L'Objet. Volume 3 – n° 3/1997, pages 241 à 276

La compilation de l'envoi de message dans les langages dynamiques

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, à un rythme aussi rapide que l'augmentation de la puissance des processeurs. 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.*

ABSTRACT. *The compilation of message sending is the key for efficiency of object-oriented languages. In the particular framework of dynamically typed languages, various techniques based on indexing classes and selectors in large tables have been studied for some years. These techniques provide a constant-time message dispatch, at the cost of very large memory consumption, despite a quasi optimal compacting. Moreover, these techniques are not adapted to the incremental run-time modifications usual in these dynamic languages. This article presents the problem of message dispatch in dynamically typed languages, the various techniques already known, and proposes a synthesis more suited to run-time modifications.*

MOTS-CLÉS : *programmation par objets, envoi de message, sélection de méthodes.*
KEYWORDS: *object-oriented programming, method dispatch, message sending.*

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 [GOL 83], 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 s'illustrerait en LISP — très schématiquement et en se restreignant à l'héritage simple [1] — par les deux fonctions `send` et `lookup`.

```
(defun lookup (s c)
       (or (get-method s c)
           (when (super c)
                 (lookup s (super c)))))

(defun send (s o . args)
       (let ((m (lookup s (class-of o))))
            (if m
                (apply-method m o args)
                (unknown-method s o args))))
```

L'échec de la recherche se traduit par une erreur `message inconnu` (fonction `unknown-method`) qui peut être traitée, comme en SMALLTALK, par un envoi de message particulier.

Les fonctions génériques de CLOS [STE 90], considérées ici en sélection simple uniquement, se réalisent en remplaçant ou en encapsulant la fonctionnalisation de l'envoi de message par une fonctionnalisation du sélecteur lui-même. Un sélecteur `foo` s'implémente alors par la fonction de même nom :

```
(defun foo (o . args)
       (apply #'send 'foo o args))
```

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. Des langages moins uniformément objet comme YAFOOL [DUC 91] peuvent pourtant s'en contenter. Deux grandes catégories de solutions ont été proposées à ce problème d'efficacité : des techniques dynamiques de cache et des techniques statiques de tables.

1.1. *Techniques dynamiques de cache*

Les techniques dynamiques 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.

1. Le fait que l'héritage soit multiple n'intervient que dans le *lookup* : dans les autres techniques, on supposera que l'on accède à la méthode héritée, sans considération de la façon dont elle est héritée.

1.1.1. *Cache en ligne*

Une première façon de faire est la *prédiction de type* du langage SELF [CHA 92, HöL 91] : 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. Chaque occurrence d'envoi de message a l'allure suivante :

```
(if (eq *expected-class* (class-of o))
    (apply-method *cached-method* o args)
    (apply #'send s o args))
```

où `*expected-class*` et `*cached-method*` sont des valeurs propres à l'occurrence considérée. C'est une technique très utile pour compiler les envois de message à des types primitifs, dont il est peu probable que les opérateurs seront surchargés et, encore moins, redéfinis.

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 (`*expected-class*` et `*cached-method*`) par les valeurs obtenues par le *lookup* lorsque le receveur déçoit l'attente [DEU 84]. Dans notre illustration en LISP, cela reviendrait à faire de la fonction `send` une macro mais nous l'épargnerons au lecteur.

Des applications successives sur des objets du même type donnent lieu à un seul *lookup*. D'après certaines études [DRI 95b], 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. Comme nous le verrons, sa simplicité indéniable cache aussi un coût non négligeable.

La technique du cache en ligne a donné lieu à diverses variantes :

- le cache en ligne peut être polymorphe [HöL 91] : plusieurs couples classe-méthode sont alors mémorisés en ligne et la compilation se traduit par un `case` sur la classe du receveur ;
- [HUL 85] propose, pour le langage CEYX, une variante statistique de ce cache polymorphe : une première passe sur un jeu de test permet de déterminer, pour chaque appel, les classes de receveur les plus fréquentes. Une technique analogue est utilisée en SELF sous le nom de *type feedback* [AGE 95].

Ces techniques de cache en ligne 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. Or, comme cette méthode n'est pas forcément périmée pour les autres classes, il n'est pas possible de modifier physiquement son code pour la rediriger sur la nouvelle méthode.

1.1.2. *Cache global*

Au lieu d'être en ligne, le cache peut être global. Il contient alors 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 [GOL 83]. On remplacera ainsi, dans l'envoi de message, la fonction `lookup` par la fonction `lookthere` :

```
(defun lookthere (s c)
       (let ((cache (vref *cache* (sindex s))))
            (if (eq c (car cache))
                (cdr cache)
                (let ((m (lookup s c)))
                     (when m
                           (rplac cache c m)
                           m)))))
```

Cette technique se prête mieux à la modification dynamique : il suffit d'invalider l'entrée de la table correspondant au sélecteur. Une variante consiste à incorporer le cache dans la fonction générique [KIC 90].

1.2. *Technique statique de table à accès direct*

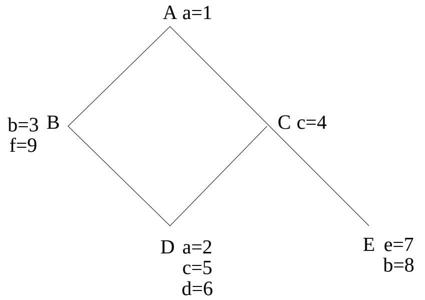
L'autre grande catégorie de techniques consiste à associer un indice à chaque classe `c` et à chaque sélecteur `s`, puis à construire une table dont les classes sont les lignes et les sélecteurs les colonnes, et dont la case $(i_{\mathtt{c}}, i_{\mathtt{s}})$ contient la méthode cherchée, ou `nil` si la classe ne connaît pas le sélecteur [COX 87]. Un *lookin* remplace alors le *lookup* pour accéder directement à la méthode :

```
(defun lookin (s c)
       (aref *table* (cindex c) (sindex s)))
```

Cette technique est, en théorie, absolument optimale : la sélection se fait en un temps constant très faible. Son inconvénient réside dans la taille gigantesque de la table : plusieurs millions d'entrées pour l'environnement SMALLTALK standard, soit plus d'une dizaine de M-octets. Dans la suite, on l'appellera la méthode « de la grande table ». Il apparaît heureusement que cette table gigantesque est majoritairement vide, l'occupation étant à peine de l'ordre de 5 %. Il y a donc des espoirs de compactage et de nombreux travaux lui ont été consacrés [DIX 89, AND 92, DRI 93, VIT 94, DRI 95a, DRI 95b].

1.3. *Plan de l'article*

Cet article est consacré à ces techniques de compilation par tables, principalement du point de vue de leur coût spatial, dans l'optique d'une compilation



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	

Figure 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.*

permettant d'effectuer dynamiquement des modifications durant l'exécution. C'est en effet un problème 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. Nous montrerons les limites de ces techniques — la mise à jour de ces tables est difficile — et la nécessité de les dépasser en posant un peu différemment le problème, comme le font [VIT 94] ou [QUE 97].

Après cette introduction, la deuxième section donne le principe de la simulation que nous avons effectuée, en précisant les paramètres qui entrent en jeu dans l'évaluation des différentes techniques, les jeux de tests utilisés et les divers modes de compilation à considérer. La section 3 décrit les deux grandes catégories de techniques à base de tables ; la section suivante en détaille les limites et décrit une approche concurrente, par arbres de sélection, qui rompt avec le postulat de l'accès en temps constant. La section 5 propose alors une synthèse de ces différentes techniques. Avant la conclusion, la section 6 pose rapidement le problème de la sélection multiple des fonctions génériques.

Vocabulaire On dira qu'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` dans l'exemple cité. Soulignons qu'il n'y a pas de surcharge interne à une classe : à chaque sélecteur connu dans une classe ne correspond qu'une seule méthode. Enfin, on parlera d'*occurrence* d'envoi de message pour désigner l'expression physique qui, dans le code, réalise un envoi de message particulier.

	NEXTSTEP [b]	SMALLTALK [a]			VISUALAGE [b]
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	—
μ	230	**236**	230	272	322
x	434 [d]	**411**	399 [d]	419	750 [d]
e	—	**31 280** [e]	35 042 [f]	57 680 [e]	152 059 [e]
C	3 174 [c]	—	—	—	—

[a] La version centrale d'après [DRI 95a], les deux autres (v4.0, en gras, et v4.1) utilisées pour nos propres tests. Ces trois versions diffèrent encore marginalement de celle utilisée dans [AND 92] et [DUC 92].
[b] D'après [DRI 95a].
[c] En K-octets, d'après [VIT 94].
[d] Valeur extrapolée à partir de l'évaluation de la coloration dans [DRI 95a].
[e] Valeur extrapolée à partir de celle de [DRI 95b] en supposant que e/m est une constante.
[f] D'après [DRI 95b].

Figure 2. *Les paramètres des jeux de tests.*

2. La mesure

L'objectif de ce travail est de mesurer l'impact spatial des différentes techniques. Cette mesure repose sur une simulation dans la mesure où le compilateur d'un langage particulier n'a pas été modifié pour incorporer successivement chacune des techniques considérées. Si cela avait été le cas, il aurait suffit de comparer la taille des différents exécutables obtenus. Une simulation a l'avantage de rester abstraite et relativement indépendante d'un langage particulier. Elle a l'inconvénient de nécessiter une définition précise des objets de la mesure : l'objet à mesurer aussi bien que la règle elle-même.

2.1. *Les jeux de test et leurs paramètres*

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

- c, le nombre de classes du système considéré ;
- s, le nombre de sélecteurs : en CLOS ce serait le nombre de fonctions génériques (`defgeneric`) ;
- m, le nombre de méthodes : le nombre de `defmethod` en CLOS ;
- $s_{\texttt{A}}$, le nombre des sélecteurs connus par une classe `A` donnée, et $m_{\texttt{A}}$, le nombre de méthodes définies dans cette classe : on a bien sûr $m = \sum_{\texttt{A}} m_{\texttt{A}}$, ainsi que $m_{\texttt{A}} \leq s_{\texttt{A}}$;

– x, le nombre maximum de sélecteurs connus d'une classe : $x = \max_{\mathtt{A}} s_{\mathtt{A}}$;

– M, le nombre de couples classe-sélecteur valides, c'est-à-dire le nombre d'entrées non vides de la grande table : $M = \sum_{\mathtt{A}} s_{\mathtt{A}}$;

– μ, le nombre moyen de sélecteurs dans une classe : $\mu = M/c$.

