the context of the HAL system [7] was reported in [9]. These previous preliminary experimental results provided initial evidence on the overall potential of the approach, but were limited in that they studied only a partial implementation. It was left as future work to perform further experimentation in order to understand the many issues and trade-offs left open in the design and to show that the proposed models scale and are usable in the context of large, real-life modular programs.

The aim of this paper is two-fold. On one hand we provide an empirical comparison of the different models proposed in [3], as well as experimental data on the different choices left open in those designs. To this end we have completed a full implementation in CiaoPP of the framework for context-sensitive analysis described in [11] and its different instances and we have studied experimentally the behavior of the resulting system. These results have been compared with traditional, non modular analyses in several parameters.

Our second aim is to explore the scalability of these models and of the implementation. To this end we have used some larger modular programs as benchmarks, including some real-life examples such as a working partial evaluator and parts of the Ciao compiler.

In the following section we present an overview of the general problems in analyzing large modular programs, and the solutions proposed in previous work, including the major design trade-offs. Section 3 then describes the tests performed and analyzes the results obtained. Finally, Section 4 presents our conclusions.

2 Analysis of modular programs

As mentioned in the previous section, the framework used herein is based on [12, 11], where a detailed description of the issues related to the analysis of modular programs and the different approaches to it can be found. The following subsections present an overall summary of [11], with special emphasis on the issues that are most relevant to our experimental study.

2.1 Modular programs

A program is said to be modular when its source code is distributed in several source units named modules, and they contain language constructions to clearly define the interface of every module with the rest of the modules in the program. This interface is composed of two sets of predicates: the set of exported predicates (those accessible from other modules), and the set of imported predicates. For concreteness, and because of its appropriateness for global analysis, in our implementation we will use the module system of [4]. This module system is *strict* in the sense that procedures external to a module are visible to it only if they are part of its *interface*. A predicate defined in a given module can be called from another module only if it appears in the exported list of its module and in the imported list of the caller module, i.e., procedures which are not exported are not visible outside the module in which they are defined.

We note the distinction between *global* tasks and *local* tasks. In global tasks the results of processing a part of the program (say, a procedure or a module) may be needed in order to process other parts of the program. In contrast, a local task processes only one procedure or module at a time and, most importantly, all the information required for performing the task can be obtained by inspecting that procedure or module. The fundamental issue is that global processing often requires iterating on the whole program until a fixed-point is reached.

Context-sensitive program analysis is an example of a *global* task: in a modular setting, it may well be the case that part of the information needed to perform the analysis on (a procedure in) module m has to be computed in modules other than m. We will refer to the information originated in modules different from m as *inter-modular* information in contrast to the information originated in m itself, which we will call *intra-modular*.

2.2 Flattening a Program Unit vs. Modular Processing

Applying a framework for non-modular programs to a module m which belongs to a modular program has the difficulty that m may not be self-contained. However, there should be no problem in applying the framework if m is a leaf module. Furthermore, given a global process such as program analysis, at least in principle, it is not obvious that it makes much sense to apply the process to a module m alone. In fact, it makes sense to apply analysis to the complete program instead, since it is conceptually self-contained.

Given a modular program P it is always possible to build a single module m_{flat} which is equivalent to P and which is a leaf. The process of constructing such a module m_{flat} usually only amounts to renaming apart identifiers in the different modules in P so as to avoid name clashes. We will use $\mathit{flatten}(P) = m_{flat}$ to denote that the module m_{flat} is the result of renaming apart the code in each module in P and concatenating its code into a monolithic module m_{flat}. This points to a simple solution to the problem of processing modular programs (at least for the case in which all the code is available): to transform P into the equivalent monolithic program m_{flat}. It is then straightforward to apply any tool for non-modular programs to the leaf module m_{flat}. In the rest of this work, we will refer to this approach as the *flattened* or *monolithic* approach.

