A Model for Inter-module Analysis and Optimizing Compilation

Francisco Bueno[1], María García de la Banda[2], Manuel Hermenegildo[1], Kim Marriott[2], Germán Puebla[1], and Peter J. Stuckey[3]

[1] Technical University of Madrid (UPM), Spain
{bueno,herme,german}@fi.upm.es
[2] Monash University, Australia
{mbanda,marriott}@csse.monash.edu.au
[3] University of Melbourne, Australia
pjs@cs.mu.oz.au

Abstract. Recent research into the implementation of logic programming languages has demonstrated that global program analysis can be used to speed up execution by an order of magnitude. However, currently such global program analysis requires the program to be analysed as a whole: separate compilation of modules is not supported. We describe and empirically evaluate a simple model for extending global program analysis to support separate compilation of modules. Importantly, our model supports context-sensitive program analysis and multi-variant specialization of procedures in the modules.

1 Introduction

Decades of software development have demonstrated that the use of modules to structure programs is crucial to the development of large software systems. It has also shown that separate compilation of these modules is vital since it is too inefficient to have to compile the entire program including library files each time something is changed. Thus, virtually all commercially used compilers for almost all programming languages allow compilation of modules in isolation. Typically this is achieved by import and export declarations in the module providing information about each module's interface to other modules.

Recent research into the implementation of logic programming languages has demonstrated that information from global program analysis can be used, for example, to guide compile-time optimizations which can speed up execution by an order of magnitude [8] or to guide automatic parallelization [3]. In order to perform such optimizations, program analyses must determine useful information at compile-time about the run-time behavior of the program. However, a severe limitation of most existing analyses is that they require the program to be analysed as a whole: thus separate compilation of modules is not supported.

One of the main reasons to need the whole program is that, at least for logic programs, accurate program analysis seems to require *context-sensitive* analysis in which procedures are analysed with respect to a number of calling patterns, rather than a *context-free* analysis in which analysis is only performed once for each procedure. Context-sensitive analysis is also important because it naturally supports specialization of the program procedures for particular cases. Such *multi-variant specialization* [19] allows different optimizations to be performed on the same procedure, a technique which has been shown to be quite important in practice [8, 10].

Here we describe and empirically evaluate a rather simple model for extending abstract interpretation based global program analysers for logic programming languages to support separate module compilation. Our model supports a program development scenario in which the program is divided into modules and each module is developed separately, possibly by different programmers. Each module may be compiled separately, taking into account the available analysis information of the modules it imports for optimization. Each compilation updates available analysis information as well as marking other modules where recompilation could improve performance. The whole process migrates towards a point equivalent to that obtained by analysing and optimizing the whole program at once. Importantly, the model supports context-sensitive program analysis and multi-variant specialization of procedures in the modules.

The issue of how to combine global program analysis with separate compilation of modules is not new: however, most work has been limited to context-free program analysis. In this context, `make` style dependencies [6] can be used to order compilation of modules. If a module is modified, other modules depending upon it must be recompiled if the changes may have affected them. Thus, one area of research has been to determine precisely which changes to a module force recompilation of another module [13, 14, 16]. These approaches do not extend naturally to context-sensitive analyses.

Another proposal consists in a two stage approach to program analysis, in which a fast imprecise global analysis is applied to the whole program, then more precise analysis (possibly context sensitive) is applied to each module in turn [12]. Similarly, the work described in [4] proposes an approach to analysis of Prolog modules in which the system first pre-analyses modules, and then analyses the program as a whole by combining the result of these pre-analyses. This method is basically restricted to context-free analyses. These approaches contrast to our approach in which no analysis of the whole program is ever required.

In the specific context of logic programming languages, Mercury [15] is a modular logic programming language which makes use of sophisticated modular analyses to improve execution speed, but context sensitive analysis information is required to be specified by the user for exported predicates.

The present paper extends our previous work on the incremental analysis of logic programs [7] (although this work did not consider modules as such) and also [11] which contains a general discussion of different scenarios and the overall issues which appear in the analysis and specialization of programs decomposed into modules. The paper is organized as follows. In the next section we introduce our notation. Then, Section 3 gives our model for modular compilation where modules are not changing. Section 4 extends this to the edit-compile-test cycle. Some brief experimental results are provided in Section 5, while Section 6 concludes.

2 Preliminaries and notation

We assume the reader is familiar with the concepts of abstract interpretation [5] which underlie most analyses of logic programs.

The context-sensitive analysis of a module starts from descriptions of the initial calls to the module. The analysis computes descriptions of all the possible calls to the procedures in the module and, for each possible call, a description of its possible answers (returns of the calls). Descriptions of procedure calls and answers are "abstract" values in a domain $\mathcal{D}$ of descriptions, which is a poset with ordering relation $\preceq$. The infimum and supremum of the poset are denoted $\perp$ and $\top$, respectively, while the operations of least upper bound and greatest lower bound are denoted $\sqcup$ and $\sqcap$, respectively.

