Intégration d'Optimisations Globales en Compilation Séparée des Langages à Objets

Jean Privat, Roland Ducournau

Intégration d'optimisations globales en compilation séparée des langages à objets

Jean Privat — Roland Ducournau

LIRMM – Université Montpellier II
161 rue Ada
34392 Montpellier Cedex 5 France
{privat, ducour}@lirmm.fr

RÉSUMÉ. *Les compilateurs majoritairement utilisés sont basés sur un principe de compilation séparée, alors que la plupart des optimisations des langages à objets nécessitent une connaissance globale du programme, en particulier l'analyse de type et les techniques d'implémentation de la liaison tardive. Les deux approches ont leurs avantages et leurs inconvénients et nous proposons d'intégrer des techniques d'optimisation globales, en particulier l'analyse de types et la coloration, dans un cadre de compilation séparée. Le code généré est décoré par des balises et complété par des informations sur le schéma des classes et la circulation des types dans leurs méthodes. Une phase globale précédant l'édition de liens effectue les optimisations globales à partir de ces dernières informations et fait les substitutions nécessaires dans le code généré par la phase locale.*

ABSTRACT. *Mainly used compilers are based on seprate compilation, whereas optimizations of object-oriented programs mostly need a complete knowledge of the whole program. This is especially the case for type analysis and late binding implementations. Both approaches have pros and cons. Therefore, this paper proposes an integration of global optimizations in a separate compilation framework. The code generated by the local step is tagged and completed with a class schema and a template abstracting the circulation of types in the class' methods. A global step coming before linking makes all global computations and subtitutes the code generated by the local step.*

MOTS-CLÉS : *langages à objets, compilation globale, compilation séparée, édition de liens, analyse de types, coloration de méthodes.*

KEYWORDS: *object-oriented languages, global compilation, separate compilation, linking, type analysis, selector coloring.*

1re soumission à *LMO'2004*, le 6 octobre 2003.

1. Introduction

Les compilateurs industriels les plus utilisés actuellement pour la production de logiciels sont basés sur un principe de compilation séparée. Chaque unité de code est compilée séparément, indépendamment de son utilisation effective, puis différentes unités sont assemblées pour construire des programmes exécutables. Pourtant, depuis de nombreuses années, les travaux de recherche sur la compilation des langages à objets ont permis de valider des techniques globales qui améliorent l'efficacité du code généré mais nécessitent la connaissance de la totalité du programme [CHA 89, DIX 89, VIT 94, DRI 95, COL 97, ZEN 97, DUC 97, GRO 01, ZIB 02]. Deux catégories de techniques sont principalement concernées : l'analyse de types qui permet de réduire le polymorphisme effectif des envois de messages et d'éliminer le code mort, et les structures de données nécessaires à l'implémentation des envois de message.

Cet article propose un schéma complet de compilation qui tente de réconcilier ces deux points de vue sur la compilation. Nous nous plaçons dans un cadre de typage statique. D'une part, les techniques globales comme l'analyse de types sont décomposées en une phase locale, réalisable au cours d'une compilation séparée, et une phase globale, qui peut s'effectuer sur la totalité des codes compilés. La phase de compilation séparée produit un code analogue à celui d'une compilation traditionnelle, mais ce code est accompagné des informations nécessaires à l'optimisation globale ultérieure et il contient des balises permettant des substitutions à l'issue de cette optimisation. Les techniques de compilation globale sont ensuite appliquées aux produits de la compilation séparée, avant l'édition de liens.

L'article est organisé comme suit. La section 2 décrit les avantages respectifs des compilations séparées et globales dans le contexte des langages à objets. La section suivante présente le schéma de compilation que nous proposons, en détaillant comment les techniques globales utilisées se projettent sur les deux phases locale et globale. La section 4 propose une comparaison avec SMART EIFFEL, décrit l'état d'avancement d'un démonstrateur et ouvre quelques perspectives.

2. Compilation globale et séparée

Contrairement à la compilation séparée, la compilation globale consiste à compiler globalement un ensemble d'unités de code source d'un programme, en connaissant son ou ses points d'entrée : éventuellement, cet ensemble d'unités peut faire appel à des unités extérieures mais l'inverse n'est pas possible. La question du caractère global ou séparé de la compilation concerne *a priori* tous les paradigmes de programmation, mais les spécificités des langages à objets rendent le problème plus crucial. Le principe même de l'envoi de message (ou liaison tardive) ne permet pas de savoir, dès la compilation, quelle méthode sera effectivement appelée. Le problème est à peu près le même pour les attributs mais il concerne alors leur localisation dans la structure de données de l'objet. Enfin, malgré le principe de typage fort, il est souvent néces-

saire de savoir si un objet est bien instance d'un certain type (coercition de type ou *downcast*), ce qui revient à un test de sous-typage.

2.1. *Avantages de la compilation globale*

La connaissance de la totalité du code d'un programme et de son point d'entrée rend possible des analyses fines sur la façon donc chaque unité du programme est utilisée par les autres, ce qui permet de la compiler plus efficacement. Parmi les nombreuses techniques d'optimisation nécessitant cette connaissance globale du programme, trois sont concernées par notre approche : l'analyse de types, la spécialisation de code et les techniques d'implémentation des objets et de l'envoi de messages. Les améliorations attendues concernent la réduction de la taille du programme et du polymorphisme des envois de message ainsi que du coût des trois mécanismes propres aux langages à objets. Le surcoût de l'héritage multiple peut ainsi être complètement éliminé.

