La compilation de l'envoi de message dans les langages à objets à typage dynamique

Roland Ducournau
LIRMM – Université Montpellier II
161, rue Ada – 34392 Montpellier cedex 5
ducournau@lirmm.fr

Résumé. La compilation de l'envoi de message, c'est-à-dire de la sélection de la méthode spécifique à appliquer, suivant le type de son argument, est la clé des performances des langages d'objets. Depuis quelques années, plusieurs techniques de tables ont été développées en vue d'obtenir une sélection en temps constant pour les langages typés dynamiquement. Cependant, malgré un compactage quasi optimal, ces tables restent d'une taille d'autant plus prohibitive que la taille des grosses bibliothèques de classes ne cesse d'augmenter. De plus, ces techniques se prêtent assez mal aux modifications incrémentales qui sont la raison même des langages à typage dynamique. Cet article présente la problématique de la compilation de l'envoi de message dans ces langages dynamiques, les diverses techniques en lice et propose une solution de synthèse.

Mots-clefs : programmation par objets, envoi de message, sélection de méthodes.

1 Introduction

Dans les langages de programmation par objets, la clé des performances réside dans la bonne compilation de l'envoi de message, c'est-à-dire de la sélection de la méthode spécifique à appliquer suivant le type de son argument, le *receveur*. Lorsque le typage est dynamique, comme en SMALLTALK, la sélection de la méthode est souvent décrite comme une recherche, appelée *lookup*, partant de la classe du receveur pour se terminer lors de la rencontre de la première de ses super-classes qui possède une méthode pour le *sélecteur* — c'est-à-dire le « nom » du message ou de la méthode — cherché. L'envoi de message se décrit alors par deux fonctions, `send` et `lookup`. L'échec de la recherche se traduit par une erreur `message inconnu` (gérée par une fonction `unknown-method`) qui peut être traitée, comme en SMALLTALK, par un envoi de message particulier. Les fonctions génériques de CLOS, considérées ici en sélection simple uniquement, se réalisent en remplaçant la fonctionnalisation de l'envoi de message par une fonctionnalisation du sélecteur lui-même.

Le *lookup* est une mise en œuvre directe d'un héritage dynamique qui favorise les modifications dynamiques à l'exécution. Si cette technique purement interprétée est simple, pédagogique et particulièrement compacte, ses performances trouvent vite leurs limites, surtout dans un langage comme SMALLTALK où l'envoi de message est universel. Trois catégories de solutions ont été proposées à ce problème d'efficacité. D'abord, des techniques dynamiques de cache (*lookhere* et *lookthere*), qui consistent à mémoriser les envois de message les plus récents, fréquents ou probables. Ensuite, des techniques statiques de tables (*lookin*), qui consistent à associer

Journées du GDR Programmation.
12, 13, 14 novembre 1997.
Rennes.

un indice à chaque sélecteur et classe et à constituer une table de tous les couples classe-sélecteur. Enfin, des techniques dynamiques d'arbre de décision (*lookdown*), qui consistent à descendre de la classe la plus générale en testant si la classe du receveur est une sous-classe de la classe qui étiquette le nœud courant de l'arbre.

Un compromis espace-temps apparaît vite nécessaire : les tables (*lookin*) sont temporellement très efficaces pour un coût spatial énorme, même une fois « compactées », alors que les techniques dynamiques (*lookup* et *lookdown*) sont beaucoup plus économes en espace, pour des performances temporelles médiocres. Quant aux caches, leur coût spatial non négligeable semble avoir été négligé, alors que leur efficacité temporelle n'est que statistique. Enfin, la problématique de la compilation de l'envoi de message varie sensiblement, suivant que l'on considère une compilation fermée et définitive, ou une compilation ouverte à des modifications dynamiques ultérieures. C'est un point crucial dans ces langages que l'on peut qualifier de *dynamiques* car le dynamisme du typage y sert, avant tout, à ce dynamisme à l'exécution.

Plan de l'article Cet article est consacré à l'analyse spatiale de ces techniques de compilation dans les langages dynamiques. Après cette introduction, la deuxième section présente brièvement les techniques en lice : caches, arbres de décision et tables. La section 3 décrit ensuite les paramètres des jeux de tests utilisés et les divers modes de compilation à considérer. La section 4 décrit les deux grandes catégories de techniques de compactage pour les tables et en détaille les limites. La section suivante propose enfin diverses façons de combiner les avantages des différentes techniques.

Vocabulaire Un sélecteur est *connu* dans une classe si une méthode a été définie pour ce sélecteur dans cette classe ou dans l'une de ses super-classes. Un sélecteur est *défini* dans une classe si une méthode lui est associée et que le sélecteur n'est pas connu dans les super-classes. Le sélecteur est *redéfini* dans la classe si une méthode lui est associée alors qu'il est connu d'au moins une super-classe. Nous utilisons le terme de *surcharge* dans une acception très précise, pour désigner le fait qu'un même sélecteur est défini dans des classes incomparables par l'héritage : dans la figure 1, c'est le cas du sélecteur `b` qui est défini dans les classes `B` et `E`. On ne confondra pas la surcharge avec la redéfinition (ou masquage) comme pour les sélecteurs `a` et `c` de la classe `D`. Enfin, on parlera d'*occurrence* d'envoi de message pour désigner l'expression physique qui, dans le code, réalise un envoi de message.