À ces paramètres qui ne dépendent que de la hiérarchie de classes et des méthodes qui leurs sont associées, il faut ajouter :

– la taille totale du code compilé du système considéré, C, qui est l'aune, malheureusement inconnue, à laquelle mesurer la part spécifique de l'envoi de message ;

– un paramètre assez représentatif du précédent, le nombre d'occurrences d'envois de message, noté e : comme nous ne connaissons ce paramètre e que pour un seul système, nous l'extrapolerons en considérant que le nombre moyen d'envois de message par méthode (e/m) est constant [2].

Enfin, T est la taille de la table produite, en nombre d'entrées. Il apparaît qu'elle est bornée par l'inégalité suivante :

$$M \leq T \leq c \times s \qquad [1]$$

En effet, M représente le nombre de couples sélecteur-classe valides. On peut théoriquement envisager de descendre au-dessous de cette limite inférieure dans la mesure où deux couples sélecteur-classe valides pourraient occuper la même entrée de la table s'ils ont même méthode et donc même sélecteur. D'un autre côté, suivant la technique de compactage, l'optimal de T peut être bien supérieur à M (en M^2 par exemple).

La littérature sur la compilation de l'envoi de message est très riche en jeux de tests significatifs pris dans les grandes bibliothèques de classes de différents systèmes, en particulier SMALLTALK. Nous prendrons ainsi comme exemple diverses versions de SMALLTALK, NEXTSTEP et VISUALAGE 2.0 dont les paramètres sont donnés dans la figure 2. Sauf mention du contraire, on utilisera dans la suite les chiffres de la version de gauche de SMALLTALK (en gras).

L'inégalité [1] donne alors, sur ces chiffres :

$$165 \text{ K-entrées} \leq T \leq 3153 \text{ K-entrées} \qquad [2]$$

ce qui confirme une affirmation précédente : le taux d'entrées occupées dans la grande table, $M/(c \times s)$, est compris entre 2 et 5 % suivant les systèmes. On voit alors bien la nécessité d'un compactage : la grande table non compactée occuperait 12 M-octets sur une machine 32 bits. On voit aussi les limites de ces techniques : dans le cas de VISUALAGE, M vaut 1 M-entrées. La table la mieux compactée devrait y occuper de l'ordre de 4 M-octets. Malgré la puissance des machines actuelles, cela reste un chiffre à méditer.

2. On considère donc que le code des méthodes a une « complexité » moyenne constante.

2.2. *Que mesurer ? Pour quelle compilation ?*

Il est très difficile d'isoler l'envoi de message du reste du langage pour mesurer son impact propre. Chaque technique se compare à un existant, qui n'est pas connu, ce qui imposera de comparer les techniques « toutes choses égales par ailleurs ». Mais chaque technique impose des solutions qui peuvent préexister ou non dans l'existant (ainsi des fonctions génériques ou du codage de la hiérarchie). La compilation va ainsi concerner différentes entités — classes, sélecteurs, méthodes, occurrences d'envoi de message — ainsi que les structures de données ou fonctions qui les représentent, sans parler de structures de données globales comme la grande table. Il va donc y avoir une possible redondance entre les structures de données préexistantes et celles dont chaque technique de compilation va avoir besoin. Cette redondance n'est pas évitable puisqu'aucune de ces structures de données ne peut *a priori* remplacer l'autre.

En tout cas, il serait présomptueux de prétendre exhiber « une » meilleure solution universelle, et nous allons surtout chercher à analyser les divers aspects des différentes approches. 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.

2.2.1. *Tactique : compilation en ligne ou par appel de fonction*

Au niveau tactique, chaque occurrence d'envoi de message peut se compiler de deux façons différentes :

- par un appel fonctionnel, soit à une primitive d'envoi de message (`send`), soit à une fonction générique, ce qui revient à spécialiser l'envoi de message pour un sélecteur donné ;
- en ligne, par substitution à l'une des deux formes précédentes du contenu de la fonction considérée, dans lequel ce qui est spécifique au sélecteur (indice ou structure de donnée) pourra figurer comme une constante.

La compilation en ligne de l'envoi de message 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).

À l'opposé, la primitive d'envoi de message est la plus compacte, mais sa généralité en diminue l'efficacité. Entre les deux, les fonctions génériques occupent un espace proportionnel à s. Leur contenu peut être compilé en ligne, avec à peu près le même coefficient que la compilation en ligne de l'envoi de message : vu le rapport e/s ($\approx 6,9$ en SMALLTALK), le surcoût en est plus acceptable. Par rapport à la fonction générique, la primitive d'envoi de message nécessite un argument supplémentaire, le sélecteur, qui occupera un espace e.

2.2.2. *Stratégie : compilation ouverte ou fermée*

Au niveau stratégique, 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 d'une grande majorité de la littérature sur le sujet [VIT 94, DRI 95a, DRI 95b], 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.

Dans l'optique d'une compilation incrémentale, la compilation en ligne de l'envoi de message est peu défendable parce qu'elle rend difficiles les modifications dynamiques. De plus, son évaluation précise nécessite des connaissances fines du langage cible (LISP, C, *bytecode* ou machine virtuelle, assembleur), voire de l'architecture du processeur qui interdisent d'en parler généralement. Nous privilégierons donc la fonctionnalisation de l'envoi de message lui-même, que ce soit par une primitive `send` ou par une fonction générique. Compte-tenu de notre méconnaissance du rapport M/e, l'extrapolation aux langages qui sont actuellement compilés en ligne devra se faire avec prudence. Pour ce qui est des fonctions génériques, le fait d'y avoir recours pour la compilation n'impose pas qu'elles soient accessibles au programmeur : les techniques qui vont être décrites ne sont en rien réservées aux langages qui disposent de cette abstraction fonctionnelle.

Nous n'insisterons pas sur l'autre approche d'une compilation fermée et définitive, qui ne remet pas fondamentalement en cause les techniques envisageables, mais qui change les critères de choix :

– les paramètres de la mesure (nombres de classes, sélecteurs, méthodes, taille du code, etc.) peuvent être estimés de façon précise et spécifique, au prix d'un typage minimum des entrées : on peut donc envisager des modes de compilation variés suivant les caractéristiques de l'application, privilégiant l'espace ou le temps ;

– comme les compilations en ligne ne posent plus de problème d'incrémentalité, il faut les réexaminer avec attention : le nombre de classes et de sélecteurs effectivement utilisés peut être assez faible et la compilation en ligne des accès aux indices des sélecteurs sera certainement intéressante ;

– le typage d'une expression peut entraîner, dans certains cas, une sélection statique de la méthode : une inférence de type devient donc particulièrement utile. C'est le seul cas où un appel explicite de fonction peut être évité puisque la méthode elle-même peut être compilée en ligne.

[DRI 95b] présente ainsi une analyse très fine de la place occupée par la compilation en ligne suivant diverses techniques de compilation de l'envoi de message.

3. Les tables à temps d'accès constant et uniforme

Le compactage de la grande table a été envisagé suivant deux approches : la coloration de sélecteurs et le décalage de sélecteurs. Ces deux techniques sont assez différentes mais leurs résultats sont comparables et elles participent d'un même schéma. 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`.

Le compactage a un effet non désiré mais inévitable : il fait abandonner l'intégrité des lignes ou des colonnes. Cela revient à privilégier, dans l'implémentation, une vision des classes comme des structures, à la Cardelli [ABA 96], ou une vision des méthodes comme des fonctions surchargées, à la Castagna [CAS 96], ces deux visions théoriques reproduisant l'opposition entre l'approche originelle de SIMULA et l'approche fonctionnelle de CLOS.

3.1. *La coloration*

Il s'agit d'affecter à chaque classe et à chaque sélecteur un indice, en abandonnant l'injectivité de l'une des deux numérotations. Plusieurs classes (resp. sélecteurs) pourront avoir le même indice. Cela revient à fusionner des lignes (resp. colonnes) lorsqu'une seule au plus a une case remplie dans chaque colonne (resp. ligne).

3.1.1. *Coloration de sélecteurs ou de classes*

La coloration de sélecteurs a été proposée par [DIX 89] avant d'être reprise par [AND 92]. 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 [BER 70].

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.

Une brève réflexion conduit à écarter la technique duale de la coloration de classes qui consisterait à colorer le graphe de la relation R' définie par : $c_1 R' c_2$

ssi les deux classes ont un sélecteur en commun. Dans tous les langages comme SMALLTALK qui possèdent une classe racine `Object` définissant le comportement par défaut de tous les objets du système, cette relation R' est trivialement totale : toutes les classes ont en commun les sélecteurs de `Object`. On trouve également, dans la littérature, une proposition de bi-coloration qui ne semble pas avoir été testée en vraie grandeur [HUA 92]. Nous ne considérerons, par la suite, que la coloration de sélecteurs.

3.1.2. *La gestion de* `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 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 [3]

Ces tests supplémentaires sont inutiles pour les sélecteurs communs à toutes les classes — soit, dans le cas de SMALLTALK, les 110 méthodes de `Object` — ou plus généralement pour les sélecteurs qui sont seuls de leur indice. Cette optimisation a un poids assez faible (20 % en SMALLTALK) vis-à-vis de m qui est lui-même peu significatif devant M. Nous n'en connaissons pas la portée pour e, en cas de compilation en ligne, mais une telle optimisation est surtout à envisager, en compilation dynamique, dans le cas des fonctions génériques.

De même, un minimum d'inférences de type — au sein d'une méthode, le receveur courant (`self`) est toujours typé statiquement — doit permettre de s'assurer que certains envois de message sont garantis contre l'erreur `message inconnu`. Une compilation en ligne pourrait en tenir compte.

3.1.3. *Premier essai de coloration*

La coloration de sélecteurs a été proposée par [DIX 89], puis testée en vraie grandeur par [AND 92] avec des résultats assez décourageants. Le jeu de tests utilisé est une variante du SMALLTALK que nous avons décrit plus haut : nous en transposons les résultats à nos chiffres. Sur ces données, le nombre chromatique est 411 : il est en fait égal [4] à x, le nombre maximum de sélecteurs dans une classe, c'est-à-dire $\max_{\mathtt{A}} s_{\mathtt{A}}$. La matrice obtenue vérifie donc $T \approx 287$ K-entrées.

La coloration de graphe est un problème difficile — NP-difficile pour être précis — et le graphe des conflits de sélecteurs est énorme : 4518 sélecteurs, avec beaucoup de grosses cliques (tous les sélecteurs d'une classe forment une

3. Comme le sélecteur doit être utilisé de toute façon dans la compilation en ligne de l'envoi de message ou des fonctions génériques, l'effectivité de ce surcoût est assez relative.