A *call pattern* is a pair $P : CP$ of a procedure P and a description CP of the values of the arguments of P (in logic programming terms, the head variables) when it is called. We assume for simplicity that all procedures P appear with a unique set of arguments (head variables), so we can refer to the argument positions in the description. An *answer pattern* is a triple $P : CP \mapsto AP$ where AP is a description of the values of the argument positions of P on return of calls described by call pattern $P : CP$. Analysis of a module M proceeds from a call pattern $P : CP$ for a procedure P exported by M and computes a set of answer patterns for all procedures that are visible in M and are reachable from $P : CP$. The description $\perp$ indicates an unreachable situation (e.g. an unreachable program point) while the description $\top$ conveys no information. The answer patterns computed by analysis are kept in the *answer table* of module M. There will be only one answer pattern per call pattern.

A *call dependency* is an arc $P : CP \to P' : CP'$ between call patterns which indicates that calls of the form $P' : CP'$ (might) occur during execution of calls of the form $P : CP$. The analysis also computes all possible call dependencies in the module, which are kept in the *call dependency graph*.

Example 1 A widely used analysis domain in the context of (constraint) logic programs is groundness analysis using the *Pos* abstract domain [1]. Descriptions in *Pos* are positive Boolean functions denoting groundness dependencies, for example the description $X \wedge (Y \leftrightarrow Z)$ denotes that X is ground and Y is ground iff Z is ground. In this domain the least description $\perp_{Pos}$ is $false$, and the greatest description $\top_{Pos}$ is $true$, least upper bound $\sqcup_{Pos}$ is $\vee$ and greatest lower bound $\sqcap_{Pos}$ is $\wedge$.

Consider the top-down analysis of the following module, from the initial call pattern $app(X, Y, Z) : X$, that is, where `app` is called with the first argument ground.

```
:- module app.
app(X,Y,Z) :- X = [], Y = Z.
```

```
app(X,Y,Z) :- X = [A|X1], Z = [A|Z1], app(X1,Y,Z1).
```

Analysis proceeds by iterative fixpoint evaluation. The first iteration through the program produces answer pattern $app(X,Y,Z) : X \mapsto X \wedge (Y \leftrightarrow Z)$ (the answer from the first rule), as well as a call dependency $app(X,Y,Z) : X \rightarrow app(X,Y,Z) : X$. In the second iteration the same answer pattern is produced and so we have reached a fixpoint. $\square$

3 A model for separate module compilation

The principal idea of our compilation model is that at each stage the analysis information and associated executable code is correct, but recompilation may use more accurate analysis information to better optimize the generated code. Each compilation of a module asks the modules it calls for new (more accurate) analyses of the procedures it calls, and tells its calling modules whether more accurate information is available for their use. In this section we present a framework where the individual modules are not being modified and, thus, the information inferred during each compilation is never invalidated. In the next section we examine the use of the separate compilation model in the edit-compile-test cycle, where modules are being modified.

During modular analysis, information about each exported procedure is kept in the *analysis registry.* The analysis registry, or registry for short, is an extended answer table with the most up to date analysis information about procedures together with book keeping information for the compilation process. It is used when compiling a module to access information about procedures in other modules. Information about procedures is updated when their module is (re-)compiled. For simplicity, we assume that the analysis registry is global, but in practice we will store each answer pattern $P : CP \mapsto AP$ attached to the module M in which procedure P resides (see Section 5 for more details).

Entries in the analysis registry may be marked, written $P : CP \mapsto^{\bullet} AP$. A marked entry indicates that recompilation of the module containing P may infer better information. Entries may also be followed by another call pattern $P : CP'$ for the same procedure P, written $P : CP \mapsto AP$ $(P : CP')$, where $P : CP'$ is called the *version call pattern.* At any time there is only one entry of the form $P : CP \mapsto \_$ (with or without version call pattern) for each $P : CP$.

A version call pattern indicates that a specialized procedure for $P : CP$ does not exist and instead the specialized procedure for $P : CP'$ should be used for linking. It also indicates that the compiler has not (yet) performed context sensitive analysis for $P : CP$, but rather has used information in the analysis registry to infer the answer pattern for $P : CP$. In an entry $P : CP \mapsto AP$ $(P : CP')$ it is always the case that $CP \preceq CP'$ since it must be legitimate to use the existing procedure $P : CP'$ for calls of the form $P : CP$.

The analysis registry is initialized with (marked) entries of the form $P : \top \mapsto^{\bullet} \top$ for each exported procedure P in the program. This gives no information, but ensures a correct answer is available for each procedure.

The *inter-module dependency graph* is a call dependency graph with entries of the form $P_1 : CP_1 \rightarrow P_2 : CP_2$ where P_1 and P_2 are procedures exported by different modules. It is used to assess which modules may benefit from recompilation after the module exporting P_2 is (re-)compiled.