2 Les différentes techniques

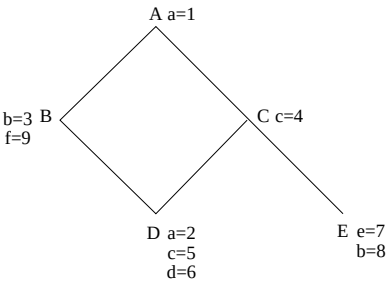
Outre le *lookup*, on peut distinguer les techniques dynamiques de cache (ou *memoization*) et d'arbres de décision, et les techniques statiques à base de tables.

2.1 Les techniques de cache

Elles consistent à mémoriser les accès les plus fréquents ou les plus récents. Elles sont couramment utilisées par différents langages, dont SMALLTALK ou OBJECTIVE-C. Il en existe deux grandes familles : le cache peut être « en ligne » ou global.

Cache en ligne Une première façon de faire est la *prédiction de type* du langage SELF [HCU91] : elle consiste à prédire statiquement, à la compilation, le type le plus probable de l'argument pour chaque occurrence d'envoi de message. C'est un *lookhere*, qui revient à remplacer l'envoi de message par un appel fonctionnel, sauf si l'objet n'est pas du type attendu. C'est une technique très utile pour compiler les envois de message à des types primitifs [JM97].

La technique du *cache en ligne* n'est qu'une complication de la prédiction de type, dans laquelle le code se modifie lui-même pour remplacer les valeurs attendues



sélecteurs

classes	*c*	*b*	*a*	*d*	*e*	*f*
A			1			
B		3	1			9
C	4		1			
D	5	3	2	6		9
E	4	8	1		7	

FIG. 1 – *Une hiérarchie d'héritage (classes en majuscules, sélecteurs en minuscules), et la grande table correspondante (les chiffres représentent les méthodes). Sur le graphe, la présence d'un sélecteur traduit la définition d'une méthode.*

par les valeurs obtenues par le *lookup* lorsque le receveur déçoit l'attente [DS84]. Des applications successives sur des objets du même type donnent lieu à un seul *lookup*. D'après certaines études [DHV95], dans chaque site d'envoi de message d'un programme SMALLTALK, le type du receveur reste constant 95 % du temps. Mais, dans le pire des cas, cette technique n'améliore strictement rien.

La technique du cache en ligne a donné lieu à diverses variantes : le cache peut être polymorphe [HCU91] (plusieurs couples classe-méthode sont alors mémorisés) ou même statistique : une première passe sur un jeu de test permet de déterminer, pour chaque occurrence, les classes de receveur les plus fréquentes. C'est la technique de CEYX [Hul85] et de SELF [AH95]. Ces techniques se prêtent mal aux modifications dynamiques : après une modification (redéfinition d'une méthode), un cache en ligne peut contenir une méthode périmée pour la classe considérée.

Cache global Le cache global contient les couples classe-sélecteur les plus fréquemment ou récemment utilisés, et il est implémenté comme une table de hachage, voire comme une simple table indexée par les sélecteurs et contenant le dernier couple classe-sélecteur [GR83]. Cette technique se prête mieux à la modification dynamique : il suffit d'invalider l'entrée de la table correspondant au sélecteur. Le cache peut être incorporé dans la fonction générique [KR90].

2.2 Les arbres de décision

Au lieu de remonter (*lookup*) de la classe du receveur vers la classe possédant la méthode à activer, le *lookdown* repose sur un arbre de décision dont les nœuds sont étiquetés par des classes et dont les feuilles possèdent des méthodes. La recherche de la méthode consiste à descendre depuis la racine de l'arbre, en testant si la classe du receveur est une sous-classe de la classe étiquette du nœud courant : si oui, la procédure se répète sur le sous-arbre de gauche, si non, sur le sous-arbre de droite. Lorsque le sous-arbre considéré est une méthode, on l'applique. Une façon naïve de construire un arbre de sélection consiste à prendre la représentation fils-frère d'un arbre recouvrant du sous-graphe d'héritage constitué des classes définissant une méthode pour le sélecteur considéré. Une méthode plus sophistiquée cherchera à équilibrer l'arbre. L'efficacité du *lookdown* repose sur l'existence d'un test de sous-classe, en temps constant si possible (§ 5.3), et sur la faible taille (profondeur) de l'arbre de sélection, donc sur le petit nombre de méthodes du sélecteur.

2.3 Les techniques de tables

La dernière catégorie de techniques est statique et consiste à associer un indice à chaque classe `c` et à chaque sélecteur `s`, puis à construire une table dont les

	NEXTSTEP	SMALLTALK			VISUALAGE
c	310	**698**	774	1 370	3 241
s	2 707	**4 518**	5 086	7 685	17 342
m	4 324	**7 623**	8 540	14 057	37 058
M	71 334	**164 839**	178 230	337 721	1 045 333
$s_{\texttt{Object}}$	—	**110**	—	122	—
x	434	**411**	399	419	750
e	—	**31 280**	35 042	57 680	152 059

FIG. 2 – *Les paramètres des jeux de tests : en gras la version utilisée dans le cours de l'article (voir [Duc97] pour le détail des sources des différents paramètres).*

classes sont les lignes et les sélecteurs les colonnes, et dont la case $(i_{\mathtt{c}}, i_{\mathtt{s}})$ contient la méthode cherchée, ou `nil` si la classe ne connaît pas le sélecteur [Cox87].