4. Le nombre chromatique est l'invariant d'un graphe représentant le plus petit nombre de couleurs permettant de le colorer. Il se trouve que presque tous nos exemples exhibent cette égalité mais on construit facilement des exemples où le nombre chromatique est strictement plus grand que x (n sélecteurs, n classes ayant chacune $n - 1$ sélecteurs).

clique). Il est donc hors de question d'en effectuer la coloration de façon optimale et les auteurs ont recours à une heuristique gloutonne classique. Malgré de nombreuses optimisations — par exemple mise à l'écart de la centaine de sélecteurs de `Object`, qui sont en conflit avec tous les autres — le résultat de son application au graphe des conflits est sans appel : temps de calcul et espace mémoire nécessaire démesurés[5]. Les auteurs abandonnent alors le traitement global pour une heuristique incrémentale qui reste d'une efficacité moyenne : le temps de compilation reste énorme.

L'erreur fondamentale était de calculer explicitement le graphe des conflits. Nous présentons dans la suite une variante plus efficace de la coloration. On peut juste regretter que ce premier résultat, négatif, ait joué un rôle de repoussoir et de faire-valoir auprès des acteurs du domaine, [DRI 95a] par exemple.

3.2. *Une heuristique de coloration*

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.

3.2.1. *Une grande matrice ou plusieurs petits vecteurs*

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 optimisation naturelle consiste à remplacer cette énorme 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 alors changé : il s'agit maintenant de minimiser la somme des tailles des vecteurs. Pour ce nouveau problème, on se contentera d'une heuristique dont il se trouve qu'elle est aussi une bonne heuristique pour la coloration : sur nos jeux de tests, le résultat est souvent optimal pour la coloration mais sans doute pas pour la minimisation des vecteurs.

La sélection s'opérera comme suit, avec un test supplémentaire pour empêcher les débordements qui traduiraient l'erreur `message inconnu` :

```
(defun lookin (s c)
       (let ((tab (ctable c))
             (i (sindex s)))
            (when (< i (vlen tab))
                  (vref tab i))))
```

5. Même en considérant le caractère désuet du SUN3-80 sur lequel les tests ont été effectués.

	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	

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

Il faut aussi rajouter dans la structure de données la taille de chaque vecteur (soit c petits entiers), voire le surcoût occasionné par la gestion de la mémoire, GC par exemple : mais ce surcoût reste marginal ($\mu = M/c \approx 230$).

3.2.2. *Schéma d'algorithme*

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 ;
     finpour
finpour
```

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 l'implémentation qui permet d'obtenir une certaine efficacité. Nous nous contenterons de discuter des deux paramètres de ce schéma d'algorithme qui participent à la définition de son résultat et qui sont manifestement très importants pour sa qualité : les ordres sur les classes et sur les sélecteurs.

Soulignons d'abord que l'approche ne nécessite qu'un ordre sur les sélecteurs. Passer par un ordre sur les classes n'est qu'une simplification traduisant le fait que les sélecteurs définis dans une classe doivent être placés, à l'évidence, avant ceux définis dans ses sous-classes. Suivant cet argument, l'ordre sur les classes doit donc être une extension linéaire de l'ordre de spécialisation mais il n'y a pas de raison que l'ordre optimal sur les sélecteurs (celui qui minimise la taille des vecteurs) se projette aussi bien sur un ordre sur les classes. Ces restrictions énoncées, il reste encore à trouver une « bonne » extension linéaire. Nous avons expérimenté cette heuristique avec deux paramètres de poids des classes, associés respectivement au nombre de sélecteurs définis dans la classe, et au sous-ensemble de sélecteurs surchargés. Un poids ayant été abstraitement défini, il reste à calculer une extension linéaire « respectant » ce poids. Une solution naïve consiste à calculer l'extension linéaire en prenant successivement des minimaux de la relation d'ordre, en choisissant une classe de poids minimal lorsqu'il y a le choix. Le calcul de l'extension linéaire est alors *ad hoc* et son résultat un peu indéterminé : on cherche à optimiser le critère à court terme, sans souci du long terme. Une autre façon de faire privilégie, au contraire, le long terme au détriment du court terme (malgré l'image, on ne sait pas ce qu'il faut privilégier), en étant tout à la fois simple et efficace : au lieu de construire l'extension linéraire « en descendant », en prenant à chaque fois la sous-classe la meilleure, on construit l'extension linéaire « en remontant », pour inverser le résultat. On retrouve alors une situation assez comparable au problème posé par le calcul des linéarisations dans la gestion de l'héritage multiple [DUC 89, DUC 95]. Plus précisément, on reprend ces techniques de linéarisation, en choisissant la linéarisation la plus simple, dénommée $\mathcal{P}_1$ dans les références précédentes et $\mathcal{L}_{LOOPS}$ dans [DUC 94]. Cette linéarisation se calculera sur une classe virtuelle $\perp$, sous-classe directe de toutes les classes terminales, en associant à chaque classe une *multiplicité* (ou *local precedence order*) induite par le poids de chaque classe. Comme l'extension linéaire obtenue sera inversée, on prendra les super-classes directes d'une classe dans l'ordre croissant de leur poids [6]. L'extension linéaire obtenue est *gloutonne* (en fait *df-gloutonne*), c'est-à-dire qu'elle est construite en montant tant que c'est possible, ce qui assure qu'en sens inverse, la compilation descendra directement jusqu'à la classe ayant le plus grand poids. Sur ces notions d'extension linéaire et leur application aux problèmes de l'héritage, voir [DUC 89].

Le choix d'un poids a donné lieu à quelque surprise : il semblait naturel qu'il soit croissant avec le nombre de sélecteurs surchargés définis dans chaque classe. L'expérimentation a montré le contraire et a conduit à placer en premier les classes définissant le moins de sélecteurs surchargés.

L'ordre sur les sélecteurs a moins d'importance puisqu'il est largement imposé par l'ordre sur les classes. Sur la base de nos principes abstraits, nous avons cherché à placer d'abord les sélecteurs les plus surchargés, ceux qui sont définis par le plus de classes, sans expérimenter d'autres solutions.

3.2.3. *Évaluation de l'heuristique*

L'heuristique a été appliquée à SMALLTALK et le résultat est doublement satisfaisant : la taille de la table obtenue est $T \approx 174$ K-entrées, à comparer avec l'optimal de $M \approx 164$ K, ce qui fait 6 % de « trous » [7]. Et le temps de compilation reste raisonnable : moins de 20s CPU sur une SparcStation 5 à

6. En cas d'héritage simple comme en SMALLTALK, cet ordre local ne concerne que $\perp$.

7. On construit des exemples artificiels où T est au mieux de l'ordre de M^2 : n sélecteurs, n classes ayant chacune 1 sélecteur, plus une $(n+1)$-ième ayant tous les sélecteurs (par surcharge ou héritage multiple).

70 Mhz [8]. Sur une hiérarchie de 600 classes YAFOOL avec beaucoup d'héritage multiple, le taux de trous est du même ordre.

Ces résultats remettent ainsi dans la course, si course il y a, la technique de coloration. Notre évaluation est moins fine que celle de [DRI 95a] pour la technique du décalage (§ 3.3), et les résultats moins bons, mais ils restent suffisamment bons pour que l'on continue à considérer cette approche. Le morcellement de la grande table ne peut d'ailleurs qu'améliorer la souplesse d'adaptation face aux modifications incrémentales (§ 4.2 et § 5).

3.2.4. *La coloration d'attributs*

La technique de coloration est appliquée en YAFOOL aux attributs (ou variables d'instance) du langage. En effet, la coloration d'attributs permet d'imposer à un attribut un rang unique dans toutes les instances d'une classe qui définit l'attribut : l'accès à l'attribut se fait alors en un temps constant et uniforme dans toutes les méthodes. Dans un langage comme YAFOOL qui ne pratique pas une encapsulation orthodoxe, il faut aussi se prémunir contre l'erreur `attribut inconnu` : chaque classe se voit associer un vecteur de ses attributs.

Cette technique provoque des trous dans les objets : son utilisation assez marginale en YAFOOL — restreinte à des types d'attributs particuliers — n'a pas posé de problème, mais il faudrait y réfléchir à deux fois avant de l'utiliser plus systématiquement.

3.2.5. *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++ [ELL 90]. La différence fondamentale tient au traitement de la surcharge : lorsque le typage est dynamique, un même sélecteur désigne partout la même entité puisqu'une même occurrence d'envoi de message peut s'adresser à des *définitions* différentes du même sélecteur [9]. En revanche, avec un typage statique il est possible de distinguer des entités différentes désignées par le même sélecteur, un renommage adéquat éliminant la surcharge. Il s'ensuit que la numérotation des sélecteurs, qui doit être globale dans le premier cas, peut être relative à une classe — et à ses sous-classes — dans le second. 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 (modulo une permutation). L'héritage multiple dégrade de la même manière toutes ces techniques, la solution pratiquée alors par C++ différant de l'heuristique présentée ici.

8. Le compilateur de YAFOOL utilisé pour notre simulation est écrit en YAFOOL, langage dont l'efficacité n'a jamais été l'objectif premier, et il repose sur un héritage particulièrement dynamique : une réécriture optimisée devrait améliorer les choses d'un ordre de grandeur.

9. Le langage ROME [CAR 90] est une exception : bien que typé dynamiquement, il traite la surcharge comme un langage à typage statique. L'ensemble de cette problématique est discutée dans [DUC 95] avec les distinctions d'héritage et de conflits *de nom* et *de valeur*.

3.3. *Le décalage*

La seconde approche de compactage de la grande table est la technique dite des tableaux clairsemés (*sparse arrays*) [DRI 93], renommée par la suite « tables par décalage de rangées » (*row displacement tables*) [DRI 95a].

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.

3.3.1. *Décalage des sélecteurs*

Comme pour la coloration, la question du sens du compactage se pose : décale-t-on les lignes des classes ou les colonnes des sélecteurs ? Mais, contrairement à la coloration, les deux variantes se discutent. L'expérience tranche en faveur du décalage de sélecteurs, le décalage de classes produisant une table de taille $T \approx 278$ K-entrées, soit sensiblement plus que la coloration ou que le décalage de sélecteurs.

Dans les deux cas, la recherche de la méthode est particulièrement simple et ne nécessite aucun test complémentaire :