The key to the method is that the analysis registry always contains correct information, hence (optimizing) compilation of a module can safely make use of this information.

3.1 Compiling a single module

The first step in compiling a module M is determining its initial call patterns. This is simply the list of $P : CP$ appearing in a marked entry in the analysis registry where P is a procedure exported by M. Analysis proceeds from these initial call patterns as usual. During analysis of M, information for procedures in other modules is accessed by looking in the analysis registry. Suppose imported procedure P is being called with call pattern CP. Two cases are considered:

1. If $P : CP \mapsto AP$ (or $P : CP \mapsto AP$ $(P : CP')$) exists in the analysis registry then the analysis can use AP as the answer description.
2. Otherwise, the analysis selects an entry in the registry of the form $P : CP' \mapsto AP'$ (or $P : CP' \mapsto AP'$ $(P : CP'')$) such that $CP \preceq CP'$ and uses AP' as a correct (but possibly inaccurate) answer description. Note that such an entry must exist since there is always an entry $P : \top \mapsto AP'$ for every exported procedure.

   If there are several entries of the appropriate form, analysis will choose one whose answer is no less accurate than the rest, i.e., an AP' will be chosen such that for all other entries of the form $P : CP'_i \mapsto AP'_i$ or $P : CP'_i \mapsto AP'_i$ $(P : CP''_i)$ in the registry for which $CP \preceq CP'_i$ we have that $AP'_i \not\prec AP'$.[1]

   A new entry $P : CP \mapsto^{\bullet} AP'$ $(P : CP_{imp})$ is then added to the analysis registry, where $P : CP_{imp}$ is either $P : CP'$ if the selected entry does not have a version call pattern or $P : CP''$ otherwise. The mark indicates that the new call pattern could be refined by recompilation, and the version call pattern indicates its best current implementation.

The algorithm $\mathsf{get\_answer}(P : CP)$ below defines the answer pattern returned for a calling pattern $P : CP$. For the purposes of matching in the algorithm we assume that an entry without version call pattern $P : CP \mapsto AP'$ is equivalent to an entry with the version call pattern $P : CP \mapsto AP'$ $(P : CP)$

$\mathsf{get\_answer}(P : CP)$

[1] If the behaviour of code resulting from multivariant specialisation is the same as the original program code with respect to the analysis domain, then a better choice for AP' is $\sqcap_i AP'_i$. Even better, if the analysis domain is downwards closed, the answer description $CP \sqcap AP'$ is also correct for $P : CP$ and, in general, more accurate since $CP \sqcap AP' \preceq AP'$. Thus, in this case, we can choose AP' to be $CP \sqcap (\sqcap_i AP'_i)$.

if exists $P : CP \mapsto AP'$ $(P : CP')$ in registry
  return AP'
else
  $AP := \top$; $CP_{imp} := \top$
  foreach $P : CP' \mapsto AP'$ $(P : CP'')$ in registry
    if $CP \preceq CP'$ and $AP' \preceq AP$
      $AP := AP'$
      $CP_{imp} := CP''$
  add $P : CP \mapsto^{\bullet} AP$ $(P : CP_{imp})$ to the registry
  return AP

Once analysis is complete, the analysis registry is updated as follows. For each initial call pattern $P : CP$ the entry in the analysis registry is modified to $P : CP \mapsto AP$ as indicated by the answer table for module M. Note that any previous mark or additional version call pattern is removed. If the answer pattern has changed then, for each $P_1 : CP_1 \rightarrow P : CP$ appearing in the inter-module dependency graph, the entry in the analysis registry for $P_1 : CP_1$ is marked. Note that after analysis of a module, none of its procedures will be marked or have version call patterns in the analysis registry.

The inter-module dependency graph is updated as follows. For each initial call pattern $P : CP$ any entries of the form $P : CP \rightarrow P_2 : CP_2$ are removed. For each $P_3 : CP_3$, where P_3 is an imported procedure reachable from $P : CP$, the entry $P : CP \rightarrow P_3 : CP_3$ is added to the inter-module dependency graph.

Example 2 Consider the analysis of a simple program consisting of the modules

```
:- module main.
:- import rev.
main(X) :- A = [1,2,3], rev(A,X).

:- module rev.
:- import app.
rev(X,Y) :- X = [], Y = [].
rev(X,Y) :- X = [A|X1], rev(X1,Y1), B = [A], app(Y1,B,Y).

:- module app.
app(X,Y,Z) :- X = [], Y = Z.
app(X,Y,Z) :- X = [A|X1], Z = [A|Z1], app(X1,Y,Z1).
```

using groundness analysis domain *Pos*. Assume we want to compile module `rev` after we have compiled modules `main` and `app`, with current analysis registry

$$
\begin{aligned}
&main(X) : true \mapsto true,\\
&rev(X,Y) : true \mapsto^{\bullet} true,\\
&rev(X,Y) : X \mapsto^{\bullet} true\ (rev : true),\\
&app(X,Y,Z) : true \mapsto (X \wedge Y) \leftrightarrow Z,
\end{aligned}
$$

and inter-module dependencies $main(X) : true \to rev(X,Y) : X$. See Example 5 to determine how the compilation process might reach this state.