Du point de vue temporel et en théorie, cette technique est optimale : la sélection se fait en un temps constant très faible. Son inconvénient réside dans le gigantisme de la table : plusieurs millions d'entrées pour SMALLTALK. Dans la suite, on l'appellera la méthode « de la grande table ». Il apparaît heureusement que cette table est majoritairement vide, l'occupation étant à peine de l'ordre de 5 %. De nombreux travaux ont été consacrés à son compactage (voir section 4).

3 La mesure

L'objectif de ce travail est de mesurer l'impact spatial des différentes techniques. Cette mesure repose sur une simulation qui nécessite une définition précise des objets de la mesure : l'objet à mesurer aussi bien que la règle elle-même.

3.1 Les jeux de test et leurs paramètres

La comparaison des résultats de la mise en œuvre des différentes techniques repose sur de nombreux paramètres du système considéré, dont les principaux sont :

- c, s et m, les nombres respectifs de classes, de sélecteurs et de méthodes ;
- x, le nombre maximum de sélecteurs connus d'une classe ;
- M, le nombre de couples classe-sélecteur valides, c'est-à-dire le nombre d'entrées non vides de la grande table ;
- e, le nombre d'occurrences d'envois de message, assez représentatif de la taille totale du code.

Nous prendrons enfin comme exemple diverses versions de SMALLTALK, NEXTSTEP et VISUALAGE 2.0 dont les paramètres, empruntés à la littérature ou calculés sur des bibliothèques particulières, sont donnés dans la figure 2. Sauf mention du contraire, on utilisera dans la suite les chiffres de la version en gras de SMALLTALK.

3.2 Que mesurer ? Pour quelle compilation ?

Il est difficile d'isoler l'envoi de message du reste du langage pour mesurer son impact propre. En effet, dans les langages dynamiques, des structures de données sont allouées en permanence pour représenter les entités du langage : classes, sélecteurs, méthodes, etc. Les structures de données spécifiques à chaque technique de compilation sont partiellement redondantes avec les structures qui préexistent. Cette redondance est inévitable puisqu'aucune de ces structures de données ne peut *a priori* remplacer l'autre mais il n'est pas possible d'en tenir compte.

Globalement, le schéma de compilation dans lequel s'insère l'envoi de message peut se décrire à un niveau tactique et à un niveau stratégique.

	a	*b*	*e f*	*c*	*d*
A	1				
B	1	3	9		
D	2	3	9	5	6
C	1			4	
E	1	8	7	4	

FIG. 3 – *La coloration de la hiérarchie de la figure 1, dont le nombre chromatique vaut 5.*

Tactique Chaque occurrence d'envoi de message peut se compiler en ligne ou par un appel fonctionnel. La compilation en ligne est très efficace d'un point de vue temporel, mais son surcoût spatial est proportionnel à e. Ainsi, le cache en ligne impose une compilation en ligne, avec un coefficient de e qui ne peut pas être inférieur à 4, puisqu'il faut stocker dans le code l'accès à la classe du receveur, la classe du receveur précédent, la méthode précédente et un test : les envois de message finissent par occuper au moins autant de place que l'optimal des tables compactées ($M/e \approx 5$ en SMALLTALK). De leur côté, les fonctions génériques occupent un espace proportionnel à s mais leur contenu peut lui-même être compilé en ligne: vu le rapport e/s ($\approx 6,9$ en SMALLTALK), le surcoût en est plus acceptable.

Stratégie La compilation peut être ouverte et incrémentale ou fermée et globale. Nous privilégions, dans cet article, le premier point de vue d'une compilation où l'ensemble de classes compilées reste modifiable à l'exécution, par ajout de classes, de sélecteurs ou de méthodes. Sur ce point, cet article se démarque de la grande majorité de la littérature sur le sujet, où la compilation incrémentale est citée mais pas directement traitée. C'est une nécessité évidente de la famille de langages considérée, mais il n'en est pas moins vrai que les programmes développés dans ces langages peuvent nécessiter *aussi* une compilation définitive et fermée, interdisant toute modification ultérieure, pour produire un exécutable opérationnel. Nous n'insisterons pas sur cette autre approche, qui ne remet pas fondamentalement en cause les techniques envisageables, mais qui change les critères de choix.

La compilation en ligne est peu compatible avec des modifications dynamiques: nous privilégierons donc la fonctionnalisation de l'envoi de message lui-même, que ce soit par une primitive `send` ou par une fonction générique.

4 Les tables à temps d'accès constant et uniforme

Soit T la taille de la table produite, en nombre d'entrées: elle est bornée par l'inégalité $M \leq T \leq c \times s$. En effet, M représente le nombre de couples sélecteur-classe valides. Pour SMALLTALK, cela donne $165 \leq T \leq 3153$, en k-entrées: le taux d'occupation de la grande table, $M/(c \times s)$, est inférieur à 5 %. La nécessité d'un compactage est évidente, ses limites aussi: pour VISUALAGE, M vaut 1 M-entrées.