2.1.1. *Analyse de types*

Le mécanisme d'envoi de messages est bien souvent le goulot d'étranglement des programmes à objets. Les statistiques montrent que la plupart des envois de messages sont en réalité des appels monomorphes : une analyse globale, souvent simple, permet de déterminer statiquement la méthode à appeler. Les langages proposent fréquemment des dispositifs permettant de déclarer qu'une méthode n'est pas soumise à la liaison tardive (`static` en JAVA, absence de `virtual` en C++) mais il serait plus sain de laisser le compilateur optimiser lui-même les envois de messages en détectant s'ils sont monomorphes et d'autre part s'ils peuvent en plus être mis en ligne.

L'analyse de types, dont un grand nombre de techniques est décrit dans [GRO 01], permet de détecter les appels monomorphes et de réduire le polymorphisme des autres appels. Elle consiste à approximer trois groupes d'ensembles : l'ensemble des classes effectivement instanciées, celui des types dynamiques que peut potentiellement prendre chaque expression (on l'appelle son *type concret*) et d'autre part l'ensemble des procédures potentiellement appelées pour chaque site d'envoi de messages. Ces trois groupes d'ensembles sont en dépendance circulaire, puisque les méthodes qui peuvent être appelées dépendent des types dynamiques du receveur, lesquels dépendent à leur tour des classes instanciées et ces dernières des méthodes effectivement appelées. Cette circularité explique la difficulté du problème [GIL 98] et la variété des solutions, toutes approximatives par excès. L'analyse de types permet donc de déterminer le *code vivant* — c'est-à-dire les classes instanciées ainsi que les méthodes appelées et les attributs utilisés, le reste constitue le *code mort*, qui n'a pas besoin d'être présent dans le programme final — tout en optimisant tout ce qui a trait au polymorphisme : envoi de message aussi bien que test de sous-typage.

2.1.2. *Spécialisation de code*

Ce terme générique désigne le fait de compiler un même bout de code de différentes façons de manière à l'adapter à chacune de ses utilisations particulières. On

peut regrouper sous ce terme la *customization* [CHA 89] qui consiste à compiler les méthodes, même héritées, de chaque classe de façon spécialisée, de sorte que `self` soit monomorphe, ainsi que l'implémentation *hétérogène* [ODE 97] des classes paramétrées afin d'adapter le code de la classe paramétrée à chacune des instanciations de ses paramètres. L'insertion du code en ligne (*inlining*), qui est possible pour les appels statiques, participe aussi de la spécialisation de code puisque le code inséré va être compilé en tenant compte du contexte de son insertion.

2.1.3. *Implémentation des objets et de l'envoi de message*

Deux types d'optimisations ont été considérées, parmi beaucoup. La première consiste à se ramener à l'héritage simple en typage statique, avec un accès direct aux attributs et des tables de méthodes nécessitant une unique indirection. La seconde permet de se passer de toute structure de données pour les méthodes.

2.1.3.1. Coloration

L'héritage multiple pose un problème d'implémentation, surtout en compilation séparée. C++ l'a résolu avec une implémentation par sous-objets, qui induit un surcoût important [LIP 96]. Dans le pire des cas, le nombre de tables de méthodes est quadratique dans le nombre de classes (au lieu de linéaire) et la taille cubique (au lieu de quadratique). Dans l'implémentation des objets, les pointeurs sur ces tables peuvent être plus nombreux que les attributs de l'objet. Enfin, une référence à un objet dépend du type statique de la référence. Le surcoût aussi bien spatial que temporel est très important [DUC 02a].

La coloration a été introduite par [DIX 89] : [DUC 02b] fait une synthèse des travaux qui s'y rapportent. La coloration permet de s'affranchir des surcoûts de l'héritage multiple en se ramenant à l'héritage simple dans les implémentations standard à base de tables de méthodes. Comme en héritage simple, l'implémentation des classes et des objets via la coloration (figure 1) comporte deux parties : en mémoire dynamique, une zone pour chaque instance, contenant ses attributs et un pointeur vers sa classe ; en mémoire statique, une zone pour la classe, contenant les adresses des méthodes et les identifiants des ancêtres de la classe (ces derniers sont présentés à part dans la figure).