Given the above analysis registry, the initial call patterns are $rev(X,Y) : true$ and $rev(X,Y) : X$. Analyzing both of them we obtain answer table

$$rev(X,Y) : true \mapsto true,$$
$$rev(X,Y) : X \mapsto X \wedge Y$$

and call dependency arcs

$$rev(X,Y) : true \to rev(X,Y) : true,$$
$$rev(X,Y) : X \to rev(X,Y) : X,$$
$$rev(X,Y) : true \to app(X,Y,Z) : true,$$
$$rev(X,Y) : X \to app(X,Y,Z) : X \wedge Y.$$

During the analysis the new calling pattern $app(X,Y,Z) : X \wedge Y$ is generated and it uses the answer for $app(X,Y,Z) : true$ adding the entry $app(X,Y,Z) : X \wedge Y \mapsto^{\bullet} (X \wedge Y) \leftrightarrow Z\ (app(X,Y,Z) : true)$ to the registry.

The updating stage moves the answer patterns to the registry, marks the entry $main(X) : true \mapsto true$, and adds the inter-module dependency arcs (the last two call dependency arcs) to the inter-module dependency graph. □

3.2 Obtaining an executable without recompilation

The single module compilation algorithm must generate correct answers since it uses the information from the analysis registry, and that information is known to be correct (of course we assume the underlying local analysis algorithm is correct). Hence, the analysis information generated can be safely used to guide the optimization of the compiled code of the module. In particular, information about different call patterns facilitates multivariant specialisation of the procedure. Different variants of a procedure are generated if different optimizations can be performed depending on the different call patterns of the procedure.

Example 3 Several variants may be generated for the same procedure. We distinguish them by replacing the name of the procedure by the corresponding call pattern. Continuing with Example 2, compilation produces two variants for procedure *rev*, identified by `'rev:`$true$`'` and `'rev:`X`'`, respectively. The second one can be optimized to:

```
'rev:X'(X, Y) :-
   ( X == []
   -> Y = []
   ;  X =: [A|X1],
      'rev:X'(X1,Y1),
      B := [A],
      'app:X ∧ Y'(Y1,B,Y)
   ).
```

where `==` is a test, `:=` is an assignment and `=:` is a deconstruction. □

Once an object file exists for each module, an executable for the entire program can be produced by simply linking all the object files as usual (modulo the small change to the linker described below). The resulting executable is correct and any optimizations performed for specialized procedures are guaranteed to be valid.

Linking is made between specialized procedures as usual. From the linkers point of view the pair $P : CP$ is just a unique identifier for a specialized variant of procedure P. There is one additional complexity. The linker must use the analysis registry information to handle non-existent specialized variants. If the specialized procedure $P : CP$ does not exist then the link is made instead to the procedure $P : CP'$ where the entry in the analysis registry is $P : CP \mapsto AP\ (P : CP')$.

Example 4 After the compilation in Example 3, the call to '`app:`$X \wedge Y$' in the body of '`rev:`X' is linked to the procedure '`app:`$true$'. □

3.3 Obtaining an executable by performing recompilation

Although linking can be performed at any time in the program development process without requiring any additional work, recompilation of module M might improve the analysis information if there is a procedure P in M with a marked entry $P : CP \mapsto^{\bullet} AP$ in the analysis registry. In turn, recompiling a module may mark procedures in modules that it calls (if new call patterns are created) and in modules that call it (if more accurate answer information is obtained).

We may continue choosing a module M which can benefit from recompilation and recompile it as above until either there are no marked entries in the analysis registry anymore (a "completely optimized" version of the program has been reached) or we decide not to perform any further recompilation. The latter is possible because the algorithm guarantees that after any number of recompilation steps the compiled program and the corresponding linked object is correct, even if marked entries still remain in the registry. This is important since the aim of the separate compilation model we propose is not to use a long full optimization process often. Instead, during the lifetime of the program various modules move towards fully optimized status as the program is modified and recompiled.

The process of repeated compilation is guaranteed to terminate if the abstract domain has no infinite descending chains. This is because every new answer description is guaranteed to be more specific than the previous answer description, and any new call pattern must be more specific than previous call patterns. More importantly, as long as there are no cyclic inter-module dependencies the process of repeated compilation is guaranteed to be as accurate as the analysis of the entire program at once. Essentially, this is because since there are no cycles there is only one fixpoint and the greatest fixpoint discovered by repeated compilation is equivalent to the least fixpoint discovered by a usual analysis of the entire program at once (see [2] for the formal proof).