Le compactage de la grande table a été envisagé suivant deux approches: la coloration de sélecteurs et le décalage de sélecteurs. Ces techniques sont comparables par leurs résultats et par leur principe. Dans les deux cas, il s'agit de fusionner des morceaux de la grande table, de façon à en réduire la taille mais en faisant en sorte que la table résultante ne contienne toujours qu'une seule méthode par entrée. Une entrée de la table pourra ainsi correspondre à plusieurs couples classe-sélecteur, mais un seul sera valide. Aussi, lors de chaque envoi de message, l'accès à la table retournera une méthode dont il faudra vérifier qu'elle correspond bien au couple classe-sélecteur demandé: elle pourrait en effet correspondre à un autre couple de la même entrée, ce qui traduirait une erreur `message inconnu`. Pour se prémunir contre le risque d'accès à une entrée de la table avec un couple classe-sélecteur invalide, il faut que l'appel de la méthode passe un argument supplémentaire, le sélecteur effectif, que la méthode compare au sélecteur attendu. Le surcoût spatial

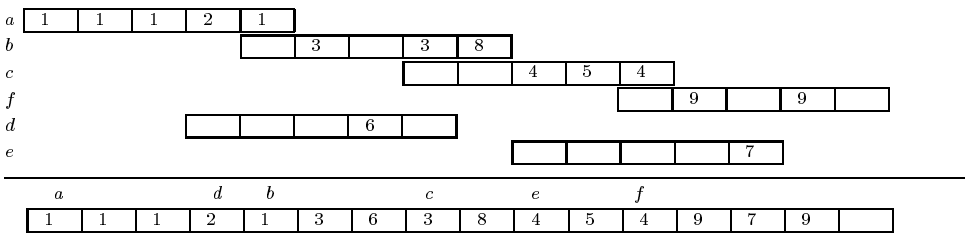


FIG. 4 – *Le décalage de sélecteurs : en haut, les colonnes de la figure 1, transposées et décalées ; en bas, le tableau résultant, obtenu par fusion verticale.*

est donc de $e + 2m$, soit 52 k-entrées, pour la compilation en ligne, $s + 2m$ soit 22 K pour les fonctions génériques, et $2m$ soit 17 K pour la primitive d'envoi de message.

Le compactage fait abandonner l'intégrité des lignes ou des colonnes et revient à privilégier, dans l'implémentation, une vision des classes comme des structures, à la Cardelli [AC96], ou une vision des méthodes comme des fonctions surchargées, à la Castagna [Cas96], ces deux visions théoriques reproduisant l'opposition entre l'approche originelle de SIMULA et l'approche fonctionnelle de CLOS.

4.1 La coloration de sélecteurs

La coloration de sélecteurs a été proposée par [DMSV89] avant d'être reprise par [AR92]. Il s'agit d'affecter à chaque sélecteur un indice, en abandonnant l'injectivité de la numérotation. Plusieurs sélecteurs pourront avoir le même indice. Cela revient à fusionner des colonnes lorsqu'une seule au plus a une case remplie dans chaque ligne. Le nom de *coloration* provient du fait qu'il s'agit, dans les premières versions de cette technique, d'un problème de coloration de graphe qui consiste en une minimisation du nombre de couleurs utilisées [Ber70].

Soit R la relation définie sur l'ensemble des sélecteurs par : $s_1 R s_2$ ssi il existe une classe dans laquelle les 2 sélecteurs sont connus. La coloration de sélecteurs correspond à la coloration du graphe de cette relation R entre les sélecteurs, c'est-à-dire une numérotation des sélecteurs telle que deux sélecteurs adjacents par R n'aient pas la même couleur. Sur la figure 1, la coloration de sélecteurs conduirait, par exemple, à choisir le même indice pour `e` et `f`. La numérotation des classes reste injective, mais elle est indifférente. Minimiser la taille de la table résultante revient à minimiser le nombre de couleurs.

Premier essai de coloration La coloration de sélecteurs a été proposée par [DMSV89], puis testée sur SMALLTALK par [AR92] avec des résultats assez décourageants. La coloration de graphe est un problème NP-difficile et le graphe des conflits de sélecteurs est énorme : 4518 sélecteurs. Les auteurs ont donc recours à une heuristique gloutonne classique. Malgré les optimisations, la coloration reste inapplicable au graphe des conflits et les auteurs abandonnent alors le traitement global pour une heuristique incrémentale d'une efficacité moyenne : le temps de compilation reste énorme. Sur ces données, le nombre chromatique est 411 : il est en fait égal à x, le nombre maximum de sélecteurs dans une classe. La matrice obtenue vérifie donc $T \approx 287$ k-entrées.

Une grande matrice ou plusieurs petits vecteurs La technique que nous présentons maintenant est utilisée en YAFOOL avec l'originalité de ne pas y être appliquée aux méthodes mais aux attributs : la différence, sans importance sur le plan technique, ne sera pas prise en compte.

Outre l'inefficacité de son calcul, la matrice produite par la coloration montre une amélioration limitée du remplissage : sur un total de $T \approx 287$ K, seuls $M \approx 165$ sont occupés. C'est insuffisant et il faut se rappeler que la coloration de sélecteurs

n'est qu'un équivalent du problème d'optimisation initial : minimiser la taille de la matrice. Une amélioration naturelle consiste à remplacer cette matrice par autant de vecteurs qu'il y a de classes. Ces vecteurs pourront se limiter, à droite, à la partie occupée des lignes, l'algorithme cherchant à les remplir en priorité par la gauche. Le problème d'optimisation a changé : il s'agit maintenant de minimiser la somme des tailles des vecteurs.

