Le hachage parfait fait-il un parfait test de sous-typage ?

Roland Ducournau

L.I.R.M.M. – CNRS et Université Montpellier 2
161, rue Ada – 34392 Montpellier cedex 5
Roland.Ducournau@lirmm.fr

RÉSUMÉ. *Le test de sous-typage est une difficulté importante de l'implémentation des langages à objets. Beaucoup de techniques ont été proposées mais aucune ne satisfait pleinement les cinq exigences que nous avons identifiées : temps constant, espace linéaire, héritage multiple, chargement dynamique et expansion en ligne (*inlining*). Dans cet article, nous proposons d'appliquer une technique bien connue, le* hachage parfait*, et nous montrons qu'elle donne de plutôt bonnes réponses, presque parfaites, à ce cahier des charges. De plus, dans le cadre des langages à la* JAVA*— caractérisés par un héritage simple des classes et un sous-typage multiple des interfaces — le hachage parfait a l'avantage de s'appliquer, en même temps, à l'invocation de méthodes lorsque le receveur est typé par une interface.*

ABSTRACT. *Subtype tests are an important issue in the implementation of object-oriented programming languages. Many techniques have been proposed but none of them perfectly fullfills the five requirements that we have identified: constant-time, linear-space, multiple inheritance, dynamic loading and inlining. In this paper, we propose to apply a well known technique,* perfect hashing*, and we show that it responds rather well in meeting these requirements. Furthermore, in the framework of* JAVA *like languages—characterized by single inheritance of classes and multiple subtyping of interfaces—perfect hashing applies also, at the same time, to method invocation when the receiver is typed by an interface.*

MOTS-CLÉS : *coloration, coercition de types, hachage parfait, héritage simple, héritage multiple, interfaces, langage à objets, tables de méthodes, test de sous-typage*

KEYWORDS: *casting, coloring, downcast, dynamic loading, linking, multiple inheritance, object-oriented languages, perfect hashing, single inheritance, subtype test, virtual function tables*

1. Introduction

Le besoin de test de sous-typage est un trait original de la programmation par objets. Etant donné une entité x d'un type statique C — le type *source* —, le programmeur ou le compilateur peut vouloir vérifier que la valeur liée à x est bien une instance d'un autre type D — le type *cible* —, en général pour appliquer à x une opération qui n'est pas connue par C. Dans le cas le plus simple, C et D sont des classes et D est une sous-classe de C (noté $D \preceq C$). Cependant, dans un cadre de typage dynamique, x est statiquement non typé et le type source est alors la racine $\top$ de la hiérarchie de types. De même, l'héritage multiple fait que D n'est pas toujours un sous-type de C. Ces deux exceptions n'ont pas d'influence sur le mécanisme qui implique seulement la valeur liée à x et le type cible D. Le test de sous-typage n'est donc qu'une vérification dynamique de type, mais « à un sous-type près ».

Le test de sous-typage peut être *explicite* ou *implicite*. Les langages de programmations offrent différentes constructions syntaxiques permettant au programmeur de faire des vérifications. La *coercition de types* — souvent appelée *cast* ou *downcast* dans le jargon des programmeurs C++ ou JAVA— en est un exemple typique, au travers de syntaxes diverses : `dynamic_cast` en C++, syntaxe parenthésée de JAVA (la syntaxe de C, interdite en C++ !), `typecase` en THETA (Liskov *et al.*, 1995) ou la *tentative d'affectation* d'EIFFEL (Meyer, 1997). Le point commun est que le type cible est *statique*, c'est une constante de l'expression. Quand le langage est doté d'un *protocole de méta-objets* (*Meta-Object Protocol*, MOP) (Kiczales *et al.*, 1991) — comme CLOS, SMALLTALK ou, dans une moindre mesure, JAVA— le type cible peut être *dynamique*, c'est-à-dire calculé. Par exemple, le programmeur peut vérifier que la valeur de x est bien instance du type dynamique de la valeur de y. De tels tests sont traités par des opérateurs booléens comme `isInstanceOf`, par exemple `x.isInstanceOf(y.class)`. Le test peut aussi être implicite, quand il est généré par le compilateur à l'insu du programmeur. Cela arrive dès que le langage est doté d'un typage qui n'est pas complètement sûr : par exemple, le sous-typage des tableaux en JAVA ou la redéfinition covariante en EIFFEL qui, indépendamment de leur intérêt (Meyer, 1997; Ducournau, 2002b), nécessitent une vérification dynamique de type. Les échecs du test sont diversement traités : par le signalement d'une exception (JAVA, C++ mais pour les références seulement, test implicite), une valeur nulle (EIFFEL, C++ pour les pointeurs), ou de façon booléenne (`typecase`, `isInstanceOf`). Ces variantes — suivant que la cible est *statique* ou *dynamique*[1], ou suivant que le test est *implicite* ou *explicite* — n'ont pas d'influence directe sur l'implémentation du mécanisme. Une cible statique permet cependant de traiter les données qui la concernent comme des constantes, voire comme des valeurs immédiates, ce qui évite un accès mémoire.

