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

Jean Privat — Roland Ducournau

LIRMM – Université Montpellier II
161 rue Ada
F-34392 Montpellier cedex 5
{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 separate 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. Before linking, a global step makes all global computations and substitutes computed values for symbols in 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, héritage multiple.*

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

MCours.com

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, COL 97, ZEN 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 mécanismes objet spécifiques.

Cet article propose un schéma complet qui tente de concilier ces deux points de vue sur la compilation. Nous nous plaçons dans un cadre de typage statique, en visant des langages comme EIFFEL et C++ et en excluant les langages à chargement dynamique comme JAVA ou C#. Pour implémenter les objets et leurs mécanismes spécifiques, nous adoptons la coloration [DIX 89, PUG 90, DUC 02b] qui nous semble le meilleur choix. L'analyse de types est décomposée 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 globales (analyse de types et coloration) sont ensuite appliquées aux produits de la compilation séparée, avant l'édition de liens. Cette approche en deux temps n'est pas strictement nouvelle : elle a été suggérée par [PUG 90] pour la coloration, développée par [FER 95] pour MODULA 3 et appliquée par [BOU 99] pour SCHEME.

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

2. Compilation globale et séparée des langages à objets

La question du caractère global ou séparé de la compilation concerne *a priori* tous les paradigmes de programmation, mais le polymorphisme des langages à objets rend le problème 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 s'agit alors de leur

localisation dans la structure de données de l'objet. Enfin, malgré le principe de typage fort, il est souvent nécessaire 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 dont chaque unité du programme est utilisée par les autres, ce qui permet de la compiler plus efficacement. Les améliorations attendues concernent la réduction de la taille du programme, du polymorphisme des envois de message et du coût des trois mécanismes propres aux langages à objets. Le surcoût de l'héritage multiple peut aussi être complètement éliminé.

2.1.1. *Analyse de types*

Le mécanisme d'envoi de messages est considéré comme 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, permettrait 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 un compilateur optimisé devrait permettre de s'en passer.

L'analyse de types, dont un grand nombre de techniques est décrit dans [GRO 01] et qu'il ne faut pas confondre avec l'inférence de types à la ML, 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 pourra prendre chaque expression dans toutes les exécutions possibles (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.

2.1.2. *Implémentation des objets et de l'envoi de message par la coloration*

La coloration s'inscrit dans le cadre des techniques d'implémentation avec tables de méthodes. L'implémentation des classes et des objets 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. Dans ce cadre, l'héritage multiple pose un problème, surtout

en compilation séparée. C++ l'a résolu avec une implémentation par sous-objets, qui induit un surcoût important [LIP 96, DUC 02a]. 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.

La coloration [DIX 89, PUG 90, COH 91, VIT 97, DUC 02b] permet de s'affranchir des surcoûts de l'héritage multiple en se ramenant à l'implémentation naturelle en héritage simple et typage statique. Elle s'applique aussi bien aux attributs et aux méthodes, qu'aux classes pour le test de sous-typage. La technique consiste à déterminer un identifiant par classe ainsi qu'une couleur par classe, méthode et attribut, de façon à garantir les invariants 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.* En corollaire, l'affectation ou le passage de paramètre polymorphe ne coûte rien.

Invariant 2 *Un attribut (resp. une méthode) possède une couleur, sa position, invariante par spécialisation. Deux attributs (resp. méthodes) de même couleur n'appartiennent pas à la même classe.* En corollaire, la position de la méthode ou de l'attribut ne dépend pas du type dynamique du receveur.

Invariant 3 *Chaque classe possède un identifiant unique et une couleur. Deux classes de même couleur n'ont pas de sous-classe commune.* La table de méthodes d'une classe contient, à la couleur de chaque super-classe, son identifiant. Vérifier que D est une sous-classe de C revient simplement à vérifier que $tab_D[k_C] = id_C$, où tab est la table de méthodes, k la couleur et id l'identifiant.

La figure 1 schématise cette implémentation, en séparant la table des couleurs de classes de la table des méthodes, par clarté. 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, DUC 03, TAK 03].