Le schéma d'algorithme pour calculer l'indice de chaque sélecteur est très simple :

```
pour chaque classe C, prise dans un certain ordre :
     pour chaque sélecteur S défini par C, non encore placé,
                              pris dans un certain ordre :
          placer S à la première place libre dans les vecteurs de
                          toutes les classes qui ont le sélecteur S ;
```

L'exemple de la figure 1 conduit aux tables de la figure 3. Lorsqu'il n'y a ni *surcharge* ni héritage multiple, et que les classes sont prises des plus générales aux plus spécifiques, cette heuristique est optimale : $T = M$, toutes les entrées des vecteurs sont occupées. La place manque pour préciser l'algorithme et le lecteur est renvoyé à [Duc97] pour plus de détails. Disons simplement que le schéma d'algorithme est paramétré par les ordres sur les classes et sur les sélecteurs. Le premier est une extension linéaire de la relation d'héritage, calculée en réutilisant les techniques de linéarisation mises au point pour la gestion de l'héritage multiple [DH89, DHH+95]. L'ordre sur les sélecteurs est plus heuristique.

L'heuristique a été appliquée à SMALLTALK et le résultat est satisfaisant : la taille de la table obtenue est $T \approx 174$ k-entrées, à comparer avec l'optimal $M \approx 164$ K, soit 6 % de « trous ». Le temps de compilation reste raisonnable (moins de 20s).

Le cas du typage statique Cette heuristique de coloration est en fait très proche de la technique des tables de fonctions virtuelles utilisée dans les langages à typage statique comme C++ [ES90]. La différence fondamentale tient au traitement de la surcharge : lorsque le typage est statique, un même sélecteur peut désigner des entités différentes dans des classes non comparables. Ce n'est plus le cas lorsque le typage est dynamique. Il s'ensuit que la numérotation des sélecteurs, qui doit être globale avec un typage dynamique, peut être relative à une classe — et à ses sous-classes — avec un typage statique. Lorsqu'il n'y a ni surcharge ni héritage multiple, l'heuristique de coloration et la technique des fonctions virtuelles donnent exactement le même résultat. L'héritage multiple dégrade toutes ces techniques.

4.2 Le décalage des sélecteurs

La seconde approche de compactage de la grande table est la technique dite des tableaux clairsemés (*sparse arrays*) [Dri93], renommée par la suite « tables par décalage de rangées » (*row displacement tables*) [DH95].

Contrairement à la coloration de sélecteurs, la technique du décalage, qui est très générale, ne remet pas en cause l'injectivité des indices des lignes et des colonnes. Elle consiste à décaler les unes par rapport aux autres les lignes (resp. les colonnes) de la matrice, jusqu'à obtenir une situation dans laquelle il n'y a pas 2 cases occupées en vis-à-vis sur la même colonne (resp. ligne), ce qui permet de les fusionner en un unique vecteur. Les indices décalés ne constituent plus une numérotation — ils ne sont plus consécutifs — mais une adresse relative au début du vecteur résultant. La question du sens du compactage se pose : décale-t-on les lignes des classes ou les colonnes des sélecteurs ? L'expérience tranche en faveur du décalage de sélecteurs, le décalage de classes produisant une table sensiblement plus grande que celle produite par la coloration ou que le décalage de sélecteurs.

Comme pour la coloration, l'efficacité du décalage dépend de l'ordre dans lequel sont pris les classes et les sélecteurs. L'avantage du décalage des sélecteurs vis-à-vis

type de compilation		compilation en ligne		`send`	fonctions génériques			
technique de sélection		cache en ligne	tables compactées			entrées mult.	mixte	*look-down*
cache en ligne	$4e$	140?						
compilation en ligne	$7e$	245?	245?					
fonctions génériques	$7s$				32?	32?	32?	32?
tables	T		165 [a]	174 [b]	174 [b]	25 [c]	82	
sélecteur à l'envoi de message	e s		(31) [d]	31	4,5	4,5	4,5	
test du sélecteur	$2m$		15	15	15	15	15	
sauf `Object` ou autres [d]	s_1 m_1		-1,5 [e]	-1,5 [e]	-3,6 [f]	-1,5 [e]	-8 [f g]	
fermeture transitive	$\frac{c^2}{2}$					8	8	8
entrées multiples						24 [c] 68 [h]		
lookdown	$4m_2$ $4m$						13,5 [g i]	30
supplément		140?	180	189	193,5	72 [c] 116 [h]	123	38
total		385?	425?	224	229,5	104 [c] 148 [h]	156	70

[a] Par décalage de sélecteurs.
[b] Par l'heuristique de coloration de sélecteurs.
[c] Avec un seuil de similarité de 10 %, d'après [VH94, DHV95].
[d] Non pris en compte dans les totaux.
[e] Pour les sélecteurs de `Object`.
[f] Pour les sélecteurs seuls de leur indice.
[g] Pour les $m_2 = 3313$ sélecteurs n'ayant qu'une méthode.
[h] D'après notre propre expérimentation.
[i] Pour les 55 sélecteurs de `Object` avec 2 méthodes.