Assuming the existence of an implementation for non-modular analysis, this approach to analyzing modular programs is often simple to apply. Also, the flattening approach has theoretical interest: in our case it will be used to compare the efficiency of different approaches to modular handling of programs w.r.t. it. However, as a practical way in which to actually perform analysis of large programs the flattening approach also has important potential drawbacks. The most important is that the complete program must be loaded into the analyzer, and thus large programs may make the analyzer run out of memory. Moreover, as the internal analysis data structures include information for all the program source code, in the monolithic case, analysis of a given procedure may take more time than keeping in memory only the module in which it resides. Another, perhaps more important drawback, is that the program must be self-contained: this can be a problem if the analyzer is used while developing the program, when some modules are not yet implemented, or if there are calls to external procedures, i.e., procedures for which the source code is not available, or which are implemented in other languages.[1]

2.3 Analyzing one module at a time

The approach taken in [11] and implemented in CiaoPP is based on the separate analysis of the modules in a modular program. The analyzer is invoked (possibly several times) for each module in the program, in order to obtain the analysis results needed by the analysis of other program modules. We denote the process of obtaining the answer value AP of any predicate P for a call CP as: $P : CP \mapsto AP$. The analysis results obtained for the exported predicates of every module are stored in a *Global Answer Table* (GAT).

Analyzing a module separately presents the difficulty that, from the point of view of analysis, the code to be analyzed is *incomplete* in the sense that the code for procedures imported from other modules is not available to analysis. More precisely, during the analysis of a module m there may be calls $P : CP$ such that the procedure P is not defined in m but instead it is imported from another module m'. We refer to determining the answer value of P, AP ($P : CP \mapsto AP$) as

[1] Several approaches have been proposed for the analysis of incomplete programs (*open programs*), for example [2, 1].

the *imported success problem*. In addition, in order to obtain analysis information for m' which is as accurate as possible we need to somehow propagate the call $P : CP$ from m to m' so that the next time m' is analyzed such a call pattern is taken into account. We refer to this as the *imported calls problem*.

Solving the Imported Success Problem The imported success problem is solved by means of a *success policy*, or *SP* for short. The behavior of the analyzer for predicates defined in m remains exactly as before. *SP* is needed because given a call pattern $P : CP$ it will often be the case that an entry of exactly the form $P : CP \mapsto AP$ does not exist in the analysis results stored in the GAT for m'. In such case, the information already present may be of value in order to obtain a (temporary) answer pattern AP, and continue the analysis of module m.

In contrast, in many formalizations of non-modular analysis there is no explicit success policy. This is because if the call pattern $P : CP$ has not been analyzed yet, the analysis algorithm forces its computation. Thus, the results of analysis do not depend on any particular success policy: when the analyzer reaches a fixed-point there is always an entry of the form $P : CP \mapsto AP$ for any call pattern $P : CP$ which appears in the analysis graph. However, in a modular setting it is often convenient to delay the analysis of predicates defined in other modules until those modules are revisited. In general, those modules may have already been analyzed or they may be analyzed in the future. We will simply do the best possible given the information available in the GAT.

Several success policies can be defined which provide over- or under-approximations of the exact answer pattern $AP^{=}$ with different degree of accuracy. Note that this exact value $AP^{=}$ is the one which the flattening approach (that we will thus denote $SP^{=}$) would compute. In this work we consider two kinds of success policies, those which are guaranteed to always provide over-approximations, i.e. $AP^{=} \sqsubseteq SP(P : CP, GAT)$, and those which provide under-approximations, i.e., $SP(P : CP, GAT) \sqsubseteq AP^{=}$. We will use the superscript $^{+}$ (resp. $^{-}$) to indicate that a success policy over-approximates (resp. under-approximates).

In the experiments shown in this work, a very precise over-approximating success policy has been used, already proposed in [12] and defined as:

$$SP^{+}_{All}(P : CP, GAT) = topmost(CP) \sqcap_{AP' \in app} AP' \text{ where}$$
$$app = \{AP' \mid (P : CP' \mapsto AP') \in GAT \text{ and } CP \sqsubseteq CP'\}$$

The function *topmost* obtains the topmost answer pattern for a call pattern. The notion of *topmost description* was already introduced in [2]. Informally, a topmost description preserves the information on properties which are *downwards closed* whereas it loses information for those which are not. Note that taking $\top$ as answer pattern is also a correct over-approximation, but often less accurate than using topmost substitutions. For example, if a variable is known to be ground in the call pattern, it will continue being ground in the answer pattern and taking $\top$ as the answer pattern would lose this information. However, the fact that a variable is free on call does not guarantee that it will keep on being free on success.