En toute généralité, l'implémentation du test de sous-typage revient à une sorte de codage de la hiérarchie de types. Malgré les nombreux travaux consacrés à ce problème, l'implémentation générale et efficace du test de sous-typage demeure un

1. L'usage de ces termes est ici sans rapport avec les opérateurs `static_cast` et `dynamic_cast` de C++, qui sont tous deux *statiques*.

	i	ii	iii	iv	v
(Schubert *et al.*, 1983)	x	x	–	–	x
(Cohen, 1991)	x	x	–	x	x
(Vitek *et al.*, 1997)	x	x	x	–	x
(Zibin *et al.*, 2001)	x	x	x	–	x
(Alpern *et al.*, 2001b)	x	–	x	x	x
(Click *et al.*, 2002)	–	x	x	x	x
C++ (g++)	–	?	x	x	–
SMART EIFFEL	–	x	x	–	?
accès direct	x	–	x	x	x
hachage parfait	x	x	x	x	x

Tableau 1. *Confrontation de différentes techniques aux cinq critères d'évaluation*

problème ouvert. A notre connaissance, aucune implémentation ne vérifie les cinq critères suivants : i) *temps constant*, ii) *espace linéaire*, iii) compatibilité avec l'*héritage multiple*, iv) le *chargement dynamique* et v) l'*expansion en ligne*. i) Un mécanisme en temps constant doit être de loin préféré, vu la difficulté qu'il y a à démontrer qu'un mécanisme en temps non constant est meilleur. ii) Les mécanismes objet sont en général implémentés avec des tables qui occupent un espace linéaire dans la taille de la fermeture transitive du graphe de spécialisation, donc quadratique dans le nombre de classes dans le pire des cas. *A contrario*, l'implémentation par sous-objets de C++ est cubique dans le nombre de classes (Ducournau, 2002a). iii) Les modèles complexes du « monde réel » se contentent mal de l'héritage simple, surtout en typage statique : l'héritage multiple est une nécessité, au moins dans sa forme dégradée du sous-typage multiple de JAVA et DOTNET. iv) Le chargement dynamique est un vieille caractéristique des plate-formes LISP et SMALLTALK, remise au goût du jour par JAVA et DOTNET. Mais l'implémentation d'un langage plus traditionnel comme C++ est parfaitement compatible avec le chargement, ou l'édition de liens, dynamiques. Le chargement dynamique impose une incrémentalité des algorithmes qui est souvent incompatible avec l'héritage multiple, ou avec l'efficacité. v) Enfin, l'expansion en ligne (*inlining*) consiste à économiser un appel de fonction en insérant le code de la fonction à l'endroit de l'appel. C'est un facteur d'efficacité temporelle et un risque d'inefficacité spatiale : la séquence de code à expanser doit restée suffisamment courte. La table 1 confronte différentes techniques examinées plus loin avec ces cinq critères.

Dans cet article, nous présentons une implémentation du test de sous-typage qui a l'avantage de satisfaire ces cinq critères. Cette proposition est basée sur une technique éprouvée, le *hachage parfait* (*perfect hashing*) (Sprugnoli, 1977; Mehlhorn *et al.*, 1990; Czech *et al.*, 1997), qui est une optimisation statique et une variante en temps constant de tables de hachage traditionnelles comme *linear probing* (Knuth, 1973; Vitter *et al.*, 1990). L'article est organisé comme suit. La section 2 présente une technique bien connue, due à (Cohen, 1991), qui satisfait tous les critères sauf l'héritage multiple, ainsi que son extension à l'héritage multiple, proposée initialement par (Vitek *et al.*, 1997), qui est incompatible avec le chargement dynamique. La section suivante examine les approches par tables de hachage et leur optimisation par le hachage par-

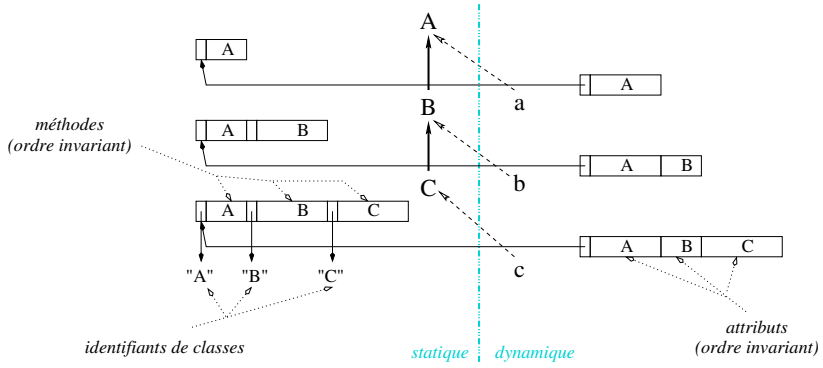

Figure 1. *Implémentation des objets et des tables de méthodes en héritage simple : 3 classes A, B et C avec leurs instances respectives, a, b et c.*

fait. L'application à JAVA est détaillée. La section 4 analyse ensuite les performances spatiales des différentes techniques examinées. La section suivante décrit quelques travaux connexes et l'article se termine par des conclusions et quelques perspectives.

2. Héritage multiple ou chargement dynamique, il faut choisir

L'implémentation efficace des objets bute souvent sur une incompatibilité entre l'héritage multiple et la compilation séparée alliée au chargement dynamique.

2.1. *Héritage simple*

En compilation séparée des langages statiquement typés, la liaison tardive est en général implémentée au moyen de tables de méthodes. Un objet est implémenté comme un tableau de ses attributs, auxquels s'ajoute un pointeur vers la table de méthode de sa classe. Lorsque l'héritage simple et le typage statique se conjuguent, cette implémentation, schématisée dans la figure 1, vérifie deux invariants essentiels, de *référence* aux objets et de *position* des propriétés. L'implémentation de la sous-classe se contente d'étendre les tables de son unique super-classe.