FIG. 5 – *Tableau récapitulatif des différentes techniques appliquées à* SMALLTALK, *en k-entrées : l'entrée de chaque ligne indique le ou les paramètres en cause.*

du décalage des classes vient ainsi du fait qu'il est possible de regrouper les classes de façon à ce que les entrées occupées pour un sélecteur soient relativement groupées (voir figure 4). Comme pour l'heuristique de coloration, le résultat est optimal en l'absence de surcharge ou d'héritage multiple. Les résultats présentés dans [DH95] sont excellents, que ce soit pour le compactage ou pour le temps de compilation.

4.3 Les limites de ces approches

Les limites des tables sont de deux ordres : leur taille est un inconvénient sévère et apparemment définitif, puisque l'optimum M est atteint, et elles se prêtent mal aux modifications dynamiques. La figure 5 présente le bilan des différentes techniques pour SMALLTALK (voir [Duc97] pour des commentaires sur ces chiffres). Rapportées à l'existant, les techniques de tables provoquent un supplément de l'ordre de 190 k-entrées. La section 5 proposera des solutions plus compactes et aussi efficaces.

La limitation principale des tables compactées réside dans leur relative inefficacité pour prendre en compte des modifications dynamiques des programmes, par ajout de méthode, de sélecteur et de classe. Ces techniques prennent un temps raisonnable pour une compilation « à froid » mais il est impossible de les envisager pour tout recompiler « à chaud », après chaque modification : la durée de la compilation reste supérieure à une durée de *garbage collector*, ne serait-ce que parce qu'elle

	1	2	3	4	5	6	7+	total
méthodes	3313	541	220	172	97	37	138	7623
définitions	3940	397	94	49	15	9	14	≈ 5400

FIG. 6 – *Analyse des 4518 sélecteurs de* SMALLTALK *: nombre de sélecteurs suivant le nombre de leurs méthodes ou de leurs définitions (surcharge).*

risque d'en provoquer plusieurs. Or, ces techniques sont mal adaptées à une compilation incrémentale durant l'exécution aussi bien qu'à une compilation séparée et au chargement dynamique de « modules ». Voir [Duc97] pour une analyse plus fouillée.

5 Les approches mixtes *lookin-lookdown*

Si chaque technique est limitée, il faut envisager leur alliance. [Que97] propose ainsi de mettre des tables dans les nœuds des arbres, alors que [VH94] propose de mettre des arbres dans les entrées des tables.

5.1 Des tables dans les arbres

La structure de données de [Que97] est un arbre de décision dont certains nœuds contiennent des tables de méthodes. Ces tables sont similaires aux tables des sélecteurs utilisées dans la technique du décalage : elles sont ici limitées aux sous-classes de la classe qui étiquette le nœud. L'originalité de la structure proposée par Queinnec réside dans le fait que, dans chaque table, les sous-classes de la classe qui étiquette le nœud sont indicées relativement à la classe étiquette. L'ensemble repose sur un double codage de la hiérarchie de classes qui sert à la fois pour le test de sous-classe et pour l'indexation des classes dans les nœuds qui comportent un tableau.

Les limites de l'approche sont doubles. D'une part, elle est restreinte à l'héritage simple à cause des difficultés du codage qui doit être compact, permettre une indexation compacte des nœuds-tableaux et autoriser enfin une compilation dynamique : les seuls ordres partiels que l'on sache coder ainsi sont les arbres. D'autre part, il ne s'agit que d'une structure de données et Queinnec ne décrit aucun algorithme de compilation, bien qu'il l'utilise en pratique dans son langage MEROON.

5.2 Les tables à entrées multiples

La proposition de [VH94] consiste à fusionner les tables des classes lorsqu'elles diffèrent de très peu. La première partie de la procédure proposée conduit à un résultat analogue à l'heuristique de coloration, mais sans traiter l'héritage multiple et en écartant les sélecteurs surchargés dans une table spécifique non compactée. C'est ce point apparemment anecdotique qui doit faire l'efficacité de la technique : du fait de l'existence de deux tables différentes, il est fréquent qu'une classe ne diffère de sa super-classe que par une seule table, ce qui permet de partager l'autre.

La deuxième partie fait l'originalité de la méthode proposée. Lorsque les tables de deux classes ne diffèrent que par un nombre d'entrées réduit, elles sont fusionnées et les entrées conflictuelles sont « surchargées », pour former des arbres de décision. La fusion s'effectue de haut en bas, en comparant la table d'une classe avec celle de sa super-classe et en les fusionnant si le pourcentage de différences ne dépasse pas un certain seuil (fixé empiriquement à 10 %).

Évaluation Pour les entrées multiples, nous avons deux évaluations, toutes les deux avec un taux de similarité de 10 %, un peu contradictoires (figure 5, colonne 5). La première est obtenue en recoupant les chiffres de [VH94, DHV95]. La seconde provient de notre propre expérimentation sur la base de l'heuristique de coloration. Les raisons de cette divergence ne sont pas claires : les travaux sur les entrées multiples

sont très elliptiques sur la place occupée par les arbres de décision et la distinction de 2 types de tables n'est certainement pas sans effet.