```
(defun lookin (s c)
       (vref *table* (+ (cindex c) (sindex s))))
```

Le traitement de l'erreur `message inconnu` se fait de la même manière que pour la coloration de sélecteurs, en passant le sélecteur en argument et en le testant dans le corps de la méthode : l'occupation mémoire est donc la même. La seule différence, marginale, réside dans l'impossibilité d'éviter le test du sélecteur pour les sélecteurs qui seraient « seuls de leur indice », puisqu'ils le sont tous. En revanche, on peut toujours le faire pour les sélecteurs d'`Object`.

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 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. Cet ordre sur les classes s'obtient comme une extension linéaire gloutonne de la hiérarchie de classes[10]. Enfin, les sélecteurs sont ordonnés par taille, c'est-à-dire par le nombre de classes pour lesquelles le couple classe-sélecteur est valide. L'insertion se fait ensuite de façon gloutonne,

10. [DRI 95a] parle de parcours en profondeur en cas d'héritage simple et discute superficiellement le problème de l'héritage multiple. Il est possible que l'extension linéaire de l'heuristique de coloration apporte ici une amélioration en présence d'héritage multiple.

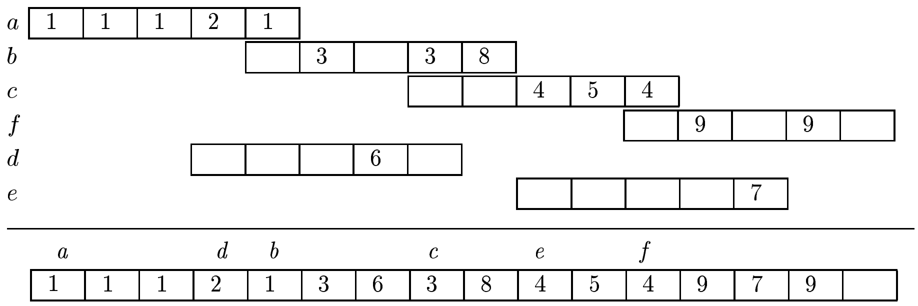


Figure 4. *Le décalage de sélecteurs : en haut, les colonnes de la figure 1, transposées et décalées ; en bas, le tableau résultant, obtenu par fusion verticale.*

en plaçant chaque rangée à la première place libre. La figure 4 donne le résultat du décalage obtenu à partir de la grande table de la figure 1.

3.3.2. *Évaluation*

Comme pour l'heuristique de coloration, cette méthode procure un résultat optimal (sans « trous ») en l'absence de surcharge ou d'héritage multiple. Les résultats présentés dans [DRI 95a] sont excellents : la table produite n'a que 0,4 % de trous, à comparer aux 6 % de notre technique de coloration. Mais la différence n'est pas significative, puisque que l'on compare en fait les 94 et 99 % complémentaires. L'héritage multiple provoque une certaine dégradation, mais l'amélioration suggérée ici peut sans doute y remédier. Le temps de compilation va de 0,4 à 4,3 secondes CPU, suivant les algorithmes, sur une SparcStation 20 à 60 Mhz, soit du même ordre de grandeur que l'heuristique de coloration.

4. Les limites de ces approches

Les limites de ces techniques de tables sont de deux ordres : leur taille est un inconvénient sévère et apparemment définitif, puisque l'optimal M est quasiment atteint, et elles se prêtent mal à l'incrémentalité qui est indispensable à des langages aussi dynamiques par leur utilisation que par leur typage.

4.1. ***Réduire encore la taille***

Suivant la technique utilisée, le bilan des tables compactées pour SMALLTALK (figure 5, colonnes 2 à 4) est compris entre 224 et 425 K-entrées, soit entre 0,9 et 1,7 M-octets. Dans cette évaluation, on a associé le décalage à la compilation en ligne (pour une compilation définitive) et la coloration aux fonctions génériques et à la primitive `send` (pour une compilation incrémentale). Dans

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		(31) [d]	31				
	s				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 [g]	
fermeture transitive	$\frac{c^2}{2}$					8	8	8
entrées multiples						24 [c] 68 [h]		
lookdown	$4m_2$ $4m$						13,5 [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 [VIT 94, DRI 95b].
[d] Non pris en compte dans les totaux.
[e] Pour les 110 sélecteurs de `Object` et leurs 687 méthodes.
[f] Pour les $s_1 = 225$ sélecteurs seuls de leur indice et leurs $m_1 = 1616$ méthodes.
[g] Pour les 120 sélecteurs seuls de leur indice et leurs 687 méthodes, ainsi que pour les $m_2 = 3313$ sélecteurs n'ayant qu'une méthode.
[h] D'après notre propre expérimentation.
[i] Pour les $m_2 = 3313$ sélecteurs avec une seule méthode et les 55 de `Object` avec 2 méthodes.

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

	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	$\approx$ 5400

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

ces chiffres, l'estimation de la place occupée par les fonctions génériques et la compilation en ligne est très grossière : un coefficient arbitraire mais plausible[11] de 7 a été appliqué uniformément aux paramètres s et e. Le degré zéro considéré pour la compilation de l'occurrence d'envoi de message est la place nécessaire pour les fonctions génériques. On remarque aussi que, sous l'hypothèse que e/m soit approximativement constant et avec le constat de la relative constance de m/s sur les systèmes considérés, le coût de l'argument supplémentaire à l'appel de `send` équilibre le coût des fonctions génériques.

Le cache en ligne (colonne 1 de la figure 5) ne serait guère moins encombrant : il suppose une compilation en ligne et augmente de quelques unités le coefficient de e, ce qui compense vite l'absence de table, mais pour une efficacité bien inférieure, au moins dans le pire des cas.

Rapportées à l'existant, les techniques de tables ne provoquent qu'un supplément de l'ordre de 190 K-entrées. Pour NEXTSTEP, dont nous connaissons le paramètre C (taille totale du code compilé, égal à 3,1 M-octets), le surcoût provoqué par les tables est de l'ordre de 320 K-octets, soit 10 % de C. Vaut-il la peine de chercher à faire beaucoup mieux ? Deux arguments incitent à répondre par l'affirmative. Le paramètre C cité par [VIT 94] est mystérieux : que représente-t-il exactement ? L'autre argument est collectif : les autres contributions de ces 3,1 M-octets ne pourraient-elles pas, elles aussi, faire l'objet d'un compactage ? Quoi qu'il en soit, nous allons chercher des solutions plus compactes et, si possible, aussi efficaces.

4.1.1. *Les arbres de décision*

Si l'on examine les chiffres donnés dans la figure 2 on constate qu'au moins $(2s - m)$ sélecteurs — soit entre 1313 et 1632 pour SMALLTALK, suivant les versions, et 1090 pour NEXTSTEP — ne correspondent qu'à une seule méthode. En fait, des statistiques précises (figure 6) montrent qu'il y en a 3313 dans notre version de SMALLTALK. Plutôt que d'intégrer ce genre de sélecteur à une grande table, où ils peuvent occuper de l'ordre de c entrées chacun[12], il paraît préférable de les représenter sous la forme d'une paire classe-méthode :

11. [DRI 95b] donne des coefficients de e compris entre 3 et 7 suivant les techniques, avec des coefficients de m compris entre 2 et 7, une partie de l'envoi de message pouvant être pris en charge par la méthode.

12. S'ils sont situés près de la racine (1285 sur 3313). En revanche, si ce sont des classes terminales (2028 sur 3313), ils occupent juste une entrée dans la table de la classe et ils boucheront idéalement les trous dans les techniques de compactage.

un simple test sur la classe du receveur permet de faire la sélection, c'est-à-dire d'appeler la méthode ou de déclencher l'erreur `message inconnu`.

Cette idée se généralise dans la notion d'arbre de décision dont les nœuds sont étiquetés par des classes et les feuilles sont des méthodes (ou l'erreur `message inconnu`). La sélection devient une procédure de *lookdown* :

```
(defun lookdown (tree c)
      (if (nodep tree)
          (lookdown (if (subclassp c (node-label tree))
                        (node-then tree)
                        (node-else tree))
                    c)
          tree))
```

Le *lookdown* est une version interprétée duale du *lookup*. Mais l'arbre de décision d'un sélecteur donné se compile aisément en une fonction propre à ce sélecteur, donnant ainsi une compilation en ligne des fonctions génériques. Lorsque le receveur est typé statiquement (`self` par exemple), la sélection peut se faire en rentrant dans l'arbre ailleurs que par sa racine. 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, qui peut occuper c^2 bits si l'on se contente d'une simple table de fermeture transitive (cf. § 5.2).

4.1.2. *Une première synthèse* lookin-lookdown

[QUE 97] propose une structure de données particulière pour tenter de concilier les approches de tables et d'arbres. C'est un arbre de décision dont certains nœuds contiennent des tables de méthodes. Ces tables sont assez similaires aux tables des sélecteurs utilisées, avant le décalage, 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. Ce codage a un surcoût de $2 \times \mathtt{depth}(\mathtt{A}) + 1$ petits entiers, pour chaque classe `A`, où $\mathtt{depth}(\mathtt{A})$ représente la profondeur de la classe, c'est-à-dire le nombre de ses super-classes (en héritage simple). Soit un total de 15 K-octets pour SMALLTALK, au lieu de 62 pour la table (et de 31 pour la solution présentée plus loin) : le gain est faible mais le codage de Queinnec a un double usage (test de sous-classe et indexation).

Les limites de l'approche sont doubles :

– 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 ;
– il ne s'agit que d'une structure de données et [QUE 97] ne décrit aucun algorithme de compilation, bien que MEROON, une extension objet de SCHEME, soit compilée suivant cette technique [QUE 93].

4.1.3. *Les tables à entrées multiples*

Pour compacter encore les tables dont la borne inférieure M est quasiment atteinte, il faut changer d'objectif et se résoudre à ne plus accéder aux méthodes en une seule indirection. Plutôt que de mettre des tables dans les nœuds des arbres, la proposition de [VIT 94] consiste à mettre des arbres dans les entrées des tables, en fusionnant des rangées (lignes ou colonnes) lorsqu'elles diffèrent de très peu. Cette fusion est appliquée aux tables produites par une variante de coloration de sélecteurs, mais elle s'appliquerait tout aussi bien au décalage de classes, ce qui pourrait justifier de réétudier cette approche à la lumière de cette nouvelle technique. En revanche, elle ne s'applique pas au décalage de sélecteurs puisque deux sélecteurs différents ne peuvent avoir aucune méthode en commun. La méthode se décompose en plusieurs étapes :

1. calcul de la « grande table », par un parcours en profondeur de la hiérarchie, en plaçant au fur et à mesure les sélecteurs des classes rencontrées ;
2. élimination des cases vides en fin de vecteurs ;
3. les cas de surcharge sont mis de côté, dans des « tables de conflits » qui ne sont pas considérées à l'étape 4 ;
4. fusion de sélecteurs (*selector aliasing*) pour les sélecteurs non surchargés : comme la coloration, cette étape revient à affecter le même indice à des sélecteurs qui ne sont pas en conflit ;
5. partage de tables lorsque leurs contenus sont identiques : du fait que deux types de tables sont distingués (point 3), il peut effectivement arriver que deux classes aient les mêmes entrées pour l'une des deux tables ;
6. lorsque deux tables ne diffèrent que par un nombre d'entrées réduit, elles sont fusionnées et les entrées conflictuelles sont « surchargées ». 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 à 10 %).

Les étapes 1–4 constituent en fait une adaptation de la technique des tables virtuelles au cas du typage dynamique, en écartant tout ce qui pourrait poser un problème. En effet, les deux cas pouvant provoquer des trous — l'héritage multiple[13] et la surcharge — sont providentiellement écartés par les auteurs : le premier parce qu'il n'est pas traité, le second grâce à une table annexe non compactée. L'étape 5 est une optimisation naturelle rendue possible par la distinction des deux sortes de tables mais son effet est marginal. Malgré cette optimisation, la mise à l'écart des cas de surcharge — justifiée par les auteurs par leur faible nombre, cf. figure 6 — est certainement néfaste (au moins si l'on compare le résultat des étapes 1–5 avec celui des autres techniques de compactage) : sur les données de NEXTSTEP, les deux tables occupent, à la fin de l'étape 4 (ou 5, ce n'est pas clair) un espace de 111 K-entrées, alors que $M \approx 71$ K-entrées et que [DRI 95a] revendique, pour le décalage de sélecteurs, un compactage de l'ordre de 99,6 %.

C'est la sixième étape qui fait l'originalité de la technique, car elle apporte une réponse qui modifie profondément la question. Le postulat de l'accès en un temps constant et uniforme est abandonné, en échange d'un taux de compactage apparemment impressionnant. Il s'agit de fusionner deux tables — en pratique les tables d'une classe et de sa (ses) super-classe(s) — lorsqu'elles diffèrent de peu[14]. La table résultante va donc contenir des entrées simples et des entrées multiples qui sont implémentées par des petites procédures de *lookdown* :