Contrairement à l'implémentation des objets, le test de sous-typage est indépendant du caractère statique ou dynamique du typage. Une technique classique, due à (Cohen, 1991) et souvent réutilisée ou réinventée (Queinnec, 1997; Alpern *et al.*, 2001b), satisfait les cinq critères, à l'exception de l'héritage multiple. Ce sera la base d'extensions ultérieures. La technique consiste à affecter une position $\chi(C)$ à chaque classe C, dans sa table de méthodes et dans celle de ses sous-classes : l'entrée correspondante des tables contient l'identifiant id_C de la classe. Les indices χ sont régis par le même invariant de position que les méthodes et les attributs. Etant donné un objet instance directe de la classe C, son type dynamique, cet objet est instance de

D, le type cible, si la table de méthodes de C, notée tab_C, contient, à la position $\chi(D)$ l'identifiant id_D :

$$C \preceq D \Leftrightarrow tab_C[\chi(D)] = id_D \quad [1]$$

Dans le pseudo-code introduit par (Driesen, 2001) — auquel nous renvoyons le lecteur pour l'interprétation de son évaluation —, le test se traduit alors par :

```
load [object + #tableOffset], table
load [table + #targetColor], classId
comp classId, #targetId                                    2L + 2
bne #fail
// succès
```

Il nécessite $2L + 2$ cycles, où L est la latence de chargement (environ 2-3 cycles). Cependant, un détail doit être soigneusement examiné : $tab_C[\chi(D)]$ n'est correct que si $\chi(D)$ ne déborde pas du tableau tab_C. Un test apparaît donc nécessaire. De nombreux auteurs l'évitent en utilisant des tableaux de taille fixe — par exemple, (Click *et al.*, 2002) qui attribue la technique à (Pfister *et al.*, 1991) — mais cela provoque un surcoût spatial d'autant moins raisonnable qu'il existe une solution simple. En effet, il suffit d'allouer les tables de méthodes dans un espace qui leur est réservé et qui ne contient donc que des adresses de méthode et des identifiants de classe. La dernière table de l'espace sera complétée, sur la droite, par une valeur non ambiguë, jusqu'à une position supérieure à $max_C(\chi(C))$. Les adresses étant des multiples de 4, des identifiants impairs garantissent que, dans cet espace exclusif, id_D ne peut se trouver qu'à la position $\chi(D)$ de quelque tab_C.

Clairement, la technique vérifie les cinq critères : i) le temps est non seulement constant mais on peut difficilement espérer mieux ; ii) l'espace est exactement d'un mot mémoire, voire un demi-mot si l'on utilise des petits entiers, par arc de la fermeture transitive et réflexive de la relation de spécialisation ; iv) au chargement d'une classe, on lui affecte un nouvel identifiant ainsi que la première position disponible dans la table de son unique super-classe directe ; v) le code est court (4 instructions) et `targetColor` et `targetId` peuvent être des constantes, si le test est *statique*.

2.2. *Héritage multiple*

En examinant la hiérarchie en losange de la figure 2, on se convainc aisément que la technique d'implémentation et de sous-typage de l'héritage simple ne fonctionne plus en héritage multiple. L'invariant de position est impossible pour la classe D, puisque des éléments (attributs, méthodes, classes) de B et de C se trouveraient à la même position dans l'implémentation de D. L'implémentation de C++ et la coloration sont des solutions qui abandonnent un critère ou relâchent un invariant.

2.2.1. *Implémentation par sous-objets*

L'implémentation de C++ rend les invariants de référence et de position relatifs au type statique. Les conséquences sont abondamment discutées ailleurs (Ellis

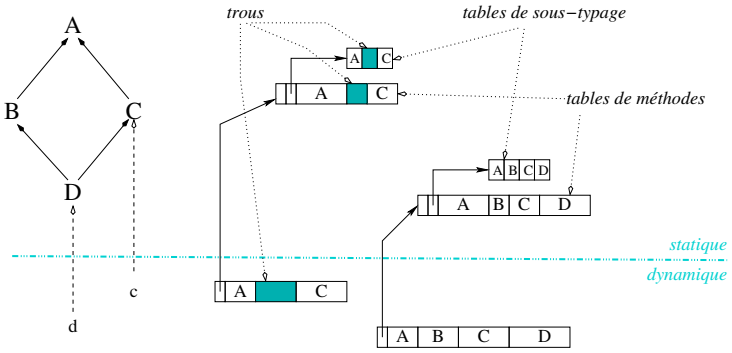

Figure 2. *Coloration unidirectionnelle appliquée aux classes, méthodes et attributs. La coloration de classes apparaît dans une table séparée, mais elle est en réalité incluse dans la table de méthodes.*

et al., 1990; Lippman, 1996; Ducournau, 2002a). La confrontation avec les cinq critères montre que i) le temps pris par les mécanismes primitifs est constant, mais plus long, ii) l'espace devient cubique dans le pire des cas, v) les séquences de code sont plus longues. En ce qui concerne le test de sous-typage, aucune technique connue n'est satisfaisante au vu des cinq critères. La plupart des compilateurs, par exemple `g++`, compilent `dynamic_cast` par un appel de fonction, et des *benchmarks* précis montrent une sensible corrélation entre le temps du test et le nombre de classes (Privat *et al.*, 2005). Seuls les critères iii) et iv) résistent. De plus, les sous-objets enlèvent au problème du test de sous-typage son caractère booléen. En cas de réussite, il faut en effet que le test retourne la position relative du sous-objet correspondant au type cible.