Les arbres de décision offrent un gain d'espace considérable pour une efficacité moyenne, que la technique de Queinnec devrait améliorer. De son côté, la technique de Vitek a un comportement erratique qui la rend difficile à maîtriser, et elle ne donne de bons résultats qu'avec un seuil de similarité relativement élevé qui diminue son efficacité temporelle. L'intérêt principal de la technique des entrées multiples réside dans son surcoût nul pour les entrées simples : il est donc envisageable de s'en servir pour gérer des modifications incrémentales.

5.3 *Lookin* ou *lookdown* suivant les sélecteurs

Le *lookdown* est simple et efficace lorsque les arbres sont (très) petits. Or la figure 6 montre que les sélecteurs avec très peu de méthodes sont très nombreux, au moins en SMALLTALK. Il est alors naturel de traiter par *lookdown* les sélecteurs ayant peu de méthodes. Plus précisément, on utilisera :

- les arbres de sélection (*lookdown*), pour les sélecteurs avec une méthode (et ceux d'*Object* avec 2 méthodes), ainsi qu'en cas d'ajout dynamique de sélecteur ;
- les tables compactées (*lookin*) par l'heuristique de coloration, préférée pour sa meilleure incrémentalité, en particulier en cas d'ajout de classe ;
- les entrées multiples, étendues pour traiter les conflits de sélecteurs produits par l'héritage multiple, pour mieux résister aux modifications ;
- le *lookup* pour les entrées de table invalidées en dernière extrémité.

En pratique, chaque envoi de message sera compilé par un accès, soit à une structure de donnée propre au sélecteur, soit à une table classe-sélecteur. Pour permettre des modifications incrémentales, l'encapsulation de ces accès dans des fonctions génériques s'impose. Les accès aux indices et aux structures des sélecteurs, voire la totalité du *lookdown*, pourront être compilés en ligne.

Le traitement des petites modifications incrémentales pourra se faire de diverses façons, suivant l'ampleur de leurs effets sur les structures de compilation. Si les tables supportent bien la modification, on les étend : ainsi rajouter une classe sans nouveaux sélecteurs consiste à calculer sa table, quitte à y résoudre les conflits d'indice sur les sélecteurs par des entrées multiples. S'il s'agit de rajouter un nouveau sélecteur dont l'incorporation dans les tables serait trop lourde, on pourra compiler le sélecteur par un *lookdown*. Rajouter des méthodes à un sélecteur compilé par un *lookdown* consiste simplement à étendre son arbre de décision. En dernière extrémité, un sélecteur compilé par coloration qui serait défini (par surcharge) dans une nouvelle classe, pourrait être recompilé par *lookdown*, quitte à ce qu'il se branche sur les tables de *lookin* pour des classes existantes.

Chaque entrée de table ou feuille d'arbre contient un pointeur sur du code qui accepte uniformément comme arguments l'objet receveur, le sélecteur et les arguments supplémentaires de la méthode, et qui provoque, suivant le cas, le déclenchement de la méthode sur les arguments en question, après vérification du sélecteur (entrée simple), le déclenchement de l'erreur `message inconnu` (entrée vide), une sélection par *lookdown* (entrée multiple), ou une sélection par *lookup* (entrée invalidée). Ainsi, les entrées multiples ou invalidées ne pénalisent pas les autres envois de message. Toutes les fonctions concernées, ainsi que les méthodes ou les fonctions d'interprétation des entrées multiples, ont les mêmes arguments. C'est cette uniformité qui autorise toutes les passerelles entre *lookin*, *lookdown* et *lookup*.

Test de sous-classe Une solution simple consiste à remplacer la table de fermeture transitive (c^2 bits) qui se prête peu aux modifications dynamiques par autant de vecteurs qu'il y a de classes, chaque vecteur de bits ayant comme longueur l'indice de

la classe, ce qui n'occupe plus que $c(c+1)/2$ bits. Ce schéma naïf vaut mieux que rien tant que le nombre de classes n'est pas trop élevé. Cela représente pour SMALLTALK 31 k-octets, soit une cinquantaine d'octets par classe, mais 650 k-octets, soit 200 octets par classe, pour VISUALAGE ! En cas de besoin, il faut envisager des techniques plus élaborées de codage de hiérarchie [AKBLN89, ABJ91, Cas93, HHN95, HNR97].

Évaluation Le fait d'exclure de la coloration les 3313 sélecteurs qui n'ont qu'une seule méthode réduit la taille des tables à 119 k. Si l'on enlève aussi les 55 sélecteurs de `Object` qui n'ont que deux méthodes, on tombe à 82 k. L'occupation totale, de l'ordre de 155 k-entrées, est alors bien inférieure à celle des autres techniques de tables (figure 5). L'avantage de cette technique mixte réside dans sa souplesse vis-à-vis des modifications incrémentales. La compacité du résultat est liée au nombre de sélecteurs haut placés ayant peu de méthodes : c'est un paramètre secondaire dont il faudrait voir s'il est aussi favorable dans d'autres systèmes que SMALLTALK.