Example 5 Consider compiling the program of Example 2 from scratch. Initially the analysis registry contains

$$\begin{aligned}&main(X) : true \mapsto^\bullet true,\\ &rev(X, Y) : true \mapsto^\bullet true,\\ &app(X, Y, Z) : true \mapsto^\bullet true.\end{aligned}$$

Compiling module `rev` first we obtain answer table $rev(X, Y) : true \mapsto true$ and call dependency arcs $rev(X, Y) : true \to app(X, Y, Z) : true$ and $rev(X, Y) : true \to rev(X, Y) : true$. The analysis registry has the mark removed from the $rev(X, Y) : true \mapsto true$ entry and the arc $rev(X, Y) : true \to app(X, Y, Z) : true$ is added to the inter-module dependency graph.

Compiling module `app` we obtain answer pattern $app(X, Y, Z) : true \mapsto (X \wedge Y) \leftrightarrow Z$. This replaces the entry in the registry and the entry for $rev(X, Y) : true \mapsto true$ is marked.

During analysis of `main` the new call pattern $rev(X, Y) : X$ is generated, which causes an entry $rev(X, Y) : X \mapsto^\bullet true$ $(rev(X, Y) : true)$ to be added to the registry. The answer $main(X) : true \mapsto true$ overwrites the previously marked version while the inter-module dependency $main(X) : true \to rev(X, Y) : X$ is added to the inter-module dependency graph. At this point we could link together an executable.

Recompiling `rev` is described in Example 2. An executable could be produced also at this point, where calls to $app(X, Y, Z) : X \wedge Y$ will be linked to the code for $app(X, Y, Z) : true$.

Recompiling `main` only changes its registry entry to $main(X) : true \mapsto X$. Recompiling `app` replaces entry

$$app(X, Y, Z) : X \wedge Y \mapsto^\bullet (X \wedge Y) \leftrightarrow Z \ (app(X, Y, Z) : true)$$

by

$$app(X, Y, Z) : X \wedge Y \mapsto X \wedge Y \wedge Z$$

and marks entry $rev(X, Y) : X \mapsto X \wedge Y$. An optimized version for $app(X, Y, Z) : X \wedge Y$ is produced. Recompiling `rev` produces no changes. The program is now completely optimized. □

4 The compilation model in the edit-compile-test cycle

Editing a module might invalidate the analysis information appearing in the registry. Hence, we need to be able to recover a correct state for the program. In this section we provide an algorithm which guarantees that the analysis information for all modules in the program is correctly updated after editing a module.

Given a sensible interface between modules we believe that, for many analyses, even substantial changes to a module are unlikely to invalidate the answer patterns in the analysis registry. Hence many edits will not cause the invalidation of information previously used in other modules and thus often it will require recompilation of few or no modules to regain a correct state.

4.1 Recompiling an edited module

Recompilation of an edited module is almost identical to the simple compilation described in Section 3.1 above. The only changes are that the initial call patterns used are all those $P : CP$ appearing in the analysis registry[2] rather than just marked ones, and that the analysis registry is updated differently.

The updating of the analysis registry is as follows. As before, for each initial call pattern $P : CP$ the entry $P : CP \mapsto AP$ in the registry is replaced by the answer pattern $P : CP \mapsto AP'$ appearing in the answer table for module M. As before, any mark or additional version call pattern of the entry is removed. If $AP' \prec AP$, then, as before, for each $P_1 : CP_1 \to P : CP$ appearing in the inter-module dependency graph the entry for $P_1 : CP_1$ is marked. If $AP' \not\preceq AP$ the answer pattern has changed in a way that invalidates old information. Then, for each $P_2 : CP_2$ such that there is an arc $P_2 : CP_2 \to P : CP$ in the inter-module dependency graph, the entry $P_2 : CP_2 \mapsto AP_2$ is changed to $P_2 : CP_2 \mapsto^{\perp} AP_2$ to indicate that the current specialized procedure is invalid.

The algorithm update_registry below describes how the registry is updated from answer table $Answers$, obtained by having analysed module M.

update_registry($Answers$)
    foreach $P : CP \mapsto AP'$ in $Answers$
        let $e \equiv P : CP \mapsto AP$ $(P : CP')$ be the matching entry in registry
        replace e in registry by $P : CP \mapsto AP$
        if $AP' \prec AP$
            foreach $P_1 : CP_1 \to P : CP$ in inter-module dependency graph
                mark the entry $P_1 : CP_1 \mapsto AP_1$ with a $\bullet$
        else if $AP \prec AP'$
            foreach $P_2 : CP_2 \to P : CP$ in inter-module dependency graph
                mark the entry $P_2 : CP_2 \mapsto AP_2$ with a $\perp$

Example 6 Consider what happens if module app is changed to

```
:- module app.
app(X,Y,Z) :- X = [], Y = Z.
app(X,Y,Z) :- X = [A], Z = [A].
```

Recompilation finds information $app(X, Y, Z) : true \mapsto (X \wedge (Y \leftrightarrow Z)) \vee X \leftrightarrow Z$ and $app(X, Y, Z) : X \wedge Y \mapsto X \wedge Y \wedge Z$. Because of the change in the answer pattern for $app(X, Y, Z) : true$ the entry $rev(X, Y) : true \mapsto true$ is updated to $rev(X, Y) : true \mapsto^{\perp} true$. The entry for $rev(X, Y) : X$ remains unchanged. □

[2] By comparing the original and edited versions of the module we could determine the procedures P that might have been affected by the edits and only reanalyse them. Of course, this would require us to have saved the previous version of the module code, as well as the answer table of the last analysis. If we also had saved the (intra-module) dependency graph, then incremental analysis can be used to reanalyse the module (see [7]).