```
(defun vlookin (s c)                            ;  V pour Vitek
       (let ((tab (ctable c))
             (i (sindex s)))
            (when (< i (vlen tab))
                  (lookdown (vref tab i) c))))
```

4.1.4. *Évaluation*

La dernière colonne de la figure 5 montre le coût spatial du *lookdown* avec un codage naïf de la fermeture transitive (cf. § 5). On a utilisé pour l'occupation des arbres de décision le nombre de nœuds, c'est-à-dire m, avec un coefficient 4, qui correspond à la taille de la structure des nœuds : pour une compilation en ligne des entrées multiples, [DRI 95b] obtient le coefficient assez voisin de 7.

Pour les entrées multiples, nous avons deux évaluations un peu contradictoires. D'après [VIT 94], en fusionnant deux tables lorsqu'elles ont moins de 10 % de différence, l'occupation totale tombe à $T/M \approx 15$ %, donc $T \approx 24700$.

13. L'héritage multiple n'est pas explicitement exclu, mais les exemples des auteurs sont SMALLTALK et NEXTSTEP (basé sur le langage OBJECTIVE-C), qui sont tous en héritage simple, et il n'apparaît nulle part que l'héritage multiple puisse être envisagé.

14. L'indice de dissimilarité est défini comme le rapport entre le nombre d'entrées de la sous-classe différentes de l'entrée correspondante de la super-classe, sur la taille de la table (de la sous-classe ?). Il serait plus judicieux de rapporter au nombre d'entrées non vides de la table, la taille n'étant qu'un paramètre assez arbitraire. L'article ne précise pas plus ce que sont deux entrées différentes, lorsque l'une est simple et l'autre multiple (par exemple, lorsque l'entrée multiple de la super-classe provoquerait bien, dans le contexte de la sous-classe, la sélection de l'entrée simple de cette dernière).

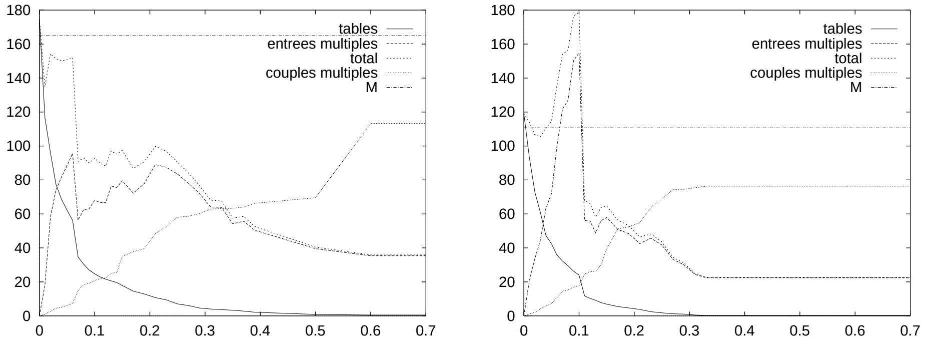


Figure 7. *La place occupée (en K-entrées) par la technique des entrées multiples, suivant le seuil de similarité : à gauche, sur la base de la coloration de sélecteurs, à droite, dans la technique mixte. Ces diagrammes mettent en évidence le caractère erratique de la place occupée par les entrées multiples (avec un coefficient de 4), et le surcoût de cette technique lorsque les sélecteurs n'ayant qu'une méthode sont enlevés.*

Bien sûr, il faut ajouter à cela l'espace occupé, d'abord par les entrées multiples[15], qui se révèle du même ordre de grandeur, ensuite par le codage de la fermeture transitive pour le test de sous-classe. Du point de vue temporel, les mesures des auteurs ne sont pas probantes, la différence entre des compactages maximal (taux de 100 %) et minimal (0 %) étant étonnamment faible.

Nous avons appliqué cette technique des entrées multiples à des tables produites en remplaçant les étapes 1-4 par l'heuristique de coloration. Avec un taux de similarité de 10 %, on obtient des tables de 25 K-entrées — comme [VIT 94] malgré la différence de technique pour les étapes 1-4 —, mais avec 2348 entrées multiples contenant au total 16998 nœuds — ce qui est très sensiblement supérieur aux chiffres des auteurs précédents[16]. Si l'on prend un coefficient de 4 pour les méthodes des entrées multiples, on obtient 93 K-entrées pour les tables et les entrées multiples, ce qui n'est plus si loin de $M = 165K$.

15. L'espace occupé par les entrées multiples n'est pas considéré ni analysé dans [VIT 94]. [DRI 95b] donne les chiffres de 7 % d'entrées surchargées, soit 1730, ayant en moyenne 3,5 nœuds, soit un total d'environ 6050 nœuds dans des entrées surchargées.

16. Les raisons de cette divergences ne sont pas claires. D'une part, les estimations des nombres et tailles des entrées multiples ne figurent qu'anecdotiquement dans [DRI 95b]. D'autre part, les étapes 1-4 ne sont pas réalisées de la même manière. De plus, la qualité du résultat apparaît très sensible au seuil de similarité, d'une façon assez erratique. Les entrées multiples peuvent ainsi avoir un effet totalement négatif comme le montre la figure 7 (droite). Il est possible que les bons résultats des auteurs soient dus au découpage arbitraire en deux tables. Un autre découpage tout aussi arbitraire, portant par exemple sur la parité du rang d'introduction ou de la longueur du nom du sélecteur, aurait alors le même effet. Enfin, la définition précise du taux de similarité et de l'algorithme étant un peu laissée à la liberté du lecteur ou du répétiteur, nous avons adopté la politique suivante : le taux de dissimilarité est défini comme le rapport entre le nombre de méthodes définies dans la classe au nombre total de sélecteurs connus par la classe et les tables sont fusionnées de haut en bas. Le taux de dissimilarité obtenu ainsi est peut-être plus faible que pour les auteurs.

La figure 5 (colonne 5) donne les résultats de la technique des entrées multiples, dans un contexte de fonctions génériques et non de compilation en ligne comme le font les auteurs, en superposant les chiffres que l'ont peut déduire de [DRI 95b] et ceux de notre propre expérimentation. L'intérêt principal de cette technique réside dans son surcoût nul pour les entrées simples : il est donc envisageable de s'en servir pour gérer des modifications incrémentales (§ 4.2.1).

L'occupation mémoire de ces nouvelles techniques est inférieure de moitié à celles des autres techniques de compactage : moins de 100 K contre plus de 200. Les arbres de décision offrent un gain d'espace considérable pour une efficacité moyenne médiocre, 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. Dans nos essais, pour le seuil de similarité optimal de 7 % (figure 7, gauche), l'entrée de près de 10 % des couples classe-sélecteur est multiple, ce qui est relativement faible, mais la longueur moyenne des entrées par couple classe-sélecteur est supérieure à 4 : cela signifie que, si tous les couples classe-sélecteur sont équiprobables, chaque envoi de message est traité en moyenne par un *lookdown* de plus de 3 nœuds.

Les performances temporelles, moins bonnes que celles des tables à entrées simples, sont difficiles à évaluer. Les mesures de [VIT 94] ne sont pas très probantes et celles de [DRI 95b] ne s'appliquent qu'à une compilation en ligne.

4.2. *Les modifications dynamiques*

La limitation principale des techniques de 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. Les techniques de compilation considérées ici prennent un temps raisonnable pour une compilation « à froid ». Il est, en revanche, impossible de les envisager pour tout recompiler « à chaud », après chaque petite modification : la durée de la compilation (de l'ordre de quelques secondes CPU pour SMALLTALK) reste supérieure à une durée de *garbage collector*, ne serait-ce que parce qu'elle risque d'en provoquer plusieurs. Or, ces techniques se révèlent mal adaptées à une compilation incrémentale durant l'exécution aussi bien qu'à une compilation séparée et au chargement dynamique de « modules ».

4.2.1. *La compilation incrémentale*

Il n'y a pas de difficultés techniques fondamentales pour adapter les techniques considérées à la compilation incrémentale. Ajouter une méthode ne demande qu'une modification des entrées de quelques tables : en cas de surcharge, il est possible qu'un conflit d'indice apparaisse (l'indice du sélecteur étant déjà occupé par un autre sélecteur dans la classe ou dans l'une de ses sous-classes). Le décalage comme la coloration sont en fait naturellement incrémentales, sur les sélecteurs et sur les classes respectivement. Ajouter une classe ne demande, pour la coloration, que le calcul de la table de la classe : en cas d'héritage multiple, il est pourtant possible qu'un conflit apparaisse pour deux sélecteurs hérités qui partageraient le même indice. Ajouter un sélecteur revient, avec le décalage, à calculer la table du sélecteur et à la décaler. En revanche, ajouter un sélecteur impose, pour la coloration, de calculer son indice et de recalculer les tables de toutes les classes qui connaissent le sélecteur : c'est immédiat si ce sélecteur est défini dans la dernière classe définie. Ajouter une classe impose, avec le décalage, de recalculer les tables de tous les sélecteurs hérités par la classe avant de les redécaler.

Toutes ces opérations semblent d'une ampleur relativement minime mais elles soulèvent divers problèmes. D'abord, l'allocation mémoire pose un problème, en particulier pour le décalage qui va nécessiter une très grande table partiellement occupée, dans laquelle les modifications créeront des trous. Les tables de la coloration sont plus petites et peuvent être allouées et libérées à la demande. Ensuite, on est confronté à un compromis entre le temps pris pour la modification et l'espace occupé par son résultat. Les algorithmes incrémentaux peuvent être très loin de l'optimal de la technique considérée : en effet, si les techniques sont intrinsèquement incrémentales, la qualité du compactage repose sur un ordre sur les classes ou sur les sélecteurs. L'algorithme incrémental peut difficilement optimiser cet ordre et on peut alors distinguer des versions incrémentales minimales de versions plus optimisées. Ainsi, avec la coloration, la version minimale consiste à calculer le premier indice libre pour un nouveau sélecteur, sans remettre en cause les autres. Le pire survient alors lorsqu'on rajoute un nouveau sélecteur pour `Object` : son indice sera $x + 1$ et tous les vecteurs auront la même taille, ce qui fait perdre l'avantage du morcellement de la matrice de coloration. Une version optimisée translaterait les indices des sélecteurs définis dans des sous-classes de `Object`, mais le coût serait beaucoup plus lourd. Enfin, les données associées aux classes et aux sélecteurs ne sont plus constantes : cela interdit de les compiler en ligne, dans les occurrences d'envoi de message bien sûr, mais aussi dans les fonctions génériques, à moins de recompiler ces fonctions lorsque les données considérées sont modifiées.

Dans tous les cas, la coloration doit être préférée pour sa meilleure gestion mémoire et le sens de son incrémentalité, qui favorise la définition « naturelle » des classes et de leurs méthodes.

Des solutions incrémentales ont été proposées dans le cadre des techniques de compactage à entrées uniques [AND 92]. [HOL 96, HOL 97] propose un cadre incrémental général, en rajoutant des indirections supplémentaires, au prix d'un surcoût non négligeable en temps et en espace. Mais une solution générale raisonnable consiste à invalider immédiatement certaines entrées de la table, par exemple toutes les entrées d'un sélecteur, pour renvoyer à une sélection dynamique en laissant le système se dégrader lentement jusqu'à ce qu'une réorganisation globale devienne possible ou nécessaire [DRI 95a].

En revanche, les techniques à base de *lookdown* se prêtent très bien aux modifications dynamiques, la modification dynamique d'un arbre de décision ne posant pas de difficulté. La structure de données de [QUE 97] a été conçue pour son incrémentalité. Les entrées multiples de [VIT 94] peuvent continuer à être surchargées sans poser d'autre problème qu'une lente dégradation. L'héritage multiple constitue pourtant une limitation pour chacune de ces techniques :

- pour la première, le codage de la hiérarchie pose un problème difficile : on ne connaît pas de codage équivalent pour un ordre partiel quelconque, ce qui explique en partie la restriction de [QUE 97] à l'héritage simple ; l'héritage multiple semble donc interdire les nœuds de type table ;
- pour la seconde, la définition d'une nouvelle classe peut amener à fusionner les tables de ses super-classes en introduisant des conflits entre sélecteurs, le même indice pouvant être utilisé, dans les super-classes, par des sélecteurs différents. La gestion de ce cas d'entrée multiple n'est pas prévue, pas plus que l'héritage multiple [17], mais il n'y a aucune difficulté technique pour le prendre en compte.

4.2.2. *Compilation séparée*

Dans l'état actuel des techniques, la compilation séparée et la compilation incrémentale sont indissociables. Ainsi, chaque modification incrémentale ne doit pas être propagée immédiatement — on ajoute rarement une méthode, un sélecteur ou une classe isolément — mais il faut attendre d'avoir fini un ensemble de modifications avant de les propager. On peut aussi envisager une recompilation paresseuse, attendant le premier envoi de message concernant une table, ou une entrée de table, invalidée. Inversement, le chargement dynamique d'un module constitué par un ensemble de définitions de classes et de méthodes peut être traité comme un ensemble de modifications incrémentales. Mais, à moins de faire une recompilation globale à chaque chargement de module, il faut s'attendre à une certaine dégradation des performances.

Même avec une définition restrictive de la notion de module, on ne voit pas bien comment les techniques de tables pourraient permettre une « édition de liens » des tables produites par de véritables compilations séparées. Pourtant, vu la taille des environnements de classes (figure 2), il faut bien envisager de ne les charger que partiellement, voire par morceaux, et il est déraisonnable d'archiver une indexation constante et globale de $c = 3241$ classes et de $s = 17342$ sélecteurs, si l'on ne doit se servir que d'une petite partie. En attendant d'hypothétiques techniques « translatables », la compilation (et la recompilation) des envois de message doit donc être suffisamment discrète pour se faire au chargement, sans occupation de mémoire irrécupérable. [HOL 96, HOL 97] font ainsi le pari d'une compilation purement incrémentale, au prix d'un surcoût important. La seule amélioration que nous pouvons suggérer consiste en une compilation incrémentale « par paquets ».

17. Dans la technique de Vitek, la fusion de deux tables peut produire des entrées doublement multiples, du point de vue des sélecteurs aussi bien que des méthodes et des classes, mais le test sur le sélecteur n'est nécessaire qu'en héritage multiple.

5. Schéma de compilation mixte

Un schéma de compilation satisfaisant consiste à tirer parti des avantages respectifs de chacune de ces techniques, en y prenant :

- les tables compactées (*lookin*) par l'heuristique de coloration, préférée pour sa meilleure incrémentalité, en particulier en cas d'ajout de classe ;
- les arbres de sélection (*lookdown*), pour améliorer encore la compilation incrémentale, en particulier en cas d'ajout de sélecteur, mais aussi dans l'espoir que le fait d'écarter quelques sélecteurs provoque des héritages de tables : en pratique, le *lookdown* sera appliqué aux sélecteurs n'ayant qu'une méthode, ainsi qu'aux sélecteurs d'`Object` n'ayant que deux méthodes (chaque arbre n'aura qu'un nœud) ;
- les entrées multiples, étendues pour traiter les conflits de sélecteurs produits par l'héritage multiple, pour mieux résister aux modifications ;
- le *lookup* complètement dynamique pour les entrées de table invalidées en dernière extrémité.

En pratique, chaque envoi de message pourra être 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 naturellement. Ainsi les sélecteurs des deux catégories seront implémentés respectivement comme suit :

