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
[2] Monash University, Australia
[3] University of Melbourne, Australia

1 Introduction

Automatic program analysis, generally based on abstract interpretation [1], is a relatively mature technique which allows obtaining at compile-time useful information on the run-time behavior of the program. This information can then be used to optimize the program by *specializing* the program procedures for particular cases. If analysis is *multi-variant* then several *variants* may be generated for a given procedure [6] allowing different optimizations to be performed, a technique which has shown quite useful in practice [3, 2].

Separating programs into modules is a well-known technique which has proven very useful in program development and maintenance. Unfortunately, though analysis and specialization of modular programs has recently received some attention [4, 5], most existing analysis and specialization techniques cannot handle modular programs. In this paper we propose a model for analysis and specialization of modular programs. We consider a program development scenario in which the program is divided into modules and each module developed separately, possibly by different programmers, and where the programmers developing each module want to perform optimized compilation of the whole program. We present a simple and efficient analysis and optimization algorithm for this scenario. See [4] for a general discussion of different scenarios and the overall issues which appear in the analysis and specialization of modular programs.

2 Preliminaries and notation

The top-down analysis of a module starts from descriptions of the initial calls to the module and computes descriptions of all the possible calls to the module procedures and of their 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

* Extended Abstracts of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

poset are denoted $\bot$ and $\top$, respectively. 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 argument positions of P when it is called. 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 $P : CP$. A description with value $\bot$ indicates an answer (or call) which does not occur. A description with value $\top$ 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 $P : CP \rightarrow P' : CP'$ between call patterns which indicates that calls of the form $P' : CP'$ (might) occur during execution of calls of the form $P : CP$. Analysis also computes all possible call dependencies in the module, which are kept in the *call dependency graph*.

3 A model for separate module compilation

The principal idea of our model is that at each stage the produced executable is correct, but recompilation may use more accurate analysis information to obtain more optimized executables. 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-test-edit 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 should store each answer pattern $P : CP \mapsto AP$ and each call dependency $P' : CP' \mapsto P : CP$ attached to the module M in which procedure P resides.

Entries in the analysis registry may be marked, written $P : CP \mapsto^* AP$, and 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*. A marked entry indicates that recompilation may infer better information. 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. The analysis registry is initialized with (marked) entries of the form $P : \top \mapsto^* \top$ for each exported procedure P in the program.

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. Note that if there are no cyclic module dependencies then there are no cycles in the graph.

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 $P : CP \mapsto AP$ exists in the analysis registry then analysis can use AP as the answer description.
2. Otherwise, analysis determines an entry in the registry $P : CP' \mapsto AP'$ 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 $P : \top$ for every exported procedure. If there are several such entries, analysis will choose an answer AP' which is as accurate as possible, i.e., for all $P : CP'_i \mapsto AP'_i$ (including the entry $P : CP' \mapsto AP'$) in the registry such that $CP \preceq CP'_i$, $AP'_i \not\prec AP'$.[1] Note that if the analysis domain is downwards closed the answer description $CP \sqcap AP'$ is also correct for $P : CP$ and, in general, more accurate.
   A new entry $P : CP \mapsto^* AP'(P : CP')$ is then added to the analysis registry. The mark indicates that the new call pattern could be refined by recompilation, and the version call pattern indicates its best current implementation.

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 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.

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 1 Consider the analysis of a simple program consisting of the modules

```
:- module main.
```