We refer to this success policy as SP^+_{all} because it uses *all* entries in GAT which are *applicable* to the call pattern in the sense that the call pattern already computed is more general than the call being analyzed.

The counter-part of SP^+_{all} is the function SP^-_{all} which is defined as:

$$SP^-_{All}(P : CP, GAT) = \sqcup_{AP' \in app} AP' \text{ where}$$
$$app = \{AP' \mid (P : CP' \mapsto AP') \in GAT \text{ and } CP' \sqsubseteq CP\}$$

Note the change in the direction of the applicability relation (the call pattern in the GAT has to be more particular than the one being analyzed) and the use of the lub operator instead of the glb. Also, note that taking, for example, $\perp$ as an under-approximation is correct but SP^-_{all} is more precise.

As shown in [11] using SP^+ policies has the advantage that at any point during the modular analysis, even when a fixpoint has not been reached yet, the information obtained for each module is always a correct over-approximation. The drawback is that when the fixpoint is reached it may not be minimal, i.e., information is not as precise as it could be. In contrast, SP^- policies obtain the least fixpoint (most precise information) but only produce correct results when the fixpoint it reached. SP^+ policies can be useful during program development.

Solving the Imported Calls Problem As the analysis is context-sensitive, the call patterns for imported predicates are only known after the calling module is analyzed, but they cannot be processed until the imported module is selected for (re)analysis. These call patterns are therefore stored in another global data structure, the *temporary answer table* (TAT for short).[2] When the imported module is scheduled for (re)analysis, all call patterns in the TAT are used as input for the analyzer.

2.4 Computing an intermodular fixed point

The intermodular fixed-point algorithm of CiaoPP takes one module of the program that needs (re)analysis, analyzes it storing the relevant information in GAT and TAT tables, and looks for another module which needs reanalysis. When a module is analyzed, it updates the entries in the global tables, and marks the modules which import it if the analysis results may improve the results of those modules. An intermodular fixed point has been reached when there are no modules which need reanalysis.

Determining the optimal order in which the different modules in the program unit should be analyzed in order to get to a fixed-point as efficiently as possible is not trivial. Finding good scheduling strategies for intra-modular analysis is a topic which has received considerable attention and highly optimized algorithms

[2] In fact, GAT and TAT are implemented using the same table, and TAT entries are marked as needing reanalysis, in order to provide more precise results than those obtained applying the success policy, as soon as the module is scheduled for (re)analysis. There are more details in Section 2.4 and [11].

exist which converge to a fixed-point quickly. Unfortunately, it is not possible to directly translate the same heuristics used in the intra-modular case to the inter-modular case. In the inter-modular case we have to take into account the time required to change from analysis of one module to another since this typically means reading a new module from disk. Thus, requests to process call patterns have to be grouped by modules in order to reduce the number of times we change context.

In the current implementation, two simple strategies have been used, in order to study the behavior of the analysis of modular programs in clearly different scenarios. Both strategies take the list of modules in a given order (a top-down and a bottom-up traversal of the intermodule dependency graph, respectively),[3] and traverse the list analyzing the modules which have pending call patterns, updating the corresponding global tables with the analysis results. This process is repeated until there are no pending call patterns for any module in the program.

We will refer to this intermodular fixed-point algorithm, scheduling one module at a time for analysis as the *modular approach*.

3 Empirical results

The CiaoPP implementation of the framework summarized above has been tested by parameterizing it in several ways, in order to study the overall behavior of the system. Different trade-offs and characteristics of the analysis of modular programs have been studied:

Flattened vs. modular First, the flattened approach of Section 2.2 has been compared to the intermodular fixpoint of Section 2.4. Although it is predictable that the analysis of a program for the first time in a modular, separate analysis fashion will be slower than the flattened approach (due to the overhead in loading/unloading modules, etc.), it is interesting to study by how much. On the other hand, in some cases the analysis of a whole program may be unfeasible due to hardware (memory) limitations, but in the intermodular fixpoint approach this limitation can be overcome.

Intermodular scheduling policies Another aspect to study is related to the influence of the module selection policy in the efficiency of the analysis. The scheduling policies used have been already described in Section 2.4. We will refer to them as *naive_top_down* and *naive_bottom_up*, respectively.