La technique consiste à déterminer un identifiant par classe ainsi qu'une couleur par classe, par méthode et par attribut, de façon à garantir les trois invariants suivants, qui sont ceux des implémentations de l'héritage simple en typage statique :

Invariant 1 *Une référence à un objet ne dépend pas du type statique de la référence.*

Invariant 2 *Un attribut (resp. une méthode) possède un indice (une couleur) invariant par spécialisation. Deux attributs (resp. méthodes) de même couleur n'appartiennent pas à la même classe.*

Invariant 3 *Chaque classe possède un identifiant unique et un indice (une couleur). Deux classes de même couleur n'ont pas de sous-classe commune.*

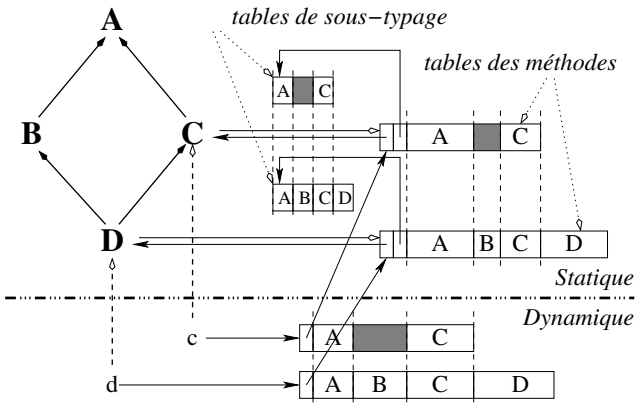

Figure 1. *Implémentation des classes et des objets via la coloration*

Le respect de ces invariants est aisé, mais la minimisation de la taille des tables — c'est-à-dire du nombre d'entrées non occupées de ces tables, en grisé sur la figure — est un problème proche de la coloration de graphes et prouvé récemment NP-difficile [TAK 03]. Heureusement, les hiérarchies de classes constituent apparemment des instances faciles du problème et de nombreuses heuristiques ont été proposées, qui donnent des résultats satisfaisants, même sur de très grosses hiérarchies [PUG 90, DUC 01, TAK 03].

2.1.3.2. Arbres de décision

A la suite de l'analyse de types, la connaissance du type concret du receveur sur un site d'envoi de message permet de générer un arbre de décision qui énumère les k cas possibles en faisant un appel direct à la méthode à activer : le nombre maximum de tests est $\lceil log_2(k) \rceil$. La technique est rendu efficace par l'architecture de *pipe-line* des processeurs actuels, car leur prédiction de branchement conditionnel leur permet d'annuler statistiquement le coût du test et du saut, alors que l'absence actuelle de prédiction de branchement indirect pénalise l'indirection par une table de méthodes [DRI 95, DRI 99].

2.2. *Limites de la compilation globale*

La compilation globale n'est pas utilisée dans le milieu industriel (C++, JAVA, ADA, C#, etc.) pour des raisons qui tiennent autant de l'histoire que d'inconvénients qui concernent aussi bien le développement que l'administration des programmes ou leur exécution. De plus, la confidentialité du code source est difficile à préserver, alors que la compilation séparée permet de ne diffuser que les unités compilées.

2.2.1. *Limites à l'administration et à l'exécution*

Le génie logiciel préconise la modularité et la réutilisabilité. Un programme complexe doit être découpé en unités plus simples, relativement indépendantes et réutilisables. La compilation séparée répond pleinement à ce principe en associant à chaque unité de code une unité compilée indépendante des autres unités compilées et des programmes dans lesquels cette unité de code sera mise en œuvre. Les unités étant compilées indépendamment de leurs utilisations, elles sont partageables par tous les programmes. Ce partage peut se faire soit lors de la construction d'un exécutable, à l'édition de liens, soit lors du chargement du programme, soit encore à l'exécution. Dans les deux derniers cas — on parle alors de *bibliothèques partagées* — le partage réduit la taille des exécutables et simplifie l'administration des systèmes puisque la mise à jour d'une bibliothèque ne nécessite pas celle de tous les exécutables qui s'en servent. Si le partage a lieu à l'exécution, la bibliothèque est chargée en mémoire lorsqu'un premier programme en a besoin et chaque programme ultérieur l'utilise. Ce partage à l'exécution accélère le chargement, et réduit la taille des programmes en mémoire.

Les techniques de compilation globale que nous avons présentées ne permettent *a priori* pas l'usage de bibliothèques partagées, dès lors que ces bibliothèques contiennent des éléments propres à la programmation par objets.

2.2.2. *Limites en développement*

Lors du développement d'un programme, le rôle d'un compilateur est double : vérification de la correction syntaxique et sémantique du programme et génération d'un exécutable pour le tester. En compilation séparée, le compilateur remplit ces deux rôles de façon satisfaisante : il vérifie la correction d'une unité de programme et produit un code compilé qui sera utilisé pour la génération d'un exécutable. En revanche, la compilation globale a le seul but de générer un exécutable : si la modification a porté sur du code mort, il n'y a pas de raison que le compilateur l'examine. Plus généralement, le code source peut être incorrect (d'après les spécifications du langage) mais celui d'une application correct, à cause de (ou grâce à) la spécialisation de code : le code source d'une classe paramétrée ou d'une méthode "customisée" peut être incorrect dans la classe d'origine mais correct dans le contexte où ses spécialisations ont été insérées.

Pour remédier à ce défaut, un compilateur global devrait être muni d'une fonctionnalité voisine d'un compilateur séparé, incluant au moins toutes les phases de la compilation qui peuvent générer des erreurs, jusqu'à la génération de code exclue.