2.2. *Limites de la compilation globale*

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

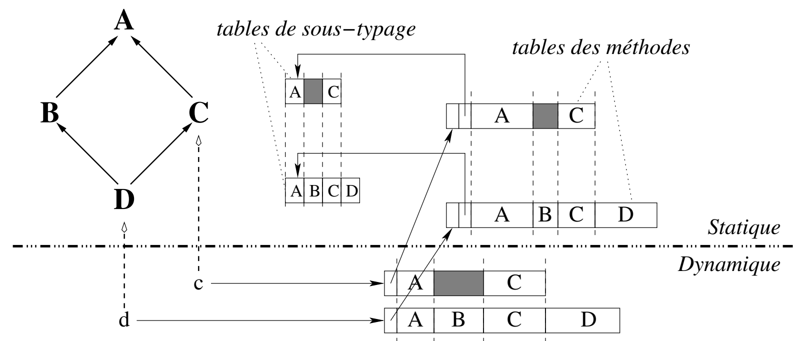

Figure 1. *Implémentation des classes et des objets via la coloration*

Au cours du développement, le rôle d'un compilateur est double : vérifier la correction syntaxique et sémantique du programme et générer un exécutable pour le tester. Deux points sont donc à considérer : la qualité de cette vérification et le temps de génération. En compilation séparée, le compilateur remplit ces deux rôles de façon satisfaisante. En revanche, la compilation globale a le seul but de générer un exécutable : il n'y a pas de raison que le compilateur examine le code mort. 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, qui consiste à traiter l'héritage ou la généricité par copie 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 donc 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.

En ce qui concerne le temps de compilation et de génération d'un exécutable, il est difficile de se prononcer sans un véritable banc d'essai mesurant la différence entre le temps passé aux optimisations et celui gagné en ne compilant pas le code mort ou comparant l'impact sur les compilations incrémentales liées aux cycles relativement courts d'édition, de modification et de test d'une unité de code.

Par ailleurs, la compilation globale semble difficilement compatible avec les bibliothèques dynamiques. En effet, celles-ci sont partagées au chargement ou à l'exécution afin de permettre une facilité d'administration et d'évolution des programmes.

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). 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 prend en entrée le code source d'une classe et produit en sortie trois niveaux de code différents (figure 2). Le *schéma externe* de la classe décrit son interface et ses super-classes directes, sans le code. Le *schéma interne* 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 externe doit pouvoir être disponible séparément. On n'oubliera pas les différents messages d'erreur et d'avertissement qui font le charme de la compilation.

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

Le schéma externe 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 externe 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 externe contient aussi les assertions. Pour des raisons d'optimisations, il peut contenir des informations étendues à ses super-classes directes et indirectes. Ce genre d'optimisation est intéressant 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.

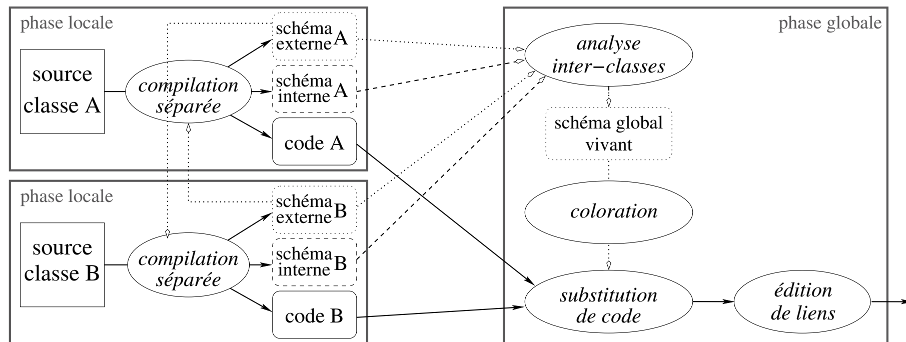

Figure 2. *Phases locales (à gauche) et globale (à droite), et flux d'information*

3.1.2. *Analyse de types et schéma interne*

Le *schéma interne* d'une classe est un graphe qui représente 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 : [BOU 99] en donne d'ailleurs une forme fonctionnelle. Les instanciations de classes sont explicitées dans le schéma interne : si la méthode est vivante, les classes qu'elle instancie le sont aussi. Une analyse de types intraclasse et intraprocédurale [PRI 02] permet de construire ces schémas internes en minimisant leur taille. Pour réduire encore celle-ci, l'analyse peut être limitée aux types polymorphes, le type concret des types non polymorphes, en particulier primitifs, étant connu statiquement. On notera qu'il s'agit de la partie la plus simple de l'analyse de types, qui est passée sous silence dans quasiment toute la littérature.

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. Dans la suite, nous utilisons des balises à la HTML et nous empruntons le pseudo-code à [DRI 99]. Les ⟦ ⟧ désignent les parties à substituer.