Une technique vient naturellement à l'esprit, qu'il faut d'abord examiner avant de l'éliminer car elle est impraticable : une table à *accès direct*. Dans un contexte de chargement dynamique, où l'on ne sait jamais quelles classes seront chargées par la suite — par exemple D après B et C, dans la figure 2 — on numérote injectivement les classes, dans l'ordre du chargement, et chaque classe possède une table contenant, pour chaque classe précédemment chargée, la position relative du sous-objet correspondant, si c'est une super-classe, ou sinon, une valeur distinguée signifiant l'échec du test. L'espace nécessaire est en N^2, où N est le nombre de classes, qui peut atteindre 10000, au moins dans les bibliothèques, si ce n'est dans les programmes effectifs. Outre que le critère ii) n'est pas respecté, ce serait vraiment rédhibitoire.

2.2.2. *Coloration*

La coloration peut se définir comme la technique d'optimisation permettant de maintenir en héritage multiple les deux invariants, de référence et de position, de l'héritage simple. Elle a été introduite, appliquée aux méthodes sous le nom de *selector coloring*, par (Dixon *et al.*, 1989), puis appliquée aux attributs par (Pugh *et al.*, 1990) et aux classes par (Vitek *et al.*, 1997), sous le nom de *pack encoding*. La figure 2 illustre la coloration sur une hiérarchie en losange. (Ducournau, 2006) fait une syn-

thèse de l'approche. La coloration de classes, ou *pack encoding*, est donc une extension directe de la technique de Cohen. Le code du test est exactement celui qui précède et tous les critères sont satisfaits, pour l'espace au prix d'une légère augmentation due aux trous, sauf iv) la compatibilité avec le chargement dynamique. On a clairement échangé l'héritage multiple contre le chargement dynamique.

3. Approche par tables de hachage

Dans ce genre de situation, l'informaticien a l'habitude de résoudre le problème avec des structures d'association, plus gourmandes en temps mais beaucoup moins en espace, comme les tables de hachage. Le lecteur en trouvera une synthèse dans (Knuth, 1973; Vitter *et al.*, 1990).

3.1. *Première analyse*

Une technique simple et classique est *linear probing*. La table est un tableau de H paires clés/valeurs, alternativement les clés entières dans un intervalle $[0..N-1]$ (les identifiants des classes) et les positions relatives des sous-objets. A ce tableau est associée une fonction de hachage $h : \mathbf{N} \rightarrow [0..H-1]$. Un identifiant de classe est recherché linéairement dans la table, à partir de sa valeur de hachage, jusqu'à ce qu'il soit trouvé où que l'on tombe sur une entrée vide. Cette technique est très efficace lorsque le taux de remplissage de la table est suffisamment faible, 1/2 par exemple. Or il va s'agir d'associer une table à chaque classe et le nombre d'entrées occupées est le nombre de super-classes de la classe (celle-ci incluse). Un premier choix s'impose donc : faut-il des tables de hachage uniformes, H ne dépendant pas de la classe, ou un paramètre H_C par classe C ? Avec des tables uniformes, le hachage peut être fait statiquement, au moins quand le test est *statique*, mais les tables n'optimisent ni le temps d'accès, ni la taille totale. On s'éloigne clairement d'un des deux critères i) temps constant et ii) espace linéaire et, si ce choix d'un hachage uniforme devait être retenu, une technique alternative, celle du *chaînage séparé*, serait sans doute préférable. Dans tous les cas, l'échec du test est pénalisé par rapport à sa réussite. Inversement, si la taille des tables n'est pas uniforme, le hachage doit être fait dynamiquement — il dépend d'un paramètre qui doit être recherché dans la table des méthodes, donc un accès mémoire en plus — mais la table elle-même peut être optimisée.

Un deuxième point est à examiner : où mettre la table de hachage ? Lorsqu'elle est de taille fixe, il est possible de la mettre dans la table de méthodes, sans se poser trop de questions. Avec des tables de longueur variable, le problème est un peu différent si l'implémentation sous-jacente vérifie l'invariant de position, car le paramètre H_C ne vérifie pas cet invariant. Il y a cependant une solution simple : la table de hachage peut être expansée dans les indices négatifs de la table de méthodes (à condition bien entendu que celle-ci ne soit pas déjà bidirectionnelle). Dans une implémentation par sous-objets, il apparaît que la meilleure solution consiste à faire pointer la table de hachage par chaque table de méthodes.

Reste un dernier point, la fonction de hachage h. Lorsque le hachage est non uniforme, il s'agit d'une famille de fonctions h_C paramétrées par la classe, ou plutôt par le paramètre H_C : $h_C(x) = \texttt{hash}(x, H_C)$. Le choix de la fonction `hash` est crucial, puisque le hachage est maintenant toujours dynamique, et que le critère v) voudrait que l'on puisse l'expanser en ligne. Une fonction mono-instruction, voire mono-cycle, s'impose. Deux fonctions viennent rapidement à l'esprit : `mod`, le reste de la division entière, et `and`, la conjonction booléenne sur la représentation binaire des entiers. Chacune correspond à une instruction des processeurs, mais seule la deuxième ne prend qu'un cycle : sur certains processeurs comme le Pentium IV, la division entière est même traitée par l'unité de calcul flottant. Question espace, les deux vérifient $0 \leq h_C(x) \leq H_C$.