Ce premier point étant acquis, le problème se pose de la génération d'un exécutable. Les cycles d'édition, de modification et de test d'une unité de code sont relativement courts et, à ce stade, l'objectif d'optimisation de la compilation globale est sans doute inutile voire nuisible, puisque l'optimisation est censée allonger la durée de la compilation. Mais la compilation globale peut paradoxalement très bien accélérer la compilation grâce à l'élimination du code mort. Comparer le temps de génération

d'un exécutable suivant le type de compilateur est donc un exercice difficile. Des expériences nombreuses et variées seraient nécessaires. Il est possible que la recompilation totale d'une application, à l'installation par exemple, soit plus rapide en compilation globale grâce à l'élimination du code mort. Mais cela peut dépendre de la proportion de code mort d'un côté, de l'utilisation de bibliothèques partagées de l'autre. Quant à la génération d'un exécutable après une petite modification, il est probable qu'un compilateur séparé sera en général plus rapide.

3. Schéma de compilation séparée avec optimisations globales

Le schéma de compilation proposé dans cet article met en œuvre les deux phases habituelles de la compilation séparée. La première est dite *locale* car son rôle est de compiler une unité de code indépendamment des programmes l'utilisant. La seconde phase est dite *globale* car elle assemble et adapte les unités compilées afin de construire un programme exécutable. Le code produit par la compilation séparée d'une unité contient des *symboles* permettant de l'accrocher aux unités utilisées [LEV 99]. Lors de l'édition de liens, les codes des différentes unités sont concaténés et les symboles sont remplacés par les adresses numériques.

Le schéma de compilation proposé ici diffère peu de ce schéma classique : la phase locale génère du code dans lequel les informations encore inconnues ne se réduisent pas à l'adresse de symboles, mais peuvent inclure des informations pour savoir s'il faut inclure une portion de code (code conditionnel) ou des symboles qui seront à remplacer par des valeurs (couleurs par exemple). Au total, il s'agit donc d'un code compilé normal, décoré de quelques balises.

3.1. *Phase locale*

Nous considérerons que la classe est l'unité de compilation. La phase locale (figure 2) prend en entrée le code source d'une classe et produit en sortie trois niveaux de code différents. Le *schéma* de la classe décrit son interface et ses super-classes directes, sans le code. Le *template* est la représentation de la circulation des types dans les méthodes de la classe. Le *code* correspond à la sortie habituelle du compilateur, dans le langage cible de la compilation (généralement du code machine). Ces trois parties peuvent être incluses dans le même fichier ou dans des fichiers distincts mais le schéma doit pouvoir être disponible séparément. On n'oubliera pas les différents messages d'erreur et d'avertissement qui font le charme des compilateurs et de la compilation.

3.1.1. *Classes et schémas de classes*

La compilation d'une unité est indépendante des autres unités jusqu'à un certain point seulement car elle doit connaître l'interface des classes dont l'unité est sous-classe ou cliente. Cette interface peut être disponible, soit dans le schéma de la classe

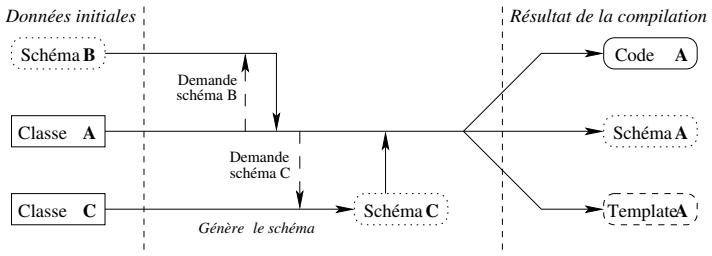

Figure 2. *Phase locale : compilation de la classe A*

cliente parce qu'elle a déjà été compilée ou parce que le schéma a été fourni séparément, soit dans le code source de la classe cliente. Dans ce dernier cas, une génération récursive du schéma sera nécessaire (figure 2).

Le schéma d'une classe peut être vu comme une instance du méta-modèle du langage. Son rôle est de fournir les briques élémentaires permettant à la phase globale de construire le méta-modèle complet d'un programme. Il contient au moins le nom de la classe, ceux de ses super-classes directes, ainsi que ses propriétés, décrites au minimum par leur nom, leur signature et le nom symbolique désignant le code à exécuter. Le schéma ressemble donc à la notion d'interface, mais il ne se limite pas à la partie non privée. Dans un langage comme EIFFEL, le schéma contient aussi les assertions. Pour des raisons d'optimisations, le schéma peut contenir des informations étendues comme par exemple les relations entre la classe et ses ancêtres (directs et indirects) ou l'ensemble des propriété connues de la classes, qu'elles soient héritées des ancêtres ou définies dans la classe, ainsi que leurs relations. Ce genre d'optimisation est intéressante dans les langages où l'héritage est complexe (comme par exemple EIFFEL via les clauses de redéfinition), mais aussi pour la compilation de programmes utilisant des hiérarchies profondes. Ce dernier point peut s'illustrer en C++ par la technique de précompilation d'en-têtes qui peut diminuer, dans des cas réels, de plus de la moitié la durée de compilation d'un programme.