6 Conclusion et perspectives

L'examen des différentes techniques de mise en œuvre de l'envoi de message en fait apparaître plusieurs catégories :

- la recherche dynamique dans la hiérarchie (*lookup*) ou dans un arbre de décision (*lookdown*), ce dernier nécessitant un test de sous-classe efficace ;
- des caches, en ligne (*lookhere*) ou globaux (*lookthere*) ;
- une compilation statique (*lookin*), à temps d'accès constant et uniforme, reposant sur des tables classe-sélecteur calculées et compactées suivant deux approches, coloration ou décalage de sélecteurs ;
- des techniques mixtes, mêlant *lookdown* et *lookin*, les nœuds de l'arbre de décision pouvant contenir des tables, ou les entrées des tables pouvant être des arbres de décision, ou enfin une compilation variable suivant les sélecteurs : arbres pour les sélecteurs avec peu de méthodes, tables pour les autres.

Cette dernière technique mixte, que l'on peut encore étendre à l'utilisation du cache en ligne pour des cas particuliers, semble offrir le meilleur compromis.

Parmi les perspectives de ce travail, il faut citer l'adaptation de ces techniques dans plusieurs directions : la compilation fermée et définitive en prenant en compte le typage statique de `self`, la compilation séparée de modules et leur chargement dynamique, et enfin la sélection multiple des fonctions génériques, déjà traitée par des techniques de tables [CTK94, AGS94, Duj95], mais qu'il reste à traiter dans une optique de compilation ouverte.

Ce travail n'a considéré que le versant spatial de la compilation, laissant l'intuition ou la littérature juger de l'efficacité temporelle des solutions. Il est clair qu'une analyse temporelle exhaustive s'impose, les seules analyses existantes étant partielles (en compilation en ligne pour [DHV95]) et souvent discutables. Il n'existe pas dans la communauté objet de tests consensuels, analogues aux tests de Gabriel pour la programmation fonctionnelle. Or l'efficacité temporelle peut s'analyser de façon abstraite comme le fait [DHV95], mais il faut aussi songer à faire des mesures pratiques qui n'auront de réelle valeur qu'avec des tests communs.

Références

[ABJ91] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient management of transitive relationships in large data-bases including is-a hierarchies. Technical Report DCS-TR-286, Rutgers University, 1991.

[AC96] M. Abadi and L. Cardelli. *A theory of objects*. Monographs in Computer Science. Springer Verlag, 1996.

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. In *Proc. OOPSLA'94*. ACM Press, 1994.

[AH95] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *Proc. OOPSLA'95*, 1995.

[AKBLN89] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM TOPLAS*, 11(1):115–146, 1989.

[AR92] P. André and J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *Proc. OOPSLA'92*, pages 110–126. ACM Press, 1992.

[Ber70] C. Berge. *Graphes et Hypergraphes.* Dunod, 1970.

[Cas93] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In *Proc. OOPSLA'93*, pages 271–287. ACM Press, 1993.

[Cas96] G. Castagna. Le modèle fondé sur la surcharge : une visite guidée. *Technique et Science Informatiques*, 15(6):673–708, 1996.

[Cox87] B. Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, 1987.

[CTK94] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In M. Tokoro and R. Pareschi, editors, *Proc. ECOOP'94*, LNCS 821, 1994.

[DH89] R. Ducournau and M. Habib. La multiplicité de l'héritage multiple. *Technique et Science Informatiques*, 8(1):41–62, 1989.

[DH95] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proc. OOPSLA'95*, pages 141–155. ACM Press, 1995.

[DHH+95] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3):309–345, 1995.

[DHV95] K. Driesen, U. Hölzle, and J. Vitek. Message dispatch on pipelined processors. In *Proc. ECOOP'95*, pages 253–282, 1995.

[DMSV89] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*, 1989.

[Dri93] K. Driesen. Selector table indexing and sparse arrays. In *Proc. OOPSLA'93*, 1993.

[DS84] L.P. Deutsch and A. Schiffman. Efficient implementation of the SMALLTALK-80 system. In *Proc. of Principles of Prog. Lang. (POPL'84)*, 1984.

[Duc97] R. Ducournau. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet*, 3(3):241–276, 1997.

[Duj95] E. Dujardin. *Flexibilité, robustesse et efficacité des modèles de données à objets.* Thèse d'informatique, Université Paris VI, 1995.

[ES90] M.A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US, 1990.

[GR83] A. Goldberg and D. Robson. SMALLTALK *: the language and its implementation.* Addison-Wesley, Reading, MA, 1983.

[HCU91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91*, 1991.

[HHN95] M. Habib, M. Huchard, and L. Nourine. Embedding partially ordered sets into chain-products. In *Proc. KRUSE'95*, pages 147–161, 1995.

[HNR97] M. Habib, L. Nourine, and O. Raynaud. A new lattice-based heuristic for taxonomy encoding. In *Proc. KRUSE'97*, pages 60–71, 1997.

[Hul85] J.M. Hullot. CEYX version 15. Technical Report 44, 45 et 46, I.N.R.I.A., 1985.

[JM97] M. Jacques and J. Malenfant. Compiling an object-oriented language by program transformation. In *Proc. LMO'97*, 1997.

[KR90] G. Kiczales and L. Rodriguez. Efficient method dispatch in PCL. In *Proc. ACM Conf. on Lisp and Functional Programming*, 1990.

[Que97] Ch. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters*, 1997. (to appear).

[VH94] J. Vitek and R.N. Horspool. Taming message passing: efficient method look-up for dynamically typed languages. In *Proc. ECOOP'94*, pages 432–449, 1994.