3.2. *Hachage parfait*

Parmi les alternatives envisagées dans l'analyse précédente, une approche consiste en un hachage dynamique, effectué à partir d'une opération efficace comme `and`, et en optimisant séparément chaque table pour minimiser les collisions. Or le contenu de chaque table de hachage doit être calculé une fois pour toutes, au chargement de la classe, à partir des identifiants des super-classes, qu'il faut supposer déjà chargées[2]. La table sera ensuite utilisée en lecture seule, sans ajout ni retrait. Il est donc possible de l'optimiser, pour obtenir, *in fine*, une table de hachage dite *parfaite*, c'est-à-dire garantie sans collisions sur les clés qu'elle contient (Sprugnoli, 1977; Czech *et al.*, 1997). La définition du problème est simple. Soit $I_C = \{id_D \mid C \preceq D\}$, l'ensemble des identifiants des super-classes de C, C incluse : il s'agit de déterminer le plus petit entier H_C tel que h_C soit injective sur I_C. Le test s'obtient en remplaçant χ par h_C dans [1] :

$$C \preceq D \Leftrightarrow tab_C[h_C(D)] = id_D \qquad [2]$$

Lorsque $h_C(x) = \texttt{mod}(x, H_C)$, la table est de taille H_C et $H_C < id_C = max(I_C)$. Avec `and`, la taille est $H_C + 1$ et $H_C < 2id_C$.

Dans une implémentation qui vérifie l'invariant de référence, on obtient le code suivant, qui est certes plus important, en instructions et en cycles, que le test de Cohen, mais qui reste très compétitif :

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #targetId, h, h
sub table, h, table                                   3L + 4
load table, id
comp #targetId, id
bne #fail
```

2. Ce ne serait pas toujours le cas dans les machines virtuelles JAVA (Alpern *et al.*, 2001b), mais les raisons n'apparaissent pas clairement. Les déchargement et rechargement de classes ne font pas partie de notre cahier des charges, mais ces mécanismes pourraient vraisemblablement être spécifiés de façon à être compatibles avec la technique proposée ici.

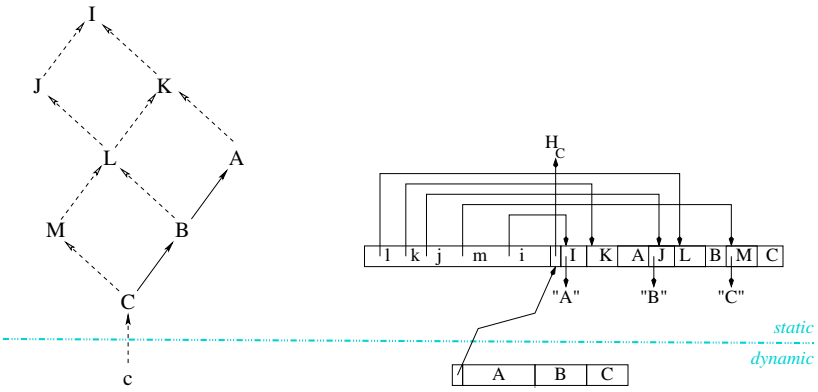

Figure 3. *Hiérarchie à la* JAVA *avec 3 classes A, B, C, 5 interfaces, I, J, K, L, M et une instance c. L'objet est le même qu'en héritage simple, mais la table de méthodes est bidirectionnelle. Les indices positifs contiennent la table de méthodes elle-même, comme en héritage simple, et les indices négatifs contiennent la table de hachage : identifiants de classe et pointeurs dans la table de méthodes. Enfin, l'indice 0 contient le paramètre H_C.*

Dans le cas d'une implémentation par sous-objets, le code est pas mal compliqué par l'ajustement de pointeurs, mais son expansion en ligne reste envisageable. Voir (Ducournau, 2005) pour plus de détails.

3.3. *Application à* JAVA *et à* DOTNET

La caractéristique principale de JAVA et DOTNET est leur système de types, qui allie l'héritage simple des classes (relation `extends` entre classes) au sous-typage multiple des interfaces (relation `extends` entre interfaces et `implements` entre classes et interfaces). Le type dynamique d'un objet est toujours une classe.

Pour le test de sous-typage, la technique de Cohen s'applique lorsque le type cible est une classe. Lorsque c'est une interface, le hachage parfait s'applique mais il faut aussi envisager l'accès direct, puisqu'il ne s'agit plus ici d'une table d'ordre $N \times N$, mais d'ordre $N_c \times N_i$, où N_c et N_i sont respectivement les nombres de classes et d'interfaces, avec donc $N = N_c + N_i$.

Mais les interfaces posent un autre problème d'implémentation, lorsque le receveur est typé par une interface : c'est l'opérateur `invokeinterface` de la JVM. L'invariant de position ne tient plus, puisque la même méthode de l'interface peut avoir des positions différentes dans les différentes classes qui implémentent l'interface. Il faut donc recourir à d'autres techniques. Or, la technique utilisée pour le test de sous-typage — qu'il s'agisse de hachage parfait ou d'accès direct — peut s'appliquer aussi à l'invocation de méthodes : il suffit que chaque entrée de la table soit constituée de l'identifiant

de l'interface et de l'adresse du groupe de méthodes introduites par l'interface dans la table de méthodes elle-même (il n'y a pas besoin de table explicite pour les interfaces). Le hachage parfait donnerait ainsi la séquence :