3.1.2. *Analyses de types intra-classe*

Le *template* d'une classe est un graphe qui schématise la circulation des types dans les méthodes de la classe, entre leurs entrées et leurs sorties. Une entrée est le receveur, un paramètre, la lecture d'un attribut ou le résultat d'un appel de méthode et une sortie est le résultat de la méthode si c'est une fonction, l'écriture d'un attribut ou les arguments d'un appel de méthode. Les sommets du graphe sont les entrées, les sorties et des nœuds intermédiaires, et ses arcs sont des contraintes d'inclusion des types concrets associés à chaque sommet. Les sorties peuvent alors êtres vues comme des fonctions des entrées. Les instanciations de classes sont explicitées dans le template : si la méthode est vivante, les classes qu'elle instancie le sont aussi.

Une analyse de types intra-classe et intra-procédurale [PRI 02] permet de construire ces templates en minimisant leur taille. Pour réduire encore celle-ci, l'ana-

lyse peut être limitée aux types polymorphes, le type concret des types non polymorphes, en particulier primitifs, étant connu statiquement. Cette information compacte et limitée du comportement d'une classe sera utilisée lors de la phase globale pour effectuer les analyses de types.

3.1.3. *Génération du code*

Le principe de la génération du code est celui de n'importe quel compilateur, mais des balises sont insérées pour permettre de sauter le code des méthodes mortes, ainsi que pour les codes conditionnels générés pour tous les mécanismes où le polymorphisme est en jeu.

Pour le code mort, le code de chaque méthode est entouré de balises :

<IFALIVE *idcodemeth*>
  code de la méthode
</IFALIVE>

L'identifiant de la méthode est le nom symbolique désignant le code à exécuter. Lors de la compilation d'un site d'envoi de message, le compilateur ne sait pas si l'appel sera reconnu monomorphe, pas plus qu'il ne connaît la couleur de la méthode. Le compilateur génère donc deux codes conditionnels différents : un appel statique en cas de monomorphisme, et un accès dans la table de méthode, à la couleur de la méthode sinon. Pour le premier, comme l'analyse intra-classe préserve une correspondance entre les appels dans le code source et leur représentation dans les templates, il est possible d'étiqueter chaque appel dans le template et dans le code et de traiter tous les appels monomorphes détectés : mais la méthode à appeler ne sera connue qu'après la phase globale. Le deuxième cas est plus simple à traiter : la couleur de la méthode (Δ_m) est un invariant représenté par un symbole dans le code généré. La question d'en faire une valeur immédiate (sur un nombre réduit de bits) est à discuter. Au total, on obtient le code suivant, y compris les balises qui incluent le type statique du receveur et l'identifiant de l'appel :

<IFMONOMORPH *idmeth idtype idcall*>
  call *symbole à générer en phase globale*
<ELSE>
  load [*object* + `#tableOffset`] → *table*
  load [*table* + Δ_m] → *method*
  call *method*
</IFMONOMORPH>

L'identifiant de la méthode est l'identifiant de la propriété générique correspondante dans le méta-modèle du langage (par exemple, son nom, seul ou concaténé aux types des paramètres s'il y a de la surcharge statique).

L'accès aux attributs s'effectue par un accès direct dans l'objet, à la couleur de l'attribut (Δ_a), par exemple pour une lecture :

<RATTRIBUTE *idattr*>
  load [*object* + Δ_a] → *val*
</RATTRIBUTE>

Quant au test de type, il s'effectue comme le traitement des méthodes : le test sera peut-être toujours ou jamais vérifié par les types concrets, auquel cas seul un saut est nécessaire. Pour le cas général, l'identifiant du type cible (Id_c) est comparé à la valeur trouvée dans la table des méthodes, à la couleur de ce type (Δ_c) :

<IFSUBTYPE *idtype idtest*>
  load [*objet* + `#tableOffset`] → *table*
  load [*table* + Δ_c] → *class*
  cmp *class*, Id_c
  jne *false*
<TRUE>
  ... //*test réussi*
  jmp *end*
<FALSE>
false : ... //*échec*
end :
</IFSUBTYPE>

Lors de l'instanciation d'une classe, un objet doit être construit. Il est représenté en mémoire par un espace délimité contenant d'une part le pointeur vers la table des méthodes à la position `#tableOffset` et d'autre part les attributs. La partie de l'instanciation qui réserve l'espace mémoire et place le pointeur de table doit donc réserver un espace correspondant à la couleur maximum de la classe et écrire l'adresse de la table à la position `#tableOffset`.

3.2. *Phase globale*

La phase globale comporte trois étapes : l'analyse de types qui détermine le code vivant, la coloration et la transformation du code (figure 3).

3.2.1. *Analyse de types inter-classes*

L'analyse de types inter-classes se base sur l'ensemble des schémas et des templates des unités. Dans les techniques d'analyse avec flux, les templates sont liés entre eux en connectant leurs entrées et sorties de façon à constituer un réseau global de contraintes d'inclusion de types. En partant du point d'entrée du programme, les classes vivantes et leurs méthodes vivantes sont identifiées, ainsi que le type concret de toutes leurs expressions, c'est-à-dire de tous les sommets du réseau. Les classes mortes ne sont pas examinées et les méthodes mortes sont enlevées des schémas des classes, de même que les attributs non utilisés en lecture.