```
(defun foo (o . args)
       (apply (lookin (sindex s) (class-of o)) 'foo o args))

(defun bar (o . args)
       (apply (lookdown (stree s) (class-of o)) 'bar o args))
```

Bien sûr, les accès aux indices et aux structures des sélecteurs, voire la totalité du *lookdown*, y seront 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 de coloration supportent la modification avec peu de réaménagement, on étend la structure des tables ; en particulier, rajouter une classe sans y définir de nouveaux sélecteurs consiste essentiellement 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.

5.1. *Mise en œuvre*

Chaque entrée des tables des classes et chaque feuille des arbres de décision contiennent un pointeur sur du code qui accepte uniformément comme arguments l'objet receveur, le sélecteur (sauf pour les méthodes de `Object`) et les arguments supplémentaires de la méthode, et qui provoque :

– soit le déclenchement de la méthode sur les arguments en question, après vérification du sélecteur (entrée simple) ;

– soit le déclenchement de l'erreur `message inconnu` (entrée vide) ;

– soit une sélection dynamique suivant un arbre de décision (entrée multiple) ;

– soit enfin une sélection complètement dynamique (entrée invalidée).

L'ensemble s'illustre par la fonction `lookdown` présentée plus haut et par la variante suivante de `lookin` :

```
(defun lookin (i c)
       (let ((tab (ctable c)))
            (if (< i (vlen tab))
                (vref tab i)
                #'dynamic-send)))
```

qui retourne, suivant les cas, la méthode cherchée, une fonction d'interprétation d'entrée multiple, `unknown-method` ou `dynamic-send`. Seule la sélection « dynamique » conserve une fonction d'envoi de message :

```
(defun dynamic-send (s o . args)
       (apply (lookup s (class-of o)) s o args))

(defun lookup (s c)
       (or (get-method s c)
           (let ((sc (super c)))
                (if sc
                    (lookup s sc)
                    #'unknown-method))))
```

Ainsi, les entrées multiples ou invalidées ne pénalisent pas les autres envois de message. Un sélecteur complètement invalidé pourra être compilé par un accès systématique à `dynamic-send` ou par un indice démesurément grand.

Toutes les fonctions concernées — `dynamic-send`, `unknown-method`, ainsi que les méthodes elles-mêmes ou les fonctions d'interprétation des entrées multiples — ont les mêmes arguments. C'est cette uniformité qui autorise toutes les passerelles entre les divers types de compilation : *lookin*, *lookdown* et *lookup*.

5.2. *Test de sous-classe*

Pour mettre en œuvre ce schéma, il faut disposer d'un test de sous-classe efficace et doucement dégradable. Une table de fermeture transitive occupe c^2 bits, ce qui n'est pas si considérable pour SMALLTALK (62 K-octets, soit une centaine d'octets par classe). Mais, contrairement à la place utilisée par la compilation qui est globalement proportionnelle au nombre de classes (et de sélecteurs), cette table de fermeture transitive a une taille quadratique : ainsi, pour VISUALAGE, elle occuperait 1,3 M-octets, soit 400 octets par classe ! Ces chiffres sont à comparer avec les tailles moyennes des tables de méthodes des classes, qui sont de l'ordre de μ entrées, soit 920 octets pour SMALLTALK et 1300 octets pour VISUALAGE.

De plus, la table de fermeture transitive se prête peu aux modifications dynamiques. Comme pour la coloration, on peut remplacer cette grande matrice par autant de vecteurs (de bits) qu'il y a de classes. On peut aussi tirer profit du fait qu'il s'agit d'un ordre, et non d'un préordre, pour jouer sur l'ordre des indices et n'occuper que $c(c+1)/2$ bits, soit 31 K-octets, chaque vecteur de bits ayant comme longueur l'indice de la classe. On suppose donc que l'ordre des indices des classes est une extension linéaire de la relation d'héritage : il suffit pour cela, dans les langages qui ne tolèrent pas de définir une classe avant ses super-classes, que l'indice de la classe soit son numéro d'ordre de définition. Le test de sous-classe se fera alors comme suit :

