Efficient Separate Compilation of Object-Oriented Languages*

Jean Privat, Floréal Morandat, and Roland Ducournau

LIRMM
Université Montpellier II — CNRS
161 rue Ada
34392 Montpellier cedex 5, France
{privat,morandat,ducour}@lirmm.fr

Abstract. Compilers of object-oriented languages used in industry are mainly based on a separate compilation framework. However, the knowledge of the whole program improves the efficiency of compilation; therefore the most efficient implementation techniques are global.
In this paper, we propose a compromise by including three global compilation techniques in a genuine separate compilation framework.

1 Introduction

According to software engineering, programmers must write modular software. Object-oriented programming has become a major trend because it fulfils this need: heavy use of *inheritance* and *late binding* is likely to make code more extensible and reusable.

According to software engineering, programmers also need to produce software in a modular way. Typically, we can identify three advantages: (i) a software component (e.g. a library) can be distributed in a compiled form; (ii) a small modification in the source code should not require a recompilation of the whole program; (iii) a single compilation of a software component is enough even if it is shared by many programs. Separate compilation frameworks offer these advantages since source files are compiled independently of future uses, and then linked to produce an executable program.

The problem is that the knowledge of the whole program allows more efficient implementation techniques. Therefore previous works use these techniques in a global compilation framework, thus incompatible with modular production of software. Global techniques allow efficient implementation of the three main object-oriented mechanisms: late binding, read and write access to attributes, and dynamic type checking.

In this paper, we present a genuine separate compilation framework that includes three global optimisation techniques. The framework described here can be used for any statically typed class-based languages.

* Position paper at ICOOOLPS Workshop at ECOOP 2006.

The remainder of the present paper is organised as follows. Section 2 presents the global optimisation techniques we consider. Section 3 introduces our separate compilation framework. We conclude in section 4.

2 Global Techniques

The knowledge of the whole program source code permits a precise analysis of the behaviour of each component and an analysis of the class hierarchy structure. Each of those allows important optimisations and may be used in any global compiler.

Type Analysis. Statistics show that most method calls are actually monomorphic calls. In order to detect them, type analysis approximates three mutually dependent sets: the set of the classes that have instances (live classes), the *concrete type* of each expression (the concrete type is the set of potential dynamic types) and the set of called methods for each call site. There are many kinds of type analysis [10]. Even simple ones give good result and can detect many monomorphic calls [1].

Coloring. Coloring is an implementation technique with *Virtual Function Table* (VFT) that avoids the overhead of multiple inheritance [12, 7]. It can be applied to attributes, to methods and to classes for subtyping check [5, 17, 4, 19, 7, 8]. Coloring is a global optimization which requires the knowledge of the whole class hierarchy and finding an optimal one is an NP-hard problem similar to the minimum graph coloring problem. Happily, class hierarchies seem to be simple cases of this problem and many efficient heuristics are proposed in [17–19].

Binary Tree Dispatch. SmartEiffel [20] introduces an implementation technique for object-oriented languages called *binary tree dispatch* (BTD). It is a systematisation of some techniques known as *polymorphic inline cache* and *type prediction* [11]. BTD has good results because VFT does not schedule well on modern processors since the unpredictable and indirect branches break their pipelines [6]. BTD requires a global type analysis in order to reduce the number of expected types of each call site. Once the analysis is performed, the knowledge of concrete types permits to implement polymorphism with an efficient select tree that enumerates types of the concrete type and provides a static resolution for each possible case.

3 Separate Compilation

Separate compilation frameworks are divided into two phases: a local one (compiling) and a global one (linking). The *local phase* compiles a single software component (without loss of generality, we consider the compilation units to be classes) independently from the other components. We denote *binary components*