3.2.2. *Coloration*

La coloration s'applique au schéma global du programme, constitué des schémas des classes vivantes, restreints à leurs attributs et méthodes vivants, et pour ces dernières aux méthodes utilisées de façon polymorphe. Une heuristique de coloration est appliquée sur ces schémas restreints et produit les valeurs des identifiants et couleurs

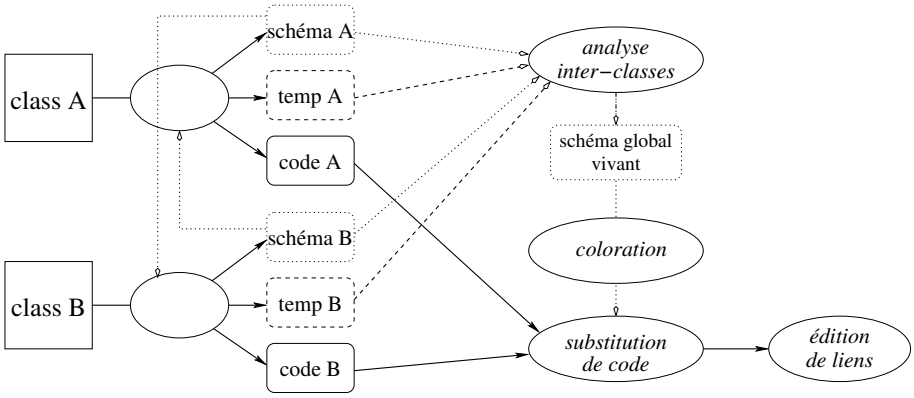

Figure 3. *Phase globale*

de classes, méthodes et attributs, ainsi que la taille des tables, c'est-à-dire les valeurs qu'il faudra substituer aux symboles correspondants.

3.2.3. *Substitutions dans le code*

Une fois les analyses terminées, la substitution dans le code est proche de celle utilisée pour la résolution des adresses par l'éditeur de liens puisqu'il s'agit de données numériques dans les deux cas. Le code compilé de chaque classe vivante est examiné séquentiellement : ni mémorisation ni retour en arrière ne sont nécessaires. Les balises <IFALIVE> permettent de sauter le code mort et seul le code vivant est substitué et écrit sur le canal de sortie. Pour chaque site d'appel de méthodes, l'ensemble des routines potentiellement appelées est connu. Suivant qu'il s'agit d'un singleton ou pas, l'une des deux parties des balises <IFMONOMORPH> est sautée. De plus, si l'appel est monomorphe, le symbole correspondant à la méthode appelée est généré, alors que dans le cas polymorphe, c'est la couleur de la méthode qui est insérée. Le traitement des accès aux attributs ou des tests de sous-type est similaire. Le code produit par la substitution peut être directement transmis à l'édition de liens.

4. Comparaisons, mise en œuvre et perspectives

L'idée de calculer la coloration à l'édition de liens avait déjà été avancée par [PUG 90] mais il ne semble pas qu'elle ait jamais été appliquée. Le terme de *template* a été introduit par [AGE 96] pour désigner le réseau de contraintes associé à une procédure dans les analyses poly-variantes : mais les phases d'analyse intra et inter-procédurales ne sont pas explicitement distinguées. [BOU 99] propose une architecture de compilateur de langages fonctionnels assez proche de la nôtre, où l'optimisation globale est restreinte à l'analyse de types : la compilation séparée produit du code dans un langage intermédiaire, ce qui nécessite une compilation de la totalité du programme en phase globale, après l'optimisation.

4.1. *Comparaisons :* SMART EIFFEL

Le compilateur GNU EIFFEL, SMART EIFFEL [COL 97, ZEN 97], est caractéristique des approches récentes à base de techniques globales et il s'agit sans doute de l'une de leurs premières applications à un langage à objets à typage statique. Il est basé sur l'analyse de types, la spécialisation de code et l'implémentation de l'envoi de messages par arbres de décision, sans tables de méthodes. Le langage cible de la compilation est C, chaque classe donnant lieu à un fichier C. Il faut donc ensuite compiler les fichiers C et faire une édition de liens classique. Les performances affichées par la code généré par SMART EIFFEL sont très significativement meilleures que les compilateurs EIFFEL précédents, aussi bien en matière d'espace que de temps. Le fait de passer par l'intermédiaire de C a des effets importants sur les performances car les fichiers C ne sont recompilés que s'ils ont été modifiés. Dans les bons cas de figure, la recompilation partielle après une petite modification est assez rapide car seul le fichier C correspondant a été modifié. Mais dans le pire des cas, une petite modification peut rendre vivant ou mort une partie du code et impliquer la recompilation d'un grand nombre de fichiers. Cependant, il manque encore au compilateur un outil de vérification complet de la correction du code, indépendamment de son utilisation.