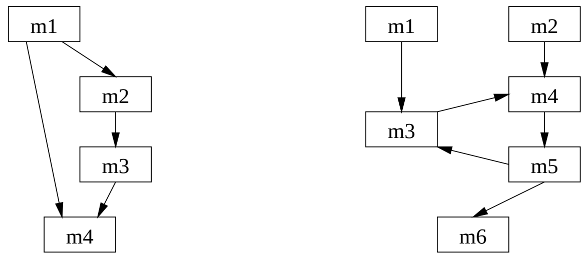

Fig. 1. Two example inter-module dependency graphs, one with a cycle

Note that any specialized procedure $P' : CP'$ from where there is a path to $P : CP$ in the inter-module dependency graph is potentially invalidated. We do not mark them all since we may be able to determine that they are not invalidated and prevent additional work.

Example 7 Consider the module import graph on the left of Figure 1. An arc from module m to m' indicates that module m imports module m'. If module $m4$ is edited then modules $m1$ and $m3$ may have entries marked as invalid. It may also mean that some procedures in module $m2$ are invalid, but we will defer marking them until we have recompiled $m3$ to determine if the invalidation of $m4$ has actually invalidated answer patterns in $m3$. □

4.2 Recompiling an invalidated module

Recompilation of an invalidated module is similar to normal compilation. The important difference is that care must be taken in order to avoid using invalid analysis information. An invalidated module can be recompiled if there are no paths in the inter-module dependency graph from its invalidated call patterns to other invalidated call patterns. Clearly for such modules we will not use (even indirectly) any invalidated information in the analysis registry, and hence the result of the recompilation will be guaranteed to be correct.

The recompilation procedure is as for an edited module but only marked entries (marked with either $\bullet$ or $\perp$) are used as initial call patterns.

Example 8 Continuing Example 7, the first module to be recompiled must be $m3$, assuming $m1$ has a path to an invalid procedure in $m3$. For the cyclic import graph on the right there may be no module that can be recompiled first after $m6$ is edited. □

Example 9 Continuing after the edit made in Example 6 module `rev` must be recompiled. The only invalidated call pattern is $rev : true$ and the same answer pattern $rev : true \mapsto true$ is found. Since it matches the old answer no further invalidation is required and the program is again in a correct state. □

The process of recompilation must continue until there are no entries marked $\perp$ in the registry before we can rebuild a valid executable.

4.3 Cyclic inter-module dependencies

If there are no cycles in the inter-module dependency graph (true if, for example, there are no cycles in the module import graph) then there is always an invalidated call pattern which does not depend on another invalidated call pattern. But if cycles exists we cannot simply recompile a single module; instead a different approach must be taken.

The simplest solution is to throw away all the possibly invalid information. For each $P : CP$ in the strongly connected component that contains an invalidated entry, we can reset the analysis registry entry to $P : CP \mapsto^{\bullet} \top$ (we could do better than this with some care). For each $P' : CP'$ not in the strongly connected component with an arc $P' : CP' \to P : CP$, we mark the entry as invalid. Each module involved in the strongly connected component can now be recompiled, since they now do not depend on invalid information. Note that for simplicity we may treat cycles in the import graph as an overestimation of cycles in the inter-module dependency graph (as we do in the examples below).

Example 10 Consider the module import graph on the right of Figure 1. If $m6$ is edited then $m5$ is invalidated and (assuming the cycle exists at the procedure level) all procedures in $m3$, $m4$ and $m5$ must be reset to $\top$ and the calling procedures in $m1$ and $m2$ invalidated to achieve a correct state. $\square$

The above approach is simple, but may lead to considerably more recompilation than is really required. A better approach is to break a cycle by choosing a module to start (re)analyse from. We then "zero" the information for those modules which it depends on and which have invalid information. We repeat this, choosing modules that depend on modules that have just been reanalysed.

Example 11 Consider again the import graph illustrated on the right of Figure 1. If $m6$ is edited causing an invalidation of old information, then all the other modules are now potentially invalidated. By resetting all the answer information in $m3$ (instead of $m3$, $m4$ and $m5$) to $\top$ we could now safely recompile $m5$, since it does not depend on any invalid information. Since there is no longer a cycle of invalidated information we can continue as in Section 4.2. $\square$

5 Implementation

We now show the first results of very preliminary experimentation with the implementation of our modular analysis system. The aim of this section is not to report on a complete evaluation, but rather to show the feasibility of the approach and the possible advantages in terms of (memory) performance.