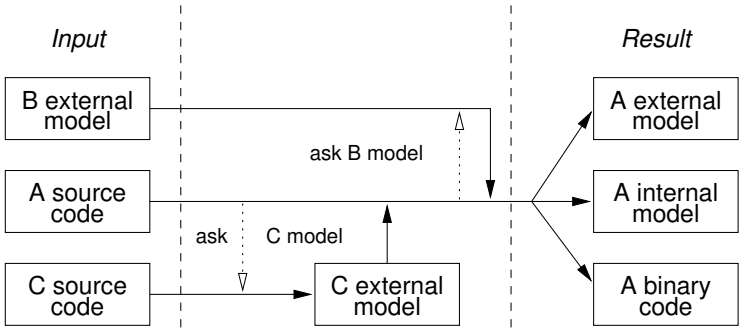

Fig. 1. Local Phase

the results of this phase[1]. *Binary components* are written in the target language of the whole compilation process (e.g. machine language) but they are not functional because some missing information is replaced by symbols. The *binary components* also contain metadata: debug information, symbol table, etc. The *global phase* gathers *binary components* of the whole program, collects some metadata, resolves symbols and substitutes them. The result of this phase is a functional *executable* that is the compiled version of the whole program.

Application of global techniques to this framework can only be done during the global phase since the knowledge of the whole program is needed. The problem is that the source code of the program is already compiled into binary components and no more available.

The idea to perform optimisations during the global phase is not new. Computing a coloring at link-time was first proposed by [17] but, to our knowledge, this has never been implemented. Other works, [9] and [2], propose a separate compilation framework with global optimisation respectively for Modula-3 and for functional languages. In both cases, the main difference with our approach is that their local phases generate code in an intermediate language. On linking, global optimisations are performed on the whole program then a genuine global compilation translates this intermediate language into the final language.

3.1 Local Phase

The local phase takes as its input the source code of a class, and produces as its results the *binary code* and two metadata types: the *external model* and the *internal model*—Fig. 1. These three parts can be included in the same file or in distinct files but the *external model* should be separately available.

[1] Traditionally, the results of separate compilation are called *object files*. Because this paper is about object-oriented languages, we chose not to use the traditional name to avoid conflicts.

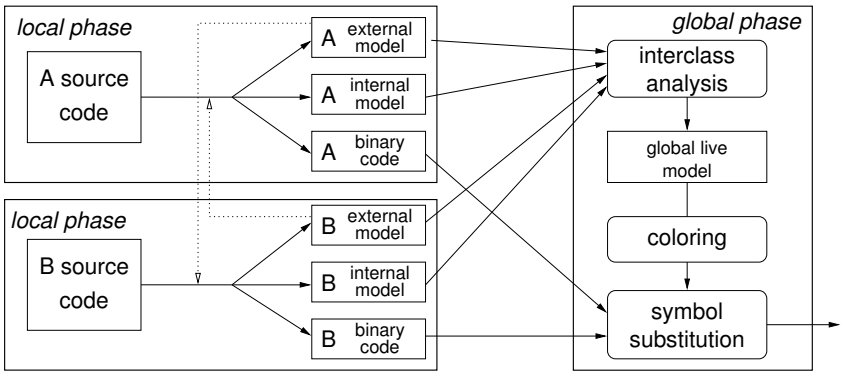

Fig. 2. Global Phase

The external model of a class describes its interface: superclasses and definitions of methods and attributes. Even if the local phase compile classes independently from their future use, classes still depend on superclasses and used classes. Thus, the external model of these classes must be available or be generated from the source file. In the latter case, a recursive generation may be performed.

The binary code contains symbols. As in standard separate compilation, symbols are used for addresses of functions and static variables. In our proposition, we also introduce other symbols related to the OO mechanism: (i) each late binding site is associated with a unique symbol, and compiled with a static direct call to this symbol; (ii) attribute accesses are compiled with a symbol representing the color of the attribute, i.e. the attribute index in the instance; (iii) type checks are compiled with two symbols representing the color and the identifier of the class to test.

The internal model of a class describes the behaviour of its methods. It gathers class instantiations, late binding sites, attribute accesses and type checks. It also contains the information about associated symbols. Using a type flow analysis, the internal model of a method also contains a graph which represents the circulation of the types between the *entries* (the receiver, a parameter, the reading of an attribute, or the result of a method call) and the *exits* (the result of the method, the writing of an attribute, or the arguments of a method call) of the method.

3.2 Global Phase

The global phase is divided into three stages: (i) type analysis which determines the *live global model*, (ii) coloring which computes colors and identifiers of classes and attributes, and (iii) symbol substitution in the binary code (Figure 2).

Type analysis is based on the internal and external models of all classes. The live classes and their live attributes and methods are identified, as well as the information on the concrete types of the live call sites.