4.2. *Mise en œuvre et résultats*

Afin de vérifier la pertinence de ce modèle de compilation et de le tester dans des cas de compilation réels, nous sommes en train de développer un compilateur EIFFEL. Le choix d'EIFFEL s'explique aisément : C++ est exclu, car son implémentation fait quasiment partie des spécifications. JAVA et C# posent un problème analogue par leur machine virtuelle ; ils sont de plus en héritage simple et sous-typage multiple, ce qui réduit un peu l'avantage de la coloration. Parmi les langages à objets à typage statique utilisés relativement massivement, il ne reste plus qu'EIFFEL. Par ailleurs, le projet est rendu matériellement possible par la réutilisation du code du compilateur SMART EIFFEL. Un méta-modèle du langage EIFFEL a été développé et l'analyse intra-classe est réalisée : schémas et templates de classes sont donc disponibles. Les heuristiques de coloration ont également été développées. Il reste à développer l'analyse inter-classes, en reprenant une des techniques classiques [GRO 01] et à adapter le générateur de code de SMART EIFFEL pour lui faire générer du code C décoré par des balises. Pour ces balises, on utilisera la fonctionnalité propre à GCC, avec l'instruction `asm`, permettant l'insertion d'assembleur dans le code C. GCC ne vérifiant pas le code ainsi inséré, l'insertion de balises ne pose aucun problème. Au total, les performances attendues sont raisonnables. D'une part, le surcoût sur la phase locale (analyse intra-classe et insertion des balises) est très faible. D'autre part, malgré la complexité théorique des problèmes associés, les heuristiques de l'analyse inter-classes et de la coloration augmenteront de très peu la durée de l'édition de liens : SMART EIFFEL l'a montré pour l'analyse de types et nous l'avons vérifié pour la coloration sur des hiérarchies de classes gigantesques (quelques secondes pour plus de 8000 classes) [DUC 03].

4.3. *Perspectives*

Ultérieurement, plusieurs améliorations pourraient être apportées à ce schéma de compilation : la mise en ligne des appels monomorphes et les bibliothèques partagées.

4.3.1. *Mise en ligne des appels monomorphes*

Il y a intérêt à mettre en ligne les procédures simples. Le surcoût de leur duplication est compensé par l'absence d'appel avec tout ce que cela implique : sauvegarde du contexte, saut long, retour et restauration du contexte. Plusieurs types de procédures simples ont été identifiées [ZEN 97] : les méthodes ne faisant rien, retournant une constante, le receveur ou un paramètre, les accesseurs d'attributs ou les méthodes appelant uniquement une autre méthode sur l'un de leurs paramètres. La mise en ligne ne serait pas très compliquée dans notre schéma : elle nécessiterait juste des balises pour indiquer qu'une méthode peut être mise en ligne, et un peu de chirurgie pour recoller les morceaux. Son principal inconvénient est sans doute qu'elle nécessite une phase globale de substitution avec mémorisation, en deux passes.

4.3.2. *Bibliothèques partagées*

Notre modèle de compilation n'est *a priori* pas plus compatible que SMART EIFFEL avec l'utilisation de bibliothèques partagées de classes EIFFEL. Plusieurs directions sont tout de même envisageables. On peut d'abord considérer une bibliothèque compilée en totalité, sans code mort, ni détection des appels monomorphes. La seule spécificité de notre approche est alors la coloration. La solution la plus grossière consiste à faire la coloration et l'édition de liens au chargement. Une seconde solution utilise des tables d'indirections : cette voie s'inspire des techniques utilisées pour les bibliothèques dynamiques des langages procéduraux, les valeurs de la coloration seraient stockées dans les données des processus plutôt que dans le code. Ainsi le code serait partageable par tous les processus et ne nécessiterait pas de modification au chargement (et *a fortiori* de modification du système d'exploitation). Une troisième solution consiste à effectuer la coloration lors de la compilation de la bibliothèque : les couleurs sont alors associées au schéma des classes. On peut alors, soit limiter le développement pour empêcher les conflits, soit les résoudre par un arbre de décision, suivant la technique décrite dans [VIT 94, DUC 97]. La technique s'applique parfaitement aux méthodes. Pour les attributs, l'indirection nécessaire peut être obtenue par la technique de simulation des accesseurs qui peut remplacer la coloration d'attribut dans le cadre de la coloration de méthodes [DUC 02b]. Dans ce cas, des balises intégreraient facilement la double compilation proposée par [MYE 95]. Le test de sous-typage nécessiterait une adaptation.

5. Conclusion

Nous avons présenté dans cet article un schéma de compilation séparée dans lequel sont intégrées deux techniques globales d'implémentation et d'optimisation : l'analyse de types et la coloration.

Par rapport à une compilation séparée classique, à la C++, deux grandes améliorations sont à attendre de cette approche. Le gain en espace mémoire devrait être très significatif, par l'élimination du code mort et l'adoption de la coloration qui réduit d'un ordre de grandeur la taille des tables de méthodes, ainsi que la longueur des séquences d'appel. Le gain en temps devrait aussi être important, grâce à l'abandon des sous-objets, qui imposent des ajustements de pointeurs permanents, et à la détection des appels monomorphes, forcément très nombreux.

Par rapport à la stratégie adoptée par SMART EIFFEL, notre proposition devrait être plus modeste : la qualité de l'analyse de types peut être identique dans les deux approches, mais il est probable que SMART EIFFEL devrait obtenir des performances temporelles un peu meilleures, grâce à son usage de la spécialisation de code. En revanche, notre approche ne souffre pas du caractère insuffisant de la vérification du code des classes, qui est le lot actuel de SMART EIFFEL et qui nécessiterait pour lui le développement d'un outil particulier.