```
(defun subclassp (c1 c2)       ; c2  est la classe connue
       (let ((t1 (ctable-trans c1))
             (i2 (cindex c2)))
            (when (≤ i2 (bvlen t1))
                  (= 1 (bvref t1 i2)))))
```

La compilation en ligne des *lookdown* est relativement compacte. D'une part, à chaque nœud, on compare la classe inconnue du receveur à une classe connue, dont on accède de façon statique à l'indice. D'autre part, l'accès à la table de transitivité de la classe du receveur se fait une seule fois.

Il s'agit là d'un schéma assez naïf, dont l'incrémentalité est parfaite tant que l'on rajoute des classes, et qui vaut mieux que rien tant que le nombre de classes n'est pas trop élevé. Dans le cas contraire, il faut envisager des techniques plus élaborées de codage de hiérarchie (fermeture transitive, bornes inférieure ou supérieure) [AïT 89, AGR 91, CAS 93, HAB 95]. Citons pour finir une curiosité du test de sous-classe : c'est un cas particulier du problème de l'envoi de message. Il suffit, pour s'en convaincre, d'associer à chaque classe un sélecteur de même nom : `A` est alors sous-classe de `B` ssi `A` sait répondre au message `B`. L'optimal M de ces tables spécifiques est de l'ordre de $c^2/2$ (bits). L'utilisation du *lookdown* étant manifestement exclue, puisqu'il suppose ce test de sous-classe, il ne sert à rien de se ramener au problème général.

5.3. *Évaluation*

Le fait d'exclure de la coloration les 3313 sélecteurs qui n'ont qu'une seule méthode permet de réduire la taille des tables à 119 K. Au passage, 25 tables seulement ont été héritées. 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, devient alors bien inférieure à celle des autres techniques de compilation par tables (figure 5). L'intégration des entrées multiples dans la compilation initiale donne un résultat sensiblement différent de celui que l'on obtient avec une coloration simple (figure 7) : pour un seuil de 10 % , il y a une dégradation de près de 50 %, et l'amélioration maximale s'obtient avec un seuil de 11 % (elle est alors de plus de 40 %, mais la taille moyenne du *lookdown* par couple classe-sélecteur est de 6).

Le grand 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. Un codage plus efficace améliorerait cette approche, mais pour le nombre de classes considérées ici, celui-ci reste tout à fait acceptable. Pour VISUALAGE, le codage coûterait 600 K-octets, ce qui devient moins raisonnable.

6. La sélection multiple

La question de la sélection multiple, sur plusieurs arguments, des fonctions génériques de CLOS [STE 90], CECIL [CHA 93], DYLAN [App95] ou ILOGTALK [ILO95] n'a pas été abordée dans cet article. [KIC 90] décrit une technique de cache global incorporé à chaque fonction générique. La sélection multiple se ramène théoriquement à la sélection simple : au lieu de travailler sur un ordre dans l'ensemble de classes, on travaille sur le produit des ordres dans le produit cartésien des ensembles de classes (voir [DUC 89] pour le traitement de l'héritage en sélection multiple). Au passage, le paramètre c est élevé au carré (pour la sélection binaire, la plus courante), et le paramètre M est lui aussi multiplié par c, ce qui dissuade d'appliquer mécaniquement des techniques de tables. Cependant, le paramètre m/s n'augmente pas dans les mêmes proportions : en pratique, il augmente peu. Pour une classe du premier receveur, le sélecteur est défini dans très peu de classes du second receveur. L'argument sur les sélecteurs n'ayant que peu de méthodes semble donc favoriser les arbres de sélection du *lookdown*, d'autant plus qu'il s'accorde bien avec les fonctions génériques puisque chaque arbre de décision est attaché à un sélecteur. Inversement, il semble déraisonnable de chercher à utiliser les tables de *lookin*, vu l'ordre de grandeur de M. [CHE 94] et [DUJ 95] proposent ainsi des techniques à base d'automates assez proches des arbres de décision mais offrant, comme les tables, un problème de compactage. Mais [AMI 94] montre que les techniques de coloration s'appliquent aussi très bien à la sélection multiple. Le manque de place et la complexité des techniques en question, nous empêchent de les présenter : elles n'introduisent pourtant pas de nouveautés techniques, mais le passage à une dimension supérieure impose une coloration d'ordre supérieur. L'envoi de message se fait alors avec deux indirections par receveur, l'indice d'une classe pour le n-ième receveur d'un sélecteur donné étant lui-même coloré.

7. Conclusion et perspectives

La compilation des langages de programmation très évolués comme les langages de programmation fonctionnelle [SER 96], logique [DIA 95] ou par objets pose des problèmes difficiles, pour lesquels l'effet de telle ou telle optimisation n'est pas prévisible. Un minimum d'expérimentation est donc nécessaire. Cette expérimentation se fait en deux dimensions, l'espace et le temps. C'est sans doute la spécificité des objets d'avoir fait apparaître cette dimension spatiale, la plupart des autres familles de langages se contentant de mesures temporelles. L'essentiel de la littérature sur l'efficacité des langages à objets repose sur des mesures statiques de l'espace occupé par les structures de données nécessaires à la compilation de telle ou telle hiérarchie de classes, en général SMALLTALK. Ces analyses spatiales sont souvent incomplètes car elles sont basées sur des simulations et certains paramètres peuvent être oubliés. Les analyses temporelles restent, de leur côté, perfectibles. Il n'existe pas, en effet, dans la communauté objet, de jeux de test consensuels, analogues aux tests de Gabriel utilisés en programmation fonctionnelle. Nous n'avons pas dérogé à cette habitude, aussi contestable soit-elle : la mise en œuvre de tests temporels crédibles dépasse de très loin l'objectif de ce travail. Que le lecteur imagine simplement la difficulté d'implémenter effectivement et efficacement chacune des techniques décrites ici dans le compilateur d'un langage existant ! Il faut pour cela être implémenteur de SMALLTALK.

7.1. *Bilan*

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

- l'envoi de message dynamique, par recherche dans la hiérarchie (*lookup*) ou dans un arbre de décision (*lookdown*), ce dernier se prêtant bien aux fonctions génériques mais nécessitant un test de sous-classe efficace ;

- des techniques de compilation dynamiques, reposant sur des caches, en ligne (*lookhere*) ou globaux (*lookthere*) ;
- des techniques de compilation statiques (*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 (Queinnec), ou les entrées des tables pouvant être des arbres de décision (entrées multiples : Vitek).

Les tables présentent l'inconvénient de leur trop grande taille et seule la coloration, par l'heuristique que nous proposons, se prête, insuffisamment encore, aux modifications incrémentales. D'un autre côté, la technique de Queinnec n'étant pas généralisable à l'héritage multiple, les arbres de décision ne sont efficacement applicables qu'aux sélecteurs ayant peu de méthodes et aux entrées multiples peu surchargées.

Dans cet article, parallèlement à un exposé synthétique de ces différentes techniques, nous avons successivement décrit une variante efficace de la coloration, présenté un bilan mitigé d'une expérimentation de la technique des tables multiples, et enfin proposé un nouveau cocktail de ces différentes techniques. Cette nouvelle approche mixte repose sur une distinction, suivant des critères statiques, entre deux catégories de sélecteurs qui sont compilés respectivement par des arbres de décision ou par des tables de coloration, les entrées multiples étant réservées aux modifications incrémentales. Le nouveau schéma n'est pas une solution miracle mais il offre un bon compromis efficacité-incrémentalité puisqu'il est basé sur deux approches orthogonales tolérant bien, respectivement, l'ajout de sélecteur ou de classe.

Parmi les perspectives de ce travail, il faut citer l'adaptation de ces techniques dans plusieurs directions relativement orthogonales :

- la compilation fermée et définitive où toutes les optimisations sont souhaitables, en particulier la prise en compte du typage statique, de `self` et des méthodes qui retournent `self` ; [DRI 95b] donne une évaluation assez fine de la compilation en ligne de différentes techniques ;
- la compilation séparée de modules (qui restent à définir), et leur chargement dynamique sous la forme d'une véritable édition de liens des tables produites par les compilations séparées ;
- la sélection multiple des fonctions génériques, dans une optique de compilation ouverte qui n'a pas été traitée par [DUJ 95].

Enfin, la démarche consistant à compiler différemment les sélecteurs suivant leurs caractéristiques statiques (nombre de méthodes, de définitions, profondeur de définition, etc.) devrait être étendue à des caractéristiques dynamiques — fréquence et intensité d'utilisation d'une méthode ou d'une occurrence d'envoi de message — pour consommer de la place là où cela en vaut la peine.

7.2. *Pour une écologie du logiciel*

Les techniques considérées dans cet article occupent un espace qui est fonction des cinq paramètres principaux M, e, m, s et c (nombres de couples classe-sélecteur valides, d'envois de message, de méthodes, de sélecteurs et de classes). L'occupation mémoire se répartit différemment suivant ces paramètres, selon les diverses options. Ainsi, la compilation en ligne aura un fort coefficient sur e, alors que les fonctions génériques le placeront sur s. Comme la compilation en ligne est inadaptée à la compilation incrémentale, nous avons privilégié l'approche de la compilation par des fonctions génériques, bien qu'elle ne permette pas de réduire le nombre d'appels dans le code généré.