```
load [object + #tableOffset], table
load [table + #hashingOffset], h
and #id_C, h, h
sub table, h, table                                    4L + B + 2
load table, table
load [table + #methodOffset], method
call method
```

Les instructions 2-5 en italiques s'ajoutent, avec $2L + 2$ cycles, au code nécessaire pour l'envoi de message en héritage simple, lequel nécessite $2L + B$ cycles. B est la latence de branchement non prédit, soit environ 10 cycles.

4. Simulation spatiale

Les différentes approches présentées ici ont été simulées sur la plate-forme et les *benchmarks* qui ont servi aux statistiques de (Ducournau, 2002a) et qui mesurent exclusivement les paramètres spaciaux des diverses implémentations. Les algorithmes de calcul du plus petit H_C sont assez immédiats et les temps de calcul insignifiants.

La table 2 présente d'abord le nombre total de classes (N) et des statistiques (moyenne et maximum par classe) sur le nombre de super-classes (n_C, c'est-à-dire la référence en matière de taille linéaire), ainsi que sur la taille des tables de sous-typage pour la coloration bidirectionnelle (COL_2) et le hachage parfait (PH) avec les deux fonctions `and` et `mod`. La table 3 présente ensuite les statistiques sur le nombre de méthodes par classe, sur la taille des tables de méthodes calculées suivant la coloration bidirectionnelle et suivant l'implémentation "standard" de C++ (SMI), et termine par le rapport entre la taille des tables de méthodes et la taille des tables de sous-typage. Il ressort globalement de ces tableaux que le hachage parfait, s'il n'atteint pas la qualité des résultats de la coloration, donne néanmoins de très bons résultats spaciaux avec `mod`. Les tables de sous-typage ne représentent en effet qu'une petite fraction des tables de méthodes. Les conclusions sont plus mitigées avec `and`, bien que les tables de sous-typage restent toujours plus petites que les tables de méthodes.

Dans les tableaux précédents, la distinction entre classes et interfaces n'est pas faite pour les hiérarchies JAVA. Les résultats suivants sont restreints à ces hiérarchies, en faisant cette distinction sur la base d'une heuristique car elle n'était pas faite dans les benchmarks d'origine. La table 4 présente les statistiques sur les hiérarchies JAVA— nombre de classes et d'interfaces, nombre d'interfaces implémentées par classe et étendues par interface — ainsi que la taille moyenne des tables en accès direct (DA). La table 5 présente ensuite les statistiques du hachage parfait des interfaces, avec le contenu des parties positives et négatives des tables ainsi que le rapport entre les tailles des parties négatives et positives des tables. Ce rapport négatif/positif est important dans la mesure où il montre que, dans presque tous les cas, la

Tableau 2. *Statistiques sur les hiérarchies de classes et le test de sous-typage*

	n_C		COL_2		PH (`mod`)		PH (`and`)		N
MI-jdk1.3.1	4.4	24	4.6	24	8.2	61	51.5	4100	7401
MI-Orbix	2.8	13	2.8	13	4.4	32	12.3	2050	2716
MI-Orbacus	4.5	19	4.7	19	8.1	50	24.0	1026	1379
IBM-XML	4.4	14	4.5	14	7.6	43	21.2	148	145
IBM-SF	9.2	30	9.7	30	22.9	128	95.9	8224	8793
MI-HotJava	5.1	23	5.5	23	9.8	85	32.8	519	736
MI-Corba	3.9	18	3.9	18	6.5	48	19.5	1026	1699
JDK.1.0.2	4.6	14	4.7	14	8.4	39	34.2	580	604
Cecil	6.5	23	6.6	23	12.7	78	53.9	608	932
dylan	5.5	13	5.5	13	9.0	26	25.5	516	925
harlequin	6.7	31	7.6	31	13.6	106	39.7	580	666
Self	30.9	41	31.2	41	66.4	140	181.7	1088	1802
vortex3	7.2	30	7.5	30	15.5	109	79.4	1232	1954
SmartEiffel	8.6	14	8.6	14	16.3	37	40.0	272	397
Unidraw	4.0	10	4.0	10	6.8	24	17.0	514	614
Lov-obj-ed	8.5	24	11.4	24	19.4	90	53.2	508	436
Geode	14.0	50	17.4	50	41.8	267	160.8	1026	1318

Tableau 3. *Statistiques sur les nombres de méthodes, la taille des tables de méthodes et le rapport entre tables de méthodes et tables de sous-typage*

	méthodes				table de méthodes		rapport		
	introduites		héritées		COL_2	SMI	COL_2	`mod`	`and`
MI-jdk1.3.1	1.3	149	19.2	243	243	1391	8.7	8.9	1.4
MI-Orbix	0.4	64	8.3	109	109	534	6.0	5.3	1.9
MI-Orbacus	1.2	74	18.0	137	137	761	7.8	8.4	2.8
IBM-XML	2.5	29	16.1	57	57	284	7.2	6.6	2.4
IBM-SF	2.8	257	44.9	346	397	2063	12.9	10.1	2.4
MI-HotJava	1.8	80	34.2	189	189	817	13.2	13.7	4.1
MI-Corba	0.4	43	8.0	67	81	427	5.1	4.1	1.4
JDK.1.0.2	5.3	75	37.0	158	158	691	15.9	15.1	3.7
Cecil	2.9	61	78.7	156	156	2058	24.2	34.9	8.2
dylan	0.9	64	77.1	139	139	1073	28.0	37.2	13.2
harlequin	0.6	62	34.8	129	129	977	9.3	16.1	5.5
Self	14.6	233	577.4	969	969	10098	37.6	55.8	20.4
vortex3	0.5	148	156.5	204	204	4994	41.8	72.0	14.1
SmartEiffel	12.2	222	135.3	324	324	1576	31.3	45.5	18.6
Unidraw	2.9	103	24.1	124	124	318	12.0	10.1	4.1
Lov-obj-ed	8.3	117	85.9	289	289	1590	19.8	21.8	7.9