Par ailleurs, et c'est vrai aussi de SMART EIFFEL, les approches de compilation globale enlèvent toute justification aux deux usages du mot-clé `virtual` en C++ : le surcoût de l'héritage multiple est à peu près annulé, comme celui de la liaison tardive lorsqu'elle n'est pas utilisée.

Remerciements

Les auteurs remercient Dominique Colnet et Olivier Zendra pour les avoir guidés dans le labyrinthe de SMART EIFFEL.

6. Bibliographie

[AGE 96] AGESEN O., « Concrete Type Inference : Delivering Object-Oriented Applications », PhD thesis, Stanford University, 1996.

[BOU 99] BOUCHER D., « Analyse et Optimisations Globales de Modules Compilés Séparément », PhD thesis, Université de Montréal, 1999.

[CHA 89] CHAMBERS C., UNGAR D., « Customization : Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Language », *Proc. OOPSLA'89*, ACM Press, 1989, p. 146–160.

[COL 97] COLLIN S., COLNET D., ZENDRA O., « Type inference for Late Binding. The SmallEiffel Compiler », *Joint Modular Languages Conference*, LNCS 1204, Springer Verlag, 1997, p. 67–81.

[DIX 89] DIXON R., MCKEE T., SCHWEITZER P., VAUGHAN M., « A fast method dispatcher for compiled languages with multiple inheritance », *Proc. OOPSLA'89*, ACM Press, 1989.

[DRI 95] DRIESEN K., HÖLZLE U., VITEK J., « Message Dispatch on Pipelined Processors », OLTHOFF W., Ed., *Proc. ECOOP'95*, LNCS 952, Springer-Verlag, 1995, p. 253–282.

[DRI 99] DRIESEN K., « Software and Hardware Techniques for Efficient Polymorphic Calls », PhD thesis, University of California, Santa Barbara, 1999.

[DUC 97] DUCOURNAU R., « La compilation de l'envoi de message dans les langages dynamiques », *L'Objet*, vol. 3, n° 3, 1997, p. 241–276.

[DUC 01] DUCOURNAU R., « La coloration : une technique pour l'implémentation des langages à objets à typage statique. I. La coloration de classes », Rapport de Recherche n° 01-225, 2001, L.I.R.M.M.

[DUC 02a] DUCOURNAU R., « Implementing Statically Typed Object-Oriented Programming Languages », Rapport de Recherche n° 02-174, 2002, L.I.R.M.M.

[DUC 02b] DUCOURNAU R., « La coloration pour l'implémentation des langages à objets à typage statique », DAO M., HUCHARD M., Eds., *Actes LMO'2002* in *L'Objet vol. 8*, Hermès, 2002, p. 79–98.

[DUC 03] DUCOURNAU R., « La coloration : une technique pour l'implémentation des langages à objets à typage statique. II. La coloration de méthodes et d'attributs », Rapport de Recherche n° 03-xxx, 2003, L.I.R.M.M.

[GIL 98] GIL J., ITAI A., « The Complexity of Type Analysis of Object Oriented Programs », *Proc. ECOOP'98*, LNCS 1445, Springer-Verlag, 1998, p. 601–634.

[GRO 01] GROVE D., CHAMBERS C., « A Framework for Call Graph Construction Algorithms », *ACM Trans. Program. Lang. Syst.*, vol. 23, n° 6, 2001, p. 685–746.

[LEV 99] LEVINE J. R., *Linkers & Loaders*, Morgan Kaufmann, 1999.

[LIP 96] LIPPMAN S., *Inside the C++ Object Model*, Addison-Wesley, New York, 1996.

[MYE 95] MYERS A., « Bidirectional Object Layout for Separate Compilation », *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), ACM Press, 1995, p. 124–139.

[ODE 97] ODERSKY M., WADLER P., « Pizza into Java : Translating Theory into Practice », *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, ACM Press, New York (NY), USA, 1997, p. 146–159.

[PRI 02] PRIVAT J., « Analyse de types et graphe d'appels en compilation séparée », Mémoire de DEA, Université Montpellier II, 2002.

[PUG 90] PUGH W., WEDDELL G., « Two-directional record layout for multiple inheritance », *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'90)*, ACM SIGPLAN Notices, 25(6), 1990, p. 85–91.

[TAK 03] TAKHEDMIT P., « Coloration de classes et de propriétés : étude algorithmique et heuristique », Mémoire de DEA, Université Montpellier II, 2003.

[VIT 94] VITEK J., HORSPOOL R., « Taming message passing : efficient method look-up for dynamically typed languages », TOKORO M., PARESCHI R., Eds., *Proc. ECOOP'94*, LNCS 821, 1994, p. 432–449.

[ZEN 97] ZENDRA O., COLNET D., COLLIN S., « Efficient Dynamic Dispatch without Virtual Function Tables : The SmallEiffel Compiler », *Proceedings of OOPSLA'97, Atlanta (GA), USA*, special issue of ACM SIGPLAN Notices, 32(10), 1997, p. 125–141.

[ZIB 02] ZIBIN Y., GIL J., « Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching », *Proc. OOPSLA'02*, SIGPLAN Notices, 37(10), ACM Press, 2002.