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 (*idcodemeth*) 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

MCours.com

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 intraclasse préserve une correspondance entre les appels dans le code source et leur représentation dans les schéma internes, il est possible d'étiqueter chaque appel dans le schéma interne 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 :

```
<IFMMORPH idmeth idtype idcall>   call ⟦symbole à générer en phase globale⟧
<ELSE>                            load [object + #tableOffset] → table
                                  load [table + ⟦Δm⟧ ] → method
                                  call method                </IFMMORPH>
```

L'identifiant de la méthode (*idmeth*) 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 :

```
<RATTR idattr>        load [object + ⟦Δa⟧ ] → val        </RATTR>
```

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 une branche est à conserver. 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 + ⟦Δc⟧ ] → class
                            cmp class, ⟦Idc⟧
                            jne §false
<TRUE>                      code test réussi
                            jmp §end
<FALSE>                     §false : code 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. Le code de l'instanciation 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 2). L'analyse de types interclasse se base sur l'ensemble des schémas internes et externes des unités. Dans les

techniques d'analyse avec flux, les schémas internes sont liés entre eux, éventuellement avec duplication pour tenir compte des contextes d'appel, 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 et attributs vivants sont incorporés au réseau et au schéma global au fur et à mesure de la détection de leur caractère vivant.

On obtient un schéma global vivant, accompagné d'informations sur les types concrets des sites d'appel vivants. La coloration s'applique à ce schéma global vivant, restreint, en ce qui concerne les méthodes, à celles qui sont utilisées de façon polymorphe. Une heuristique de coloration produit les valeurs des identifiants et couleurs de classes, méthodes et attributs, ainsi que la taille des tables, c'est-à-dire les valeurs qu'il faudra substituer aux symboles correspondants.

Une fois les étapes précédentes effectuées, 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 procédures potentiellement appelées est connu. Suivant qu'il s'agit d'un singleton ou pas, l'une des deux parties des balises <IFMMORPH> 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. Enfin, le code nécessaire à la génération des tables de méthodes doit être généré. Le code produit par la substitution peut ensuite 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. Il n'est d'ailleurs pas certain que la coloration ait été pleinement utilisée. La notion de *schéma interne* correspond à celle de *template* introduite par [AGE 96]. Mais les analyses intra et interprocédurales n'y étaient pas séparées dans le temps.

[FER 95] et [BOU 99] proposent une architecture assez proche de la nôtre, respectivement pour MODULA 3 (mais sans élimination du code mort) et pour les langages fonctionnels (dans le but de supprimer les tests de types et d'optimiser l'utilisation des variables de type fonctionnel). Dans les deux cas, la différence principale réside dans la compilation séparée qui produit du code dans un langage intermédiaire, ce qui nécessite une compilation plus ou moins légère 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. Outre l'analyse de types, il est basé sur la spécialisation de code et l'implémentation des mécanismes objet par arbres de décision, sans tables de méthodes. La spécialisation de code consiste à compiler différemment le même bout de code suivant le contexte dans lequel il s'insère : la *customization* [CHA 89], par exemple, traite l'héritage comme de la copie de code pour compiler les méthodes de chaque classe de façon spécialisée, de sorte que `self` soit toujours monomorphe. Quant aux arbres de décision, ils consistent à compiler chaque mécanisme objet polymorphe par une série de tests qui énumèrent tous les types possible dans le type concret du receveur. Le mécanisme n'est plus en temps constant car le nombre maximum de tests est $\lceil log_2(k) \rceil$ si le type concret est de cardinal k. La technique est rendue efficace par l'architecture de *pipeline* des processeurs d'aujourd'hui, 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 99]. Il faut noter que les arbres de décision et la coloration présentent le même avantage de fournir une solution uniforme aux trois mécanismes objet. Il serait envisageable de remplacer la coloration par les arbres de décision dans notre approche, mais cela imposerait de la génération de code, même simple, au cours de la phase globale.

Le langage cible de la compilation est 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 après une modification est très rapide car seul le fichier C correspondant a été modifié. Mais dans le pire des cas, une petite modification peut rendre vivant ou mort de grandes quantités de méthodes en impliquant la recompilation de nombreux fichiers. 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 sur des applications réelles, nous sommes en train de développer un prototype de compilateur EIFFEL. Le choix d'EIFFEL s'explique aisément : C++ est exclu, car son implémentation fait quasiment partie des spécifications et que sa complexité le rend peu adapté à la réalisation d'un démonstrateur. JAVA et C# opposent la même première objection par leur machine virtuelle ; ils sont de plus en héritage simple et sous-typage multiple, ce qui réduirait un peu l'avantage de la coloration ; mais surtout, les op-