Geode	6.1	193	231.8	880	892	10717	33.5	34.6	9.0

Tableau 4. *Classes et interfaces dans les benchmarks* JAVA *et taille des tables en accès direct*

	classes	interfaces	implémentées		étendues		DA
MI-jdk1.3.1	7056	345	1.2	21	3.2	8	39.9
MI-Orbix	2676	40	0.3	8	2.4	8	1.8
MI-Orbacus	1297	82	1.7	14	2.9	7	10.6
IBM-XML	129	16	1.7	9	2.9	6	3.6
IBM-SF	7920	873	6.2	27	2.8	9	283.2
MI-HotJava	681	55	2.4	19	2.9	7	10.6
MI-Corba	1634	65	1.1	14	2.7	6	6.0
JDK.1.0.2	576	28	1.1	9	3.6	8	6.4

Tableau 5. *Statistiques du hachage parfait des interfaces* JAVA

	partie positive			négative	PH		nég/pos		
	class	méth.	tot.	inter.	`mod`	`and`	`mod`	`and`	DA
MI-jdk1.3.1	3.2	19.2	20.8	1.2	1.9	4.0	0.2	0.4	1.9
MI-Orbix	2.4	8.2	9.4	0.3	0.4	1.4	0.1	0.3	0.2
MI-Orbacus	2.9	17.9	19.3	1.7	2.7	4.1	0.3	0.4	0.5
IBM-XML	2.9	16.4	17.8	1.7	2.3	3.4	0.3	0.4	0.2
IBM-SF	2.8	44.0	45.4	6.2	13.5	40.4	0.6	1.8	6.2
MI-HotJava	2.9	34.7	36.2	2.4	3.9	5.9	0.2	0.3	0.3
MI-Corba	2.7	7.7	9.1	1.1	1.6	3.0	0.4	0.7	0.7
JDK.1.0.2	3.6	37.5	39.3	1.1	1.4	2.1	0.1	0.1	0.2
Total	2.9	26.5	28.0	3.0	6.0	16.7	0.4	1.2	4.2

partie négative ne représente qu'une petite fraction de la partie positive, c'est-à-dire que l'application des techniques considérées ne changerait pas l'ordre de grandeur de la taille totale des tables. Les trois techniques considérées peuvent être améliorées sur le plan spatial, pour le prix d'une instruction et d'un cycle supplémentaires, en remplaçant les adresses par un indice (petit entier) ce qui divise le ratio par 2. Malgré cela, l'accès direct a sans doute une taille rédhibitoire, au moins pour un ou deux gros benchmarks (JDK 1.3 et IBM SF). PH `and` est acceptable à l'exception peut-être de IBM SF. Quant à PH `mod`, il est parfait pour l'espace mais un peu moins pour le temps.

Tableau 6. *Comparaison finale pour* JAVA

	appel de méthode		test sous-typage		espace
technique	cycles	code	cycles	code	nég/pos
COL_2	$2L + B$ = 16	3	$2L + 2$ = 8	4	–
DA	$3L + B$ = 19	4	$2L + 3$ = 9	7	4.2
PH `and`	$4L + B + 2$ = 24	7	$3L + 4$ = 13	7	1.2
PH `mod`	$4L + B + 7$ = 29	7	$3L + 9$ = 18	7	0.4

Finalement, la table 6 ordonne les techniques considérées par nombre de cycles des 2 opérations, en supposant $L = 3$, $B = 10$ et une latence de 6 pour la division entière (ce qui est peut-être optimiste).

5. Comparaison avec d'autres travaux

Le test de sous-typage a donné lieu à de nombreuses contributions. Deux tendances se partagent le domaine. La première, dans une optique mathématique, cherche le codage optimal d'un ordre (ou d'un treillis). (Fall, 1998) en fait une revue. Cependant, une taille minimale n'est pas forcément souhaitable en pratique car le code pour extraire les données devient trop complexe. De plus ces approches mathématiques sont essentiellement globales et non incrémentales, et comme la complexité de leur calcul est élevée, il n'est pas question de la recalculer à chaque chargement de classe. A l'opposé, figurent différentes approches, plus pragmatiques, comme celle de Cohen. Parmi les techniques les plus récentes, (Zibin *et al.*, 2001) propose une combinaison de coloration de classes et de la double numérotation proposée par (Schubert *et al.*, 1983) : appelée *PQ-encoding*, il s'agit vraisemblablement de la technique offrant le meilleur compromis espace-temps, mais elle n'est pas plus incrémentale.

Par suite de différentes lacunes dans les spécifications du langage, le test de sous-typage est omniprésent en JAVA, ce qui explique sans doute la nombreuse littérature qui lui est consacré. Le test de Cohen est couramment utilisé pour les classes (Alpern *et al.*, 2001b; Click *et al.*, 2002), mais il est en général mal optimisé, par exemple avec des tables de taille fixe, ou encore avec une indirection superflue. En ce qui concerne les interfaces, (Alpern *et al.*, 2001b) se sert de tables à accès direct, quasi-booléenne[3], donc inutilisable pour l'invocation de méthodes. D'autres implémentations utilisent des tables à accès direct (Krall *et al.*, 1997; Gagnon *et al.*, 2001) : dans le dernier cas, ce sont les méthodes qui sont accédées directement, et le gaspillage d'espace est tel que les trous de la table servent à l'allocation d'objets !

(Click *et al.*, 2002) proposent de leur côté une technique apparemment efficace, basée sur la vieille association d'une recherche linéaire naïve et d'un cache. Bien entendu, si le cache améliore les performances, il pourrait le faire pour n'importe quelle autre technique moins naïve. D'un point de vue spatial, les auteurs rapportent assez informellement une occupation de 8 à 11 mots par classe, ce qui est supérieur à celui de PH `mod`, à l'exception d'IBM SF qui semble cependant plus grand que les benchmarks utilisés par les auteurs.