Success policies Two success policies have been compared in both scheduling policies: an over-approximating policy, SP^{+}_{all}, and an under-approximating one, SP^{-}_{all}, as described in Section 2.3. Although there may be other success policies, we estimate that these ones are the most effective policies, as they bring the closest results to $SP^{=}$.

Incremental analysis of modular programs Finally, the analysis of a modular program from scratch using the monolithic approach has been compared

[3] All modules which belong to the same cycle in the graph have been considered at the same depth, and therefore those modules will be selected in any order.

to the reanalysis of that program after making specific modifications in the source code. This comparison illustrates the advantages of analyzing only the module which has changed (and the modules affected by that change) instead of reanalyzing the whole program from scratch.
Three different kinds of source code modifications have been studied: 1) a simple change that keeps the same analysis results, 2) a change that results in the exported predicates producing a more precise answer pattern, and 3) a modification in the source code such that after the change exported predicates produce more general analysis results.
Note that when there are changes in the source code which do not improve or invalidate previous analysis results, nor generate new call patterns for imported modules, i.e., 1) there are clear advantages in using the modular approach, since only one module must be analyzed at a time. In contrast, in the monolithic, non-modular analysis the whole program must be analyzed. Also note that this kind of changes may occur more often if assertions are used on a regular basis, as they can bring very precise answer patterns, similar to the results provided during the analysis.
The second kind of change studied represents a change that makes the analysis results for exported predicates be more precise than the ones obtained before. This is done by removing all clauses of exported predicates of a module except the first non recursive one.[4] This will bring in general analysis results which are more specific than the results previously obtained, making them invalid in most cases, and producing the reanalysis of the calling modules.
The third type of source change corresponds to performing a modification in an exported predicate which results in this predicate providing more general analysis results. The change consists in the addition of a clause to all the exported predicates of a module in which all arguments are pairwise distinct and free variables.[5] This approach then forces the reanalysis of the modules which call the changed module.

In the following subsections the selected benchmark programs are described, and the results of the tests are studied in detail. Two modes domains have been considered: Def [6], a simplified version of the Pos domain, and *Sharing-freeness* [8], which gets combined information on variable sharing and freeness.

3.1 Brief description of the benchmarks used

The central focus of this paper is to show how the intermodular analysis framework of CiaoPP behaves with real-life programs. Therefore, the selection of

[4] Mutually recursive predicates are also considered. If the exported predicate has only recursive clauses, they are replaced by a fact with all arguments ground.

[5] In the $Sharing - Freeness$ domain this addition might not provide a more general analysis result, as this kind of clause does not provide a top success substitution. However, the tests have been performed using the same change also in the case of $Sharing - Freeness$ to make the tests homogeneous across the different domains.

benchmark programs must include not only characteristic examples used in the LP analysis literature, but also other programs which are specially difficult to analyze in a modular setting (for example, because there are several mutually recursive predicates which conform intermodular cycles), and real-life programs. A brief description of the selected benchmarks follows:

ann This is the &-Prolog implementation of the MEL annotator (by K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo). In this case the code is distributed in 3 modules with no cycles in the intermodular dependency graph.

bid This program computes an opening bid for a bridge hand (by J. Conery). It is composed of 7 modules, with no cycles in the intermodular dependency graph.

boyer The boyer benchmark is a reduced version of the Boyer/Moore theorem prover (by E. Tick). The program has been separated in four modules with a cycle between two modules.

peephole This program is the SB-Prolog peephole optimizer. In this case, the program is split in three modules, but there are two cycles in the intermodular dependency graph, and there are several intermodular cycles at the predicate call level.

prolog_read corresponds to a simplified version of the code used by the Ciao compiler for reading terms. It is composed by three modules, having a cycle between two of them.

unfold_ is a fragment of the CiaoPP preprocessor which contains the partial evaluator. It is distributed in 7 modules with no cycles between them, although many other modules of CiaoPP source code, while not analyzed, are consulted in order to get assertion information.

managing_project is a program used by the authors for EU project management. It is distributed in 8 modules with no intermodular cycles.

check_links is an example program for the *Pillow* HTML/XML/HTTP connectivity package (by D. Cabeza and M. Hermenegildo) that checks that links contained in a given URL address are reachable. The whole Pillow package is analyzed together with the sample program, and it is composed of 6 modules without intermodular cycles.