The modular analysis has been integrated in the Plai analysis system, written in Ciao Prolog. The current implementation uses a global analysis registry, which is kept in memory throughout the analysis of a program until a fixpoint is reached (memory consumption could be optimised by keeping the analysis registry entries for each module in a separate file). The registry entries are asserted in the Prolog database. They include answer patterns (without version call patterns) but no call dependencies; instead, dependencies are stored at the module level, that is, when a new answer pattern is found in module M, all exported procedures of each module M' importing M are marked for re-analysis. This results in more processing than required, since only a few (or none) of the procedures of module M' may depend on the procedures in M which actually changed.

The registry is initially empty, but the analyser assumes a topmost answer when no applicable entry can be found in the registry, which is equivalent to initializing the registry with topmost answer patterns for all exported procedures in the program.

The current implementation's use of a single global registry is for simplicity. It would be better if the analysis registry and inter-module dependency graph was stored as a separate *info file* associated with each module. The info file for module M stores, for procedures P defined in M, answer patterns of the form $P : CP \mapsto AP$, as well as the call dependencies of the form $P' : CP' \rightarrow P : CP$ where P' is a procedure from another module M'. When compiling module M we then update the answer patterns in its info file and add the new call patterns that may have appeared to the info files of the imported modules. To determine when a module needs be recompiled, we simply examine its info file for marked entries. Note that using this arrangement, all the information required to compile a module M is found in its info files and those of the modules it imports. Our implementation works with this storage methodology. However, info files are cached in memory, so that the overall effect is that of a global analysis registry, as mentioned before.

In our experiment we analyse a set of benchmarks[3] using our modular approach: during the first iteration each module is analysed; in following iterations only those modules which have marked entries in their info files will be analysed, until no marked entry remains. This approach is compared to that of analysing the whole program at once. The comparison has been carried out on three analysis domains, namely *shf* (set-sharing and freeness information, see [9]), *son* (pair-sharing and linearity information, see [18]), and *def* (a simplified version of the *Pos* domain, see [1]).

The following table shows size statistics for each benchmark program: the number of predicates, clauses and literals in the program as well as the average and maximum number of variables per literal. The analysis results given are the the analysis time of the whole non-modular program (Whole) and of the modular program (Module) in seconds, the number of iterations of the modular analysis (#IT) and the number of modules analysed in each iteration (#Mods); the first of these numbers is the number of modules forming the complete program. The analysis times shown in the table are an average of the analysis (wall-)times for 10 executions of the analyser, performed in a PC with two Intel PII processors at

[3] The benchmarks used are available at `http://www.clip.dia.fi.upm.es/bueno`.

400MHz, SuperMicro motherboard at 100MHz, 512Mb ram, and running Red Hat Linux 6.2. Note that we do not take into account any Ciao libraries the benchmarks might use, i.e., the libraries are assumed to be previously analysed and thus do not contribute to the analysis time.

Program	Preds	Clauses	Lits	Vars/Cls Ave	Vars/Cls Max	Dom	Whole	Module	#IT	#Mods
boyer	29	144	64	3	6	shf	1	1.8	2	3-2
						son	0.5	0.8	2	3-2
						def	0.6	0.8	2	3-2
wms2	63	1339	269	6	22	shf	–	–		
						son	–	12.6	3	7-5-1
						def	5.1	32.5	2	7-2
chat80	459	2822	1416	6	27	shf	–	–		
						son	–	–		
						def	–	34.4	2	21-8
icost	1159	2681	6203	6	100	shf	–	–		
						son	–	–		
						def	–	219.6	4	12-12-12-2

The size of the programs vary considerably, and the complexity of each program is also different. The core of `boyer` is a pair of recursive predicates, one of which is also mutually recursive with a third one. In `wms2`, the central part is a multiply recursive predicate based on another two multiply recursive predicates; there is no mutual recursion. Programs `chat80` and `icost` have a large number of multiply and mutually recursive predicates.

Regarding the modular versions, only `chat80` and `icost` have cycles in the module import graph (which the implementation takes as an overestimation of the inter-module dependency graph). There is only one cycle in `chat80`, which includes two modules. In contrast, in `icost` there are many cycles, which basically amounts to ten of the twelve modules being included in a single strongly connected component.

The program `boyer` is a simplified version of the Boyer-Moore theorem prover (written by Evan Tick). The core part of the program has been divided in two separate modules (plus a third one providing data to both these). These two modules need a second iteration of modular analysis to complete. Obviously there is overhead in the modular analysis of a program that can be analysed all at once. The `boyer` benchmark shows that this is not overwhelming.