(Alpern *et al.*, 2001a) fait une revue sur l'invocation de méthode sur un receveur typé par une interface. Les auteurs citent des recherches linéaires, des tables de hachage de taille fixe, avec comme clé les identifiants de méthodes. Dans le cas de l'invocation de méthode, il est possible de compiler les chaïnages séparés dans des arbres de décision, ce qui transforme, dans une certaine mesure, ce hachage quelconque en

3. Pour des raisons obscures (cf. note 2), une troisième valeur de vérité est nécessaire pour tenir compte des classes ou interfaces non encore chargées.

hachage parfait. L'article présente des benchmarks avec 2 valeurs pour le paramètre H, 5 et 40 : la première valeur représente la moyenne de H_C pour PH `mod`, alors que la seconde est supérieure à la moyenne de l'accès direct, IBM SF mis à part, qui serait donc meilleur du point de vue du temps comme de l'espace.

Toutes les implémentations des objets ne sont pas basées sur des tables de méthodes. Le meilleur contre-exemple est le compilateur SMART EIFFEL, où tous les mécanismes objet sont compilés dans des arbres de décision basés sur l'identifiant de type des objets (Zendra *et al.*, 1997; Collin *et al.*, 1997). Si la technique semble efficace, elle n'est pas en temps constant, et elle nécessite une compilation globale et une analyse de types.

Enfin, en ce qui concerne le hachage parfait, on peut s'étonner, et l'auteur s'étonne, qu'une technique aussi vieille n'ait pas encore été utilisée dans le contexte des langages à objets. C'est du moins ce qu'il semble. Si certains auteurs citent la technique (Driesen, 1993; Zibin *et al.*, 2003), c'est en général pour l'exclure car inadaptée à leur problème, trop coûteuse à calculer, ou pas assez compacte. De fait, nos propres expérimentations sur le hachage parfait des méthodes — et non des interfaces — sont très négatives (Ducournau, 2005). (Klefstad *et al.*, 2002) constitue une exception : le hachage parfait y est utilisé pour hacher le nom des méthodes dans le cadre de CORBA et d'applications distribuées.

6. Conclusion et perspectives

Dans cet article, nous avons proposé d'implémenter le test de sous-typage à l'aide du hachage parfait, en généralisant ainsi le test de (Cohen, 1991) et la coloration de classes (Vitek *et al.*, 1997). Le test résultant satisfait les cinq critères initiaux : i) *temps constant*, moins de 2 fois la coloration dans le cas du `and` ; ii) *espace linéaire*, environ 4 fois la coloration dans le cas de `mod` ; compatibilité avec iii) l'héritage multiple et iv) le chargement dynamique ; v) une séquence de code suffisamment courte pour être expansée en ligne. Une analyse plus détaillée est disponible dans (Ducournau, 2005). Ces résultats ne sont clairement pas parfaits. La compatibilité avec le chargement dynamique est acquise au prix d'un réel surcoût, aussi bien temporel que spatial, par rapport à la coloration. On aurait préféré plus petit, plus court et plus rapide, c'est-à-dire l'efficacité temporelle de `and` alliée à la compacité de `mod`.

Néanmoins, malgré l'imperfection de ces résultats, le hachage parfait offre une alternative réelle aux implémentations actuelles qui sont souvent *ad hoc*. L'application à C++ serait assez immédiate, avec la numérotation des classes à l'édition de liens et une légère adaptation pour l'héritage non `virtual`. Le cas de JAVA est encore plus simple et plus intéressant, puisque le hachage parfait s'applique aussi à `invokeinterface`. On peut même envisager d'utiliser le hachage parfait comme implémentation de base des objets, dans n'importe quel langage, en procédant pour les attributs comme on l'a fait pour les méthodes en JAVA. Cependant, le résultat serait probablement d'une efficacité médiocre. La question qui reste en suspens et qui consti-

tue la principale perspective de ce travail — au-delà bien sûr de sa mise en pratique dans les implémentations existantes — est de savoir s'il pourrait exister une fonction de hachage offrant un meilleur compromis espace-temps que `and` et `mod`, plus compacte que la première et plus rapide que la seconde.

7. Bibliographie

Alpern B., Cocchi A., Fink S., Grove D., « Efficient Implementation of Java Interfaces : Invokeinterface Considered Harmless », *in* OOP (2001), 2001a.

Alpern B., Cocchi A., Grove D., « Dynamic Type Checking in Jalapeño », *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001b.

Click C., Rose J., « Fast Subtype Checking in the Hotspot JVM », *Proc. ACM-ISCOPE conference on Java Grande (JGI'02)*, 2002.

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

Collin S., Colnet D., Zendra O., « Type inference for Late Binding. The SmallEiffel Compiler », *Joint Modular Languages Conference*, LNCS