It should be noted that for all these programs the number of modules indicated above correspond to the user modules of the benchmark. However, they are not the only ones processed: any benchmark is likely to use quite a large number of modules from the system libraries. In particular, in Ciao all builtins are in system libraries. For efficiency, library modules are pre-analyzed for a representative set of call patterns and the analysis results are expressed using the assertion language described in [10]. Instead of analysing library modules over and over again, the analysis algorithm computes success information from such assertions using a SP^+ policy.

Performance-wise, in the current implementation we have first focused on optimizing analysis times. Loading times have not been optimized as much, but

we are confident that these times can be reduced further by storing assertions for libraries in a precompiled format.

The benchmarks have been run on a Dell PowerEdge 4600 with two Pentium processors at 2 Ghz and 4 Gb of memory, and normal workload. Each test has been run twice, reporting the arithmetic mean of these runs.

3.2 Analysis of a modular program from scratch

Table 1 shows the absolute times in milliseconds spent in analyzing the programs using the flattening approach. **Mod** reflects the number of modules comprising each benchmark (excluding system modules). For every benchmark, the total analysis time is divided into several categories, represented by the following columns:

Load This column corresponds to the time spent loading modules into Ciaopp. This time includes the time used for reading the module to be analyzed and the time spent in reading the assertions of the imported modules.

Ana. This is the time spent analyzing the program and applying the success policy for imported predicates together with some preprocessing of the code.

Gen. Corresponds to the task of generating the global information (referred to before as the GAT and TAT tables). The information generated is related to the analysis results of all exported and multifile predicates, new call patterns of imported predicates generated during the analysis of each module, and the modules that import the module and can improve their analysis results by reanalysis.

Total Time elapsed since the analyzer is called until it finishes completely. It is the sum of the previous columns, plus some extra time spent in other tasks, such as the generation of the intermodular dependency graph, handling the list of modules to get the next module to be analyzed, etc.

As we have said above, loading times in Table 1 comprise not only the time spent in loading the user modules which compose each benchmark, but also that of loading the subset of the libraries which are needed for that particular benchmark, and the selection of the relevant assertions from those libray modules. Optimization of loading time for library modules is subject of ongoing work.

Tables 2 and 3 give the summary of the weighted arithmetic and geometric means of the comparative times for the analysis domains Def and *Sharing-freeness* respectively. The numbers in these tables are relative to the monolithic case (shown in Table 1), and correspond to the weighted mean, using the number of clauses of each program as weight for each benchmark. The *naive_bottom_up* and *naive_top_down* global scheduling policies are compared, as well as the SP^{-}_{all} and SP^{+}_{all} success policies. Table columns have the same meaning as before.

The rows labeled "**From scratch**" in these tables show the overall time spent in the analysis of the different benchmarks without previous analysis information. In Table 2 the intermodular analysis from scratch using Def is only somewhat slower compared to the monolithic analysis, and in particular the

Def					
Bench	**Mod**	**Load**	**Ana.**	**Gen.**	**Total**
ann	3	873	319	140	1518
bid	8	1182	31	136	1645
boyer	4	1076	138	81	1470
peephole	3	1685	313	231	2533
prolog_read	3	829	360	304	1668
unfold_	7	3325	1357	394	5506
managing_project	8	1626	9369	496	11892
check_links	6	1792	4757	1249	8235
Sharing-freeness					
Bench	**Mod**	**Load**	**Ana.**	**Gen.**	**Total**
ann	3	873	496	196	1743
bid	8	1182	38	146	1647
boyer	4	1076	197	89	1533
peephole	3	1685	562	309	2831
prolog_read	3	829	2778	570	4340
unfold_	7	3325	548143	635	552514
managing_project	8	1626	824	390	3273
check_links	6	1792	6080	1340	9623

Table 1. Time spent (in milliseconds) by the monolithic analysis of different benchmark programs

analysis time is not much larger than the monolithic time in most cases. However, in simple domains like Def, the analysis time is not the most important fraction of the total time, and therefore other tasks such as module loading or results generation can in fact be more relevant than the analysis itself. On the other hand, more complex domains as $Sharing-freeness$ (Table 3) increase the difference with respect to the monolithic case. It is important to note that using SP^{+}_{all} is clearly not recommended for performing modular analysis from scratch in the $Sharing-freeness$ domain. The result in this case is biased a great deal by the results of the analysis of `managing_project`, in which most predicates have many arguments, resulting in large sharing sets that tend to approximate to $\top$ (which is the powerset of the variables in the clause).