[1] Note that the best choice would be an AP' composed as $\sqcap_i AP'_i$. However, this answer would not correspond in general to a version call pattern.

```
:- 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 ($\top_{Pos} = true$). Assume we want to compile module `rev` after we have compiled `main` and `app`, with current registry $main : true \mapsto true$, $rev : true \mapsto^* true$, $rev : X \mapsto^* true(rev : true)$ and $app : true \mapsto (X \wedge Y) \leftrightarrow Z$, and inter-module dependencies $main : true \rightarrow rev : X$.

The initial call patterns are $rev : true$ and $rev : X$. Analyzing both of them we obtain answer table $rev : true \mapsto true$ and $rev : X \mapsto X \wedge Y$ and call dependency arcs $rev : true \rightarrow app : true$ and $rev : X \mapsto app : X \wedge Y$. The updating stage adds the call dependency arcs to the inter-module dependency graph, moves the answer patterns to the registry, marks the entry $main : true \mapsto true$, and adds the new entry $app : X \wedge Y \mapsto^* (X \wedge Y) \leftrightarrow Z$ $(app : true)$.

Several variants may be generated for the same procedure. We distinguish them by replacing the name of the procedure by the corresponding call pattern. In our example, compilation produces two variants for procedure rev, identified by $'\texttt{rev} : true'$ and $'\texttt{rev} : X'$, respectively. The second one can be optimized to:

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

where `==` is a test, `hd` and `tl` retrieve the head and tail of a list respectively, and `:=` is an assignment.

3.2 Obtaining an executable without recompilation

It is easy to see that the single module compilation algorithm above is correct. Thus, the analysis information inferred can be safely used to optimize the compiled code of the module. This is accomplished by generating different variants of a procedure when different optimizations can be done depending on the different call patterns of the procedure. 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 2 After the compilation above, the call to `app:`$X \wedge Y$`(Y1,B,Y)` in the body of `rev:`X`(X,Y)` is linked to the procedure `app:`$true$.

3.3 Obtaining an executable 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 $P : CP \mapsto^* AP$ 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.

Example 3 Consider compiling the above example program from scratch. Initially the analysis registry contains $main : true \mapsto^* true$, $rev : true \mapsto^* true$ and $app : true \mapsto^* true$. Compiling module `rev` first we obtain answer table $rev : true \mapsto true$ and call dependency arc $rev : true \rightarrow app : true$. The analysis registry has the mark removed from the $rev : true$ entry and the arc is added to the inter-module dependencies. Compiling module `app` we obtain answer table $app : true \mapsto (X \wedge Y) \leftrightarrow Z$. This replaces the entry in the registry and the entry for $rev : true$ is marked. Compiling module `main` we obtain answer table $main : true \mapsto true$ and call dependency arc $main : true \rightarrow rev : X$ indicating it calls `rev` with first argument ground. These entries are added to the registry and inter-module dependencies, and another entry $rev : X \mapsto^* true(rev : true)$ is added to the registry. At this point we could link together an executable.

Recompiling `rev` is described in Example 1. Recompiling `main` only changes its registry entry to $main : true \mapsto X$. Recompiling `app` replaces entry $app : X \wedge Y \mapsto^* (X \wedge Y) \leftrightarrow Z(app : true)$ by $app : X \wedge Y \mapsto X \wedge Y \wedge Z$ and marks entry $rev : X \mapsto X \wedge Y$. An optimized version of $app : X \wedge Y$ is produced. Recompiling `rev` produces no changes. The program is now completely optimized.

4 The compilation model in the edit-test-edit 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.

However, since inter-module optimizations are only likely to be applied when (most of) the modules are (almost) correct, we believe the need for recovering correctness is rarely required. In addition, given a sensible interface between modules we argue that for many analyses even substantial changes to a module are unlikely to affect the analysis information in a way that it becomes wrong.

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 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 $P : CP \mapsto AP$ is modified to be 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 $P_1 : CP_1 \rightarrow P : CP$ 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. For each $P_2 : CP_2$ such that there is an arc $P_2 : CP_2 \rightarrow P : CP$ 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.

Note that any specialized procedure $P' : CP'$ 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 4 Consider modifying module `app` to

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

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

Example 5 Consider the illustration of the inter-module dependency graph on the left of Figure 1. If module $m4$ is edited then modules $m1$ and $m3$ will have entries marked as invalid. It may also mean that some procedures in module $m2$ are marked as invalid.

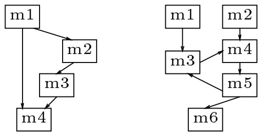

Fig. 1. Two example inter-module dependency graphs, one including cycles

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. This is achieved if the first invalidated module to be recompiled is one for which there are no paths in the inter-module dependency graph from its invalidated calling patterns to other invalidated calling patterns.

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

Example 6 Continuing Example 5, the first module to be recompiled must be $m3$, assuming $m1$ has a path to an invalid procedure in $m3$. For the cyclic dependency graph on the right there may be no module that can be recompiled first after $m6$ is edited.

Example 7 Continuing after the edit made in Example 4 module `rev` must be recompiled. The initial call pattern is only $rev : true$ and the same result $rev : true \mapsto true$ is found. Since it matches the old answer no further invalidation is required and the program is again in a correct state.

4.3 Cyclic module dependencies

If there are no cycles in the inter-module dependency graph (true if, for example, there are no cyclic dependencies among modules) then there is always an invalidated call pattern which does not depend on another invalidated call pattern. This is also true if in any strongly connected component of the graph there are invalidated call patterns of at most one module. However, in general, a different approach must be taken. Intuitively, the general solution is to throw away all the possibly invalid information.

For each $P : CP$ in the strongly connected component that contains an invalidated entry, reset the analysis registry entry to $P : CP \mapsto^* \top$ (we could do better than this with some care). For each $P' : CP'$ not in the strongly connected component with an arc $P' : CP' \rightarrow P : CP$, mark the entry as invalid. Each module involved can be recompiled before examining other invalidated modules.

Example 8 Consider the dependency graph illustrated on the right of Figure 1. If $m6$ is edited then $m5$ is invalidated and (assuming the cycle exists at the procedure level) all of the 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.

5 Discussion

We have presented a simple and efficient algorithm for analysis-based optimization of modular programs. 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 [2] and the Ciao Prolog parallelizing compiler [3] 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 $P : \top_{\mathcal{D}}$ 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 $P : (CP \sqcup CP') \mapsto^* AP_{\top}$ $(P : \top_{\mathcal{D}})$ where $AP_{\top}$ is the answer pattern for $P : \top$.

Regarding optimality of the analysis results, although the analysis information is correct at any time it can be sub-optimal, in the sense that analyzing the whole program code as a single unit may provide more accurate information. An alternative is suggested in [4] which guarantees optimality at the price of ensuring correctness only when a "completely optimized" version of the program is reached, which may require a high number of recompilation steps. The algorithm proposed herein can be easily modified as proposed in [4]. For this, the analysis registry should be initialized to $P : CP \mapsto^* \perp$ rather than $P : CP \mapsto^* \top$. It is our intuition that the proposed algorithm is much more efficient than the one which produces optimal results and in most cases the difference in accuracy is not very relevant. However, it remains as future work to compare the two approaches experimentally.

References

1. 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.
2. 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.
3. 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.
4. G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In [7].
5. W. Vanhoof and M. Bruynooghe. Towards modular binding-time analysis for first-order mercury. In [7].
6. W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.
7. *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.