Un autre paramètre à prendre en compte est la taille totale du code, C. Si M est négligeable devant C — c'est semble-t-il le cas de NEXTSTEP —, un compactage limité suffit certainement. Mais qu'en est-il si tous les éléments du compilateur ont été conçus en suivant le même raisonnement ? Inversement, que se passe-t-il si tous les autres éléments du compilateur se mettent, par imitation, à consommer de la mémoire ? Or si C vaut 3,1 M-octets pour les $c = 310$ classes de NEXTSTEP, que vaut-il alors pour les $c = 3241$ classes de VISUALAGE (figure 2) ? 35 M-octets ? Est-ce bien raisonnable ?

L'évolution de l'informatique suit une spirale vicieuse d'amélioration exponentielle des possibilités des processeurs et des besoins des logiciels. Les développeurs de logiciels pensent constamment en termes de génération suivante de processeurs, et les utilisateurs de logiciels passent leur temps à rêver à cette génération suivante en pensant au bon vieux temps. L'exponentielle est un aussi mauvais choix en économie, en écologie ou en calculabilité : il est certain que la version de SMALLTALK pour le processeur suivant proposera plus de classes, donc de plus grandes tables et un code en proportion. Enfin, il faut craindre que le compilateur ne soit encore « optimisé » par une compilation en ligne qui le gonflera de 50 %, à code source égal. Le passage des architectures CISC aux architecture RISC a aussi plus que doublé la taille de certains exécutables. [HOL 96, HOL 97] est ainsi prêt à doubler la taille des tables pour éliminer une indirection dans son schéma de compilation incrémentale. La déontologie du développeur de logiciels, en particulier des compilateurs, devrait être axée sur une approche écologique des deux principales ressources de l'informatique, l'espace et le temps, et sur la recherche d'un compromis. [SER 96] a, par exemple, développé une heuristique pour décider si un appel fonctionnel doit être compilé en ligne ou non, pour ne pas augmenter inconsidérément la taille du code en échange d'une médiocre optimisation.

Finalement, ce problème de la compilation de l'envoi de message semble autant sociologique que technologique et fait espérer une prochaine écologie du logiciel. Les autres obstacles à l'efficacité sont, eux, souvent psychologiques : ainsi de l'héritage multiple, dont l'une des tares serait l'inefficacité, alors que son impact est globalement très faible au niveau de l'envoi de message (un peu plus de trous dans les tables, un codage de la fermeture transitive moins compact). Il est vrai que nous n'avons pas traité, en même temps que de l'envoi de message, de l'accès aux attributs (ou variables d'instances) qui peut se traiter par les mêmes techniques, en remplaçant le code des méthodes par la position de l'attribut dans l'objet, avec cette circonstance simplificatrice, en cas d'encapsulation orthodoxe, que les attributs sont toujours typés par `self`.

Remerciements

L'auteur tient à remercier P. André et J.-C. Royer de lui avoir fourni, outre un joli problème, les données de SMALLTALK utilisées pour tester une première version de l'heuristique de coloration et Ch. Dony de lui avoir fourni les données ayant servi aux mesures citées ici.

Références

[ABA 96] M. ABADI et L. CARDELLI. *A theory of objects.* Monographs in Computer Science. Springer Verlag, 1996.

[AGE 95] O. AGESEN et U. HÖLZLE. « Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages ». Dans *Proc. OOPSLA'95*, pages 91–107. ACM Press, 1995.

[AGR 91] R. AGRAWAL, A. BORGIDA, et H.V. JAGADISH. « Efficient management of transitive relationships in large data-bases including is-a hierarchies ». Rapport Technique DCS-TR-286, Rutgers University, 1991.

[AMI 94] E. AMIEL, O. GRUBER, et E. SIMON. « Optimizing multi-methods dispatch using compressed dispatch tables ». Dans *Proc. OOPSLA'94*. ACM Press, 1994.

[AND 92] P. ANDRÉ et J.-C. ROYER. « Optimizing method search with lookup caches and incremental coloring ». Dans *Proc. OOPSLA'92*, pages 110–126. ACM Press, 1992.

[App95] Apple Computer. « DYLAN *reference manual* », 1995.

[AïT 89] H. AÏT-KACI, R. BOYER, P. LINCOLN, et R. NASR. « Efficient implementation of lattice operations ». *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

[BER 70] C. BERGE. *Graphes et Hypergraphes.* Dunod, 1970.

[CAR 90] B. CARRÉ et J-M. GEIB. « The point of view notion for multiple inheritance ». Dans *Proc. OOPSLA/ECOOP'90*. ACM Press, 1990.

[CAS 93] Y. CASEAU. « Efficient handling of multiple inheritance hierarchies ». Dans *Proc. OOPSLA'93*, pages 271–287. ACM Press, 1993.

[CAS 96] G. CASTAGNA. « Le modèle fondé sur la surcharge : une visite guidée ». *Technique et Science Informatiques*, 15(6):673–708, 1996.

[CHA 92] C. CHAMBERS et D. UNGAR. *Lisp and Symbolic Computation*, 4(3), 1992. Numéro spécial consacré au langage SELF.

[CHA 93] C. CHAMBERS. « The CECIL Language, Specification and Rationale ». Technical Report 93-03-05, University of Washington, 1993.

[CHE 94] W. CHEN, V. TURAU, et W. KLAS. « Efficient dynamic look-up strategy for multi-methods ». Dans Tokoro and Pareschi [TOK 94].

[COX 87] B. COX. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, 1987.

[DEU 84] L.P. DEUTSCH et A. SCHIFFMAN. « Efficient implementation of the SMALLTALK-80 system ». Dans *Proc. of Principles of Prog. Lang. (POPL'84)*, 1984.

[DIA 95] D. DIAZ. « *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système* `clp(FD)` ». Thèse d'informatique, Université d'Orléans, 1995.

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

[DRI 93] K. DRIESEN. « Selector table indexing and sparse arrays ». Dans *Proc. OOPSLA'93*. ACM Press, 1993.

[DRI 95a] K. DRIESEN et U. HÖLZLE. « Minimizing Row Displacement Dispatch Tables ». Dans *Proc. OOPSLA'95*, pages 141–155. ACM Press, 1995.

[DRI 95b] K. DRIESEN, U. HÖLZLE, et J. VITEK. « Message Dispatch on Pipelined Processors ». Dans W. OLTHOFF, éditeur, *Proc. ECOOP'95*, numéro 952 dans LNCS, pages 253–282. Springer-Verlag, 1995.

[DUC 89] R. DUCOURNAU et M. HABIB. « La multiplicité de l'héritage multiple ». *Technique et Science Informatiques*, 8(1):41–62, 1989.

[DUC 91] R. DUCOURNAU. « Y3 : YAFOOL*, le langage à objets* ». Sema Group, 1991.

[DUC 92] R. DUCOURNAU, M. HABIB, M. HUCHARD, M.-L. MUGNIER, et A. NAPOLI. « L'héritage multiple dans tous ses états ». Rapport de Recherche 92-021, L.I.R.M.M., Montpellier, 1992.

[DUC 94] R. DUCOURNAU, M. HABIB, M. HUCHARD, et M.-L. MUGNIER. « Proposal for a monotonic multiple inheritance linearization ». Dans *Proc. OOPSLA'94*. ACM Press, 1994.

[DUC 95] R. DUCOURNAU, M. HABIB, M. HUCHARD, M.-L. MUGNIER, et A. NAPOLI. « Le point sur l'héritage multiple ». *Technique et Science Informatiques*, 14(3):309–345, 1995.

[DUJ 95] E. DUJARDIN. « *Flexibilité, robustesse et efficacité des modèles de données à objets* ». Thèse d'informatique, Université Paris VI, 1995.

[ELL 90] M.A. ELLIS et B. STROUSTRUP. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US, 1990.

[GOL 83] A. GOLDBERG et D. ROBSON. SMALLTALK *: the language and its implementation.* Addison-Wesley, Reading, MA, 1983.

[HAB 95] M. HABIB, M. HUCHARD, et L. NOURINE. « Embedding Partially ordered Sets into Chain-Products ». Dans G. ELLIS, R. A. LEVINSON, A. FALL, et V. DALH, éditeurs, *Proc. KRUSE'95*, pages 147–161, 1995.

[HöL 91] U. Hölzle, C. Chambers, et D. Ungar. « Optimizing dynamically-typed object-oriented languages with polymorphic inline caches ». Dans P. America, éditeur, *Proc. ECOOP'91*, numéro 512 dans LNCS, pages 21–38. Springer-Verlag, 1991.

[HOL 96] W. Holst et D. Szafron. « Inheritance management and method dispatch in reflexive object-oriented languages ». Technical Report TR-96-27, University of Alberta, Edmonton, CA, 1996.

[HOL 97] W. Holst et D. Szafron. « A general framework for inheritance management and method dispatch in object-oriented languages ». Dans *Proc. ECOOP'97*, LNCS. Springer-Verlag, 1997.

[HUA 92] S.-K. Huang et D.-J. Chen. « Coloring approaches for method dispatching in object-oriented programming systems ». Dans *Proc. ICSA'92*, pages 39–44, 1992.

[HUL 85] J.M. Hullot. « CEYX version 15 ». Technical Report 44, 45 et 46, I.N.R.I.A., 1985.

[ILO95] ILOG, Gentilly. « Ilog Talk *reference manual, Version 3.2* », 1995.

[KIC 90] G. Kiczales et L. Rodriguez. « Efficient Method Dispatch in PCL ». Dans *Proc. ACM Conf. on Lisp and Functional Programming*, 1990.

[QUE 93] Ch. Queinnec. Designing MEROON v3. Dans Ch. Rathke, J. Kopp, H. Hohl, et H. Bretthauer, éditeurs, *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP'93 Workshop.* GMD, Rapport de recherche 778, 1993.

[QUE 97] Ch. Queinnec. « Fast and Compact Dispatching for Dynamic Object-Oriented languages ». *Information Processing Letters*, 1997. (to appear).

[SER 96] M. Serrano. « *Vers une compilation portable et performante des langages fonctionnels* ». Thèse d'informatique, Université Paris VI, 1996.

[STE 90] G.L. Steele. *Common Lisp, the Language.* Digital Press, second édition, 1990.

[TOK 94] M. Tokoro et R. Pareschi, éditeurs. *Proceedings of the Eighth European Conference on Object-Oriented Programming, ECOOP'94*, numéro 821 dans LNCS. Springer-Verlag, 1994.

[VIT 94] J. Vitek et R.N. Horspool. « Taming message passing: efficient method look-up for dynamically typed languages ». Dans Tokoro and Pareschi [TOK 94], pages 432–449.

Roland Ducournau *est professeur à l'Université Montpellier II depuis 1994. Il était auparavant ingénieur dans la SSII Sema Group, où il a conçu et réalisé l'environnement de développement de systèmes à base de connaissances* Y3 *autour du langage d'objets* YAFOOL. *Son domaine de recherche est « l'objet », sous toutes ses formes, mais surtout du point de vue de ses capacités de représentation : en représentation des connaissances, bien sûr, mais aussi en programmation ou en bases de données. La structure classificatoire des systèmes d'objets, avec l'héritage et la classification, est au cœur de ses travaux.*