On the other hand, when comparing the global scheduling policies, only a slight difference in the time taken using the *naive_top_down* or the *naive_bottom_up* strategies can be observed. This result seems to reflect that the order of the modules is not so relevant when analyzing a modular program as was initially expected.

Memory Consumption when analyzing from scratch. We have also compared the maximum memory required for the analysis in the flattened and the modular approaches to the analysis of modular programs from scratch. Table 4 shows the maximum memory consumption during the analysis of the flattened approach

***Def* domain** Global scheduling policy: naive_top_down								
	automatic SP^+				**automatic SP^-**			
Type of test	**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
From scratch - ar.mean	1.72	1.33	0.89	1.21	2.48	1.70	1.55	1.59
From scratch - geo.mean	1.70	1.04	0.77	1.15	2.44	1.32	1.37	1.49
Mod. touch - ar.mean	0.75	0.37	0.18	0.58	0.74	0.38	0.18	0.58
Mod. touch - geo.mean	0.74	0.35	0.15	0.54	0.73	0.36	0.14	0.54
Less general chg. - ar.mean	1.10	0.39	0.34	0.69	1.13	0.38	0.37	0.69
Less general chg. - geo.mean	1.08	0.15	0.26	0.55	1.11	0.14	0.26	0.54
More general chg. - ar.mean	1.10	0.58	0.39	0.76	1.13	0.60	0.41	0.77
More general chg. - geo.mean	1.08	0.49	0.30	0.69	1.11	0.50	0.30	0.70
Global scheduling policy: naive_bottom_up								
	automatic SP^+				**automatic SP^-**			
Type of test	**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
From scratch - ar.mean	1.71	1.28	0.87	1.19	2.49	1.70	1.55	1.58
From scratch - geo.mean	1.69	1.00	0.75	1.14	2.45	1.30	1.36	1.47
Mod. touch - ar.mean	0.76	0.36	0.17	0.59	0.77	0.38	0.17	0.60
Mod. touch - geo.mean	0.74	0.35	0.13	0.54	0.76	0.36	0.14	0.55
Less general chg. - ar.mean	1.08	0.30	0.28	0.64	1.12	0.34	0.33	0.68
Less general chg. - geo.mean	1.06	0.12	0.22	0.51	1.10	0.14	0.24	0.54
More general chg. - ar.mean	1.06	0.51	0.31	0.72	1.11	0.56	0.37	0.76
More general chg. - geo.mean	1.04	0.45	0.25	0.66	1.08	0.48	0.28	0.69

Table 2. Arithmetic and geometric overall results for analysis of modular programs using different global scheduling algorithms and success policies in the *Def* domain.

(column **Monolithic**), and the use of memory of the modular approach (using both global scheduling policies described before) relative to the monolithic case (columns SP^+_{all} and SP^-_{all} for the corresponding success policies). The results show that the modular approach is clearly better in terms of maximum memory consumption than the monolithic approach, except for the outlying result of `managing_project`, as mentioned above, and two of the benchmarks containing intermodular cycles at the predicate level, `peephole` and `prolog_read`. However, given a program split into N modules, the memory used for analysing it in a modular way might be expected to be M/N, where M is the memory consumed in a monolithic analysis. This is not true because the complexity of the program is in general not evenly distributed among its modules. Since Table 4 shows maximum memory consumption, figures are strongly influenced by the most complex modules.