timisations globales sont incompatibles avec leur chargement dynamique. Parmi les langages à objets à typage statique utilisés relativement massivement, il ne reste guère 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 élaboré et l'analyse intraclasse est réalisée : les schémas externes et internes des classes sont donc disponibles. Les heuristiques de coloration ont également été développées. Il reste à implémenter l'analyse interclasses, en reprenant une ou plusieurs 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 l'instruction `asm`, propre à GCC, 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 mais, comme le compilateur est en cours de développement, seules des extrapolations nous permettent d'étayer cette affirmation. Ces extrapolations sont de deux ordres : l'expérience de SMART EIFFEL et les simulations préliminaires effectuées sur les implémentations à la C++ [DUC 02a] et sur la coloration [DUC 01, DUC 03, TAK 03]. D'une part, le surcoût sur la phase locale (analyse intraclasse et insertion des balises) est très faible : l'approche ne doit avoir aucune influence significative sur le temps de la compilation séparée elle-même. Malgré la complexité théorique des problèmes associés, la phase globale ne devrait pas non plus changer l'ordre de grandeur du temps de génération d'un exécutable. Nous avons aussi montré que la coloration ne nécessite que quelques secondes pour des hiérarchies de plusieurs milliers de classes, donc beaucoup moins que l'édition de liens correspondante. De son côté, SMART EIFFEL a fait la preuve de la faisabilité de l'analyse de types. Enfin, la substitution de code relève de procédés élémentaires pour un éditeur de liens. Comparé à un schéma de compilation à la C++, la réduction du temps d'édition de liens due à l'élimination du code mort pourrait plus que compenser la phase globale. Concernant l'exécutable, il est certain que la taille de la mémoire statique nécessaire sera bien inférieure à celle d'une implémentation à la C++ : pour de très gros benchmarks, le rapport des tailles des tables de méthodes peut dépasser 5 — sous réserve d'utiliser du C++ purement objet avec les deux usages du mot clé `virtual`. La comparaison avec SMART EIFFEL est plus incertaine, la spécialisation de code compensant l'absence de tables de méthodes. En ce qui concerne l'exécution, notre approche devrait produire un résultat intermédiaire entre les deux autres, autant en mémoire dynamique qu'en temps.

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.

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 et restauration du contexte, sauts). Plusieurs types de procédures simples ont été identi-

fiées [ZEN 97]. Leur 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.

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 d'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 [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'attributs dans le cadre de la coloration de méthodes [DUC 02b]. Dans ce cas, des balises intégreraient facilement la double compilation proposée par [MYE 95]. Il resterait à trouver une adaptation pour le test de sous-typage.

Enfin, nous avons considéré que l'unité de compilation était la classe : une unité de compilation plus grande, similaire à un module, serait à considérer dans la mesure où elle serait munie de règles de portée ou de protection qui permettrait d'en déduire des optimisations locales particulières.

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. Le gain espéré de cette approche est d'apporter le bénéfice des optimisations globales aux langages compilés séparément, à la C++, sans leur faire perdre les avantages de la compilation séparée et tout en conservant des performances honorables par rapport à une compilation purement globale.

Par rapport à une compilation séparée classique, deux grandes améliorations sont à attendre. 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 de-

vrait 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, les avantages sont forcément plus modestes : 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. De plus, les arbres de décision garderont l'avantage sur les tables de méthodes tant que les processeurs ne seront pas dotés de fonctionnalités avancées de prédiction de saut indirect [DRI 99]. 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, relativement redondant. 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.

L'un des objectifs de ce travail est d'annuler le surcoût de l'héritage multiple. L'application à des langages avec héritage simple, éventuellement accompagné de sous-typage multiple à la JAVA, resterait un objectif intéressant, l'analyse de types apportant déjà beaucoup. En revanche, l'approche est grandement incompatible avec un chargement dynamique.

Remerciements

Les auteurs remercient Dominique Colnet et Olivier Zendra pour les avoir guidés dans le labyrinthe de SMART EIFFEL et pour de nombreuses discussions sur les mérites respectifs des compilations globale ou séparée, Jacques Malenfant pour leur avoir signalé les travaux de Dominique Boucher, ainsi qu'un relecteur anonyme pour leur avoir soufflé les termes de *schémas interne* et *externe*.

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.

[COH 91] COHEN N., « Type-extension type tests can be performed in constant time », *Programming languages and systems*, vol. 13, n° 4, 1991, p. 626–629.

[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 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., Montpellier.

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

[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° 03-036, 2003, L.I.R.M.M., Montpellier.

[FER 95] FERNANDEZ M. F., « Simple and Effective Link-Time Optimization of Modula-3 Programs », *SIGPLAN Conference on Programming Language Design and Implementation*, 1995, p. 103–115.

[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 and Loaders*, Morgan-Kauffman, October 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.

[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 97] VITEK J., HORSPOOL R., KRALL A., « Efficient Type Inclusion Tests », *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), ACM Press, 1997, p. 142–157.

[ZEN 97] ZENDRA O., COLNET D., COLLIN S., « Efficient Dynamic Dispatch without Virtual Function Tables : The SmallEiffel Compiler », *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), ACM Press, 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.

MCours.com