The program `wms2` solves a job allocation problem (written by Kish Shen). Analysis of the whole program with domains *shf* and *son* runs out of memory. However, a simple modularization (i.e., taking the tables of data to different modules) allows analysis with the *son* domain. The difficulty in analysing the whole program seems simply to be the large amount of code to analyse.

On the other hand, having the data in different modules increases the overhead in modular analysis in the case of domain *def*, compared to that of `boyer`. In this domain, the lack of information on the data slows down the analysis of the core part of the algorithm in the first iteration. In the second iteration, once the data modules are analyzed, this does not happen.

The program `chat80` is the famous natural language query answering system of F. Pereira and D. H. D. Warren. Again in this case, the analysis of the whole program runs out of memory. Converting each of the original files in the program in a different module, the program can now be analysed in two iterations. The difficulty in this program seems to be the amount of analysis information that needs be handled: It can be handled separately for each module, but not when it is put together.

A similar thing happens with program `icost`. This one is a version of Caslog, originally developed by Nai-Wei Lin, which has been further developed by Pedro López. In this case, every subpart of the original program has been converted into a module (which includes the time complexity, size, number of solutions, dependency and determinacy analyses which are part of the functionality of the program). This allows modular analysis with domain *def*, while the analysis of the whole program was not possible.

6 Conclusion

We have presented a simple algorithm for context sensitive analysis and optimization of modular programs. It is efficient in the sense that it keeps track of which information can be improved through reanalysis, and only performs this reanalysis. It has the advantage that correct executables can be produced after compiling each module just once and, if there are no cycles in the import graph, the final result from repeated recompilation is as accurate as analyzing the entire program globally. Our experimental results illustrate that modular analysis allows global optimization to be applied to much larger programs than if we are forced to analyse the whole program at once. To our knowledge, this is the first generic proposal for a context sensitive analysis that truly allows separate compilation of modules.

The technique may produce multiple versions of predicates to allow different optimizations. It may appear that the proliferation of different specialized versions will result in an excessive program size. However, in our experience with the CLP(R) optimizing compiler [8] and the Ciao Prolog parallelizing compiler [10] the resulting sizes are reasonable. Also, it is easy to modify the approach to only allow two versions of a procedure P, the most general $P : \top$ and a single specialized version $P : CP$. Whenever a new call pattern $P : CP'$ is generated it is compared with the (unique) specialized version in the registry $P : CP \mapsto AP$. If $CP' \preceq CP$ then there is no change. Otherwise, the entry is replaced with $P : (CP \sqcup CP') \mapsto^{\bullet} AP_{\top}$ $(P : \top)$ where $AP_{\top}$ is the answer pattern for $P : \top$.

Although throughout the paper we have used logic programs as our examples, the approach is relatively independent of the underlying language. The same approach is applicable to context sensitive analysis of other programming languages with a clean separation of exported and private parts of a module. Of course other issues which complicate the context sensitive analysis, such as tracking global state, dynamic scheduling and higher-order function calls, may require considerable additional machinery to handle.

Acknowledgements

The work of the authors from Technical University of Madrid (UPM) has been partially supported by Spanish research project EDIPIA, TIC99-1151.

References

1. T. Armstrong, K. Marriott, P.J. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In *Proceedings of the 1st International Static Analysis Symposium*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, 266–280, 1994.
2. F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P.J. Stuckey. Inter-module Analysis and Optimizing Compilation. Department of Computer Science and Software Engineering, University of Melbourne, Forthcoming Technical Report, 2001.
3. F. Bueno, M. García de la Banda, M. Hermenegildo, and K. Muthukumar. Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism. *Journal of Logic Programming 38,* 2, 165–218.
4. M. Codish, S.K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 451–464, Charleston, South Carolina, 1993. ACM.
5. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, 238–252, 1977.
6. S.I. Feldman. Make–a program for maintaining computer programs. *Software – Practice and Experience*, 1979.
7. M. Hermenegildo, G. Puebla, K. Marriott, and P.J. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, 2000.
8. A. Kelly, A. Macdonald, K. Marriott, H. Søndergaard, and P.J. Stuckey. Optimizing compilation for CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems*, 20(6):1223–1250, 1998.
9. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
10. G. Puebla and M. Hermenegildo. Abstract multiple specialization and its application to program parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, 1999.
11. G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In [20].
12. A. Rountev, B.G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Proceedings of ESEC/FSE '99*, volume 1687 of *LNCS*, pages 235–252. Springer-Verlag, 1999.
13. Z. Shao and A. Appel. Smartest recompilation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 439–450, Charleston, South Carolina, 1993. ACM.

14. R.W. Schwanke and G.E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, 1988.
15. Z. Somogyi, F. Henderson and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17-64, 1996.
16. W. Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, 1986.
17. W. Vanhoof and M. Bruynooghe. Towards modular binding-time analysis for first-order Mercury. In [20].
18. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.
19. W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.
20. *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.