3.3 Reanalysis of a modular program after a change in the code

As explained at the beginning of Section 3, we have also studied the incremental cost of reanalysis of a modular program after a change, for different typical changes, as explained above. In the first case ("**Mod. touch**" rows), a simple

***Sharing-freeness* domain**								
Global scheduling policy: naive_top_down								
	automatic SP^+				**automatic SP^-**			
Type of test	**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
From scratch - ar.mean	1.77	202.33	1.32	52.24	2.43	2.51	1.58	2.31
From scratch - geo.mean	1.75	16.08	1.14	8.67	2.37	1.99	1.53	2.13
Mod. touch - ar.mean	0.78	0.48	0.20	0.66	0.75	0.30	0.20	0.61
Mod. touch - geo.mean	0.77	0.35	0.19	0.50	0.75	0.21	0.19	0.46
Less general chg. - ar.mean	1.07	0.30	0.33	0.73	1.00	0.31	0.30	0.69
Less general chg. - geo.mean	1.06	0.17	0.28	0.48	0.97	0.10	0.22	0.45
More general chg. - ar.mean	1.21	0.96	0.52	1.02	1.23	0.75	0.49	0.96
More general chg. - geo.mean	1.19	0.64	0.45	0.72	1.20	0.49	0.44	0.69
Global scheduling policy: naive_bottom_up								
	automatic SP^+				**automatic SP^-**			
Type of test	**Load**	**Ana.**	**Gen.**	**Total**	**Load**	**Ana.**	**Gen.**	**Total**
From scratch - ar.mean	1.78	200.99	1.26	51.91	2.53	2.53	1.59	2.35
From scratch - geo.mean	1.76	16.01	1.07	8.64	2.47	2.05	1.53	2.18
Mod. touch - ar.mean	0.77	0.46	0.19	0.66	0.78	0.30	0.18	0.62
Mod. touch - geo.mean	0.77	0.35	0.18	0.50	0.77	0.21	0.17	0.47
Less general chg. - ar.mean	1.02	0.28	0.27	0.70	0.97	0.30	0.26	0.68
Less general chg. - geo.mean	1.00	0.15	0.24	0.46	0.94	0.09	0.20	0.43
More general chg. - ar.mean	1.21	0.87	0.47	0.97	1.21	0.74	0.48	0.95
More general chg. - geo.mean	1.19	0.60	0.40	0.71	1.19	0.50	0.42	0.69

Table 3. Arithmetic and geometric overall results for analysis of modular programs using different global scheduling algorithms and success policies in the *Sharing-freeness* domain.

change in a module with no implications in the analysis results of that module has been tested. It has been implemented by "touching" a module, i.e., changing the modification time without actually modifying its contents, in order to force CiaoPP to reanalyze it. "**Less general change**" rows correspond to a source code modification in which all the clauses of the exported predicates of a given module have been replaced by the first non-recursive clause of the predicate. And finally, the third case ("**More general change**" rows) is implemented by adding a most general fact to all exported predicates of a given module.

For every benchmark, all these source code modifications have been made for each module, and the weighted arithmetic and geometric means of the resulting reanalysis times have been considered. The weight has been measured as the number of clauses of the module that has been modified.

The overall results in Tables 2 and 3 indicate that in many cases the reanalysis time is better than in the monolithic case. It is important to note that the analysis domain used is very relevant to the efficiency of the modular approach: the analysis of a complete program in complex domains such as $Sharing - freeness$ is much more expensive than the reanalysis of a module, while the difference is smaller (although still significant) in the case of Def. This suggests

Global scheduling policy: naive_top_down							
		Def			***Sharing-Freeness***		
Bench	**Mod**	**Monolithic**	SP^+	SP^-	**Monolithic**	SP^+	SP^-
ann	3	3739488	0.91	0.93	4121592	0.92	0.94
bid	8	3462054	0.78	0.72	3456770	0.80	0.74
boyer	4	3424928	0.90	0.80	3579152	0.93	0.87
peephole	3	5278446	0.87	0.95	5738422	0.92	1.25
prolog_read	3	3779344	0.92	0.95	5515176	1.28	1.43
unfold_	7	8697250	0.95	0.83	15862826	0.64	0.63
managing_project	8	6998418	0.63	0.69	6044194	4.22	0.82
check_links	6	13535418	0.76	0.72	20727590	0.90	0.89
Weighted Arithm. mean			0.77	0.77		2.10	0.87
Weighted Geom. mean			0.76	0.76		1.54	0.86
Global scheduling policy: naive_bottom_up							
		Def			***Sharing-Freeness***		
Bench	**Mod**	**Monolithic**	SP^+	SP^-	**Monolithic**	SP^+	SP^-
ann	3	3739488	0.91	0.91	4121592	0.91	0.92
bid	8	3462054	0.78	0.72	3456770	0.80	0.73
boyer	4	3424928	0.90	0.80	3579152	0.93	0.93
peephole	3	5278446	0.87	0.95	5738422	0.92	1.25
prolog_read	3	3779344	0.92	0.95	5515176	1.28	1.43
unfold_	7	8697250	0.95	0.83	15862826	0.64	0.63
managing_project	8	6998418	0.64	0.67	6044194	4.22	0.82
check_links	6	13535418	0.74	0.72	20727590	0.87	0.87
Weighted Arithm. mean			0.77	0.76		2.09	0.87
Weighted Geom. mean			0.76	0.75		1.53	0.86