The coloring stage is performed once the live global model is obtained. A heuristic [17, 18] produces the values of the identifiers and the colors of the live classes, methods and attributes, as well as the size of the instances.

The last stage substitutes values to symbols. Colors and identifiers computed during the coloring stage are substituted to the corresponding symbols. For each late binding site, the symbol is replaced according to the polymorphism of the call site. On a monomorphic site, the symbol is replaced by the address of the single method: the result is a direct call. On a polymorphic site, the symbol is replaced by the address of a resolver. Resolvers are small link-time generated functions that select the correct method. On an oligomorphic site, BTD is the most efficient, therefore resolvers only contain a select tree where leaves are static jumps to the correct function. On a megamorphic site, VFT is the most efficient, therefore resolvers only contain a jump to the required method in the function table.

4 Conclusion

We present in this article a genuine separate compilation framework for statically typed object-oriented languages in multiple inheritance. It includes three global techniques of optimisation and implementation: type analysis, coloring, and binary tree dispatch. Our proposition is a compromise between efficiency and modularity. It brings the efficiency of these global techniques without losing the advantages of separate compilation.

For experiments [16], we developed a compiler prototype called `prmc` for PRM, an EIFFEL-like language. It mainly follows the separate compilation scheme presented in the present paper. The only difference is that there is no external schema the global phase uses the source code to perform the type analysis. However, the compilation is still truly separate since units are compiled separately then linked.

In comparison with classical separate compilation, the space and time reductions are significant. Monomorphic, oligomorphic and megamorphic method calls are detected by the type analysis then are implemented with the most efficient technique (respectively direct call, BTD, and VFT). Attribute accesses and subtype checks are implemented with direct access.

Comparing with pure global compilers, the performances are honourable. However, from the point of view of efficiency, even if the quality of the type analysis is the same, SMARTEIFFEL and other global compilers keep a strong advantage with their code specialisation techniques: method inlining, *customisation* [3] or heterogeneous generic class compilation [13]. At least, like global compilers, our framework removes the justification of the two uses of the `virtual` keyword in C++ because the overhead of multiple inheritance (*virtual* inheritance) and monomorphic late binding (*virtual* functions) are removed.

The remaining question about libraries linked at load-time or dynamically loaded at run-time stays open.

References

1. David F. Bacon, M. Wegman, and K. Zadeck. Rapid type analysis for C++. Technical report, IBM Thomas J. Watson Research Center, 1996.
2. D. Boucher. *Analyse et Optimisations Globales de Modules Compilés Séparément*. PhD thesis, Université de Montréal, 1999.
3. C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In OOPSLA [14], pages 146–160.
4. N. H. Cohen. Type-extension type tests can be performed in constant time. *Programming languages and systems*, 13(4):626–629, 1991.
5. R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In OOPSLA [14].
6. K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 306–323. ACM Press, 1996.
7. Roland Ducournau. Implementing statically typed object-oriented programming languages. Technical Report 02-174, L.I.R.M.M., Montpellier, 2002.
8. Roland Ducournau. Coloring, a versatile technique for implementing object-oriented languages. Technical Report 06-001, L.I.R.M.M., Montpellier, 2006.
9. Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–115, 1995.
10. D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.
11. U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proc. ECOOP'91*, volume 512 of *LNCS*, pages 21–38. Springer-Verlag, 1991.
12. S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, New York (NY), USA, 1996.
13. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL'97*, pages 146–159. ACM Press, 1997.
14. *Proceedings of the Fourth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'89*, New Orleans, 1989. ACM Press.
15. *Proceedings of the Twelfth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*, SIGPLAN Notices, 32(10). ACM Press, 1997.
16. J. Privat and R. Ducournau. Link-time static analysis for efficient separate compilation of object-oriented languages. In M. Ernst and T. Jensen, editors, *Workshop on Program Analysis for Software Tools and Engineering PASTE'05*, pages 29–36, 2005.
17. W. Pugh and G. Weddell. Two-directional record layout for multiple inheritance. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90)*, ACM SIGPLAN Notices, 25(6), pages 85–91, 1990.
18. P. Takhedmit. Coloration de classes et de propriétés : étude algorithmique et heuristique. Mémoire de dea, Université Montpellier II, 2003.
19. J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. In OOPSLA [15], pages 142–157.
20. O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In OOPSLA [15], pages 125–141.