Table 4. Overall memory consumption of Non-modular vs. SP^+ and SP^- policies.

that modular analysis can make it practical to use domains which are precise but rather costly. On the other hand, the results in Table 3 for reanalysing after a more general change are very close to monolithic analysis from scratch, although still below it. That means that even in the presence of the most agressive change in a module, modular analysis is not more time-consuming than analyzing from scratch. Simpler changes provide better results of the modular analysis with respect to the flattened approach, as is shown in Tables 2 and 3 for other kinds of changes.

4 Conclusions

We have provided an empirical study of several proposed models for context-sensitive analysis of modular programs with the objective of providing experimental evidence on the scalability of these models and, specially, on the impact on performance of the different choices left open in those models.

Our results shed some light on the different choices available. In the case of analyzing a modular program from scratch, the modular analysis approach is expected slower than the flattening approach (i.e., having the complete program in memory, and analyzing it as a whole), due to the impact of load and unload code and related analysis information, and the restriction of not being able to analyze predicates in other modules when a different one is being processed. Also, this suggests future work on reducing the time spent in loading/unloading modules and storing analysis results. However, the modular analysis times from scratch are still reasonable, excluding the case of the $Sharing-freeness$ domain with SP^{+}_{all} success policy. On the other hand, it does imply a lower maximum memory consumption which in some cases may be of advantage since it may allow analyzing programs of a certain critical size that would not fit in memory using the flattening approach.

Across the domains we can see that in simple domains SP^{+}_{all} and a naive bottom up scheduling policy appears to be the best. It is substantially better for some experiments (in particular, for more general changes) and not much worse on most experiments. Another conclusion which can be derived from our experiments is that, as already mentioned, no important difference has been observed between the top-down and bottom-up strategies.

We have also considered the case of reanalyzing a previously analyzed program, after making changes to it. This is relevant because this is the standard situation during program development, in which some modules change while others (and the libraries) remain unchanged. While in this phase the analysis results may not be needed in order to obtain highly optimized programs, they are used for static program validation and debugging. In this context the modular analysis, because of its more incremental nature, shows advantages in both time and memory consumption over the monolithic approach in some cases.

Acknowledgements

The authors would like to thank María García de la Banda, Kim Marriott, and Peter Stuckey for many interesting discussions on analysis of modular programs. This work was funded in part by projects ASAP (EU IST FET Programme Project Number IST-2001-38059), CUBICO (MCYT TIC 2002-0055), FEDER infrastructure UNPM-E012, and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. Part of this work was performed during a research stay of Germán Puebla at Roskilde University supported by a grant from the Secretaría de Estado de Educación y Universidades.

References

1. F. Besson and T. Jensen. Modular class analysis with datalog. In *10th International Symposium on Static Analysis, SAS 2003*, number 2694 in LNCS. Springer, 2003.
2. F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

3. F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.
4. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.
5. P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Eleventh International Conference on Compiler Construction, CC 2002*, number 2304 in LNCS, pages 159–178. Springer, 2002.
6. V. Dumortier, G. Janssens, W. Simoens, and M. García de la Banda. Combining a Definiteness and a Freeness Abstraction for CLP Languages. In *Workshop on Logic Program Synthesis and Transformation*, 1993.
7. María J. García de la Banda, Bart Demoen, Kim Marriott, and Peter J. Stuckey. To the Gates of HAL: A HAL Tutorial. In *International Symposium on Functional and Logic Programming*, pages 47–66, 2002.
8. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
9. Nicholas Nethercote. The Analysis System of HAL. Master's thesis, Monash University, 2002.
10. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.
11. G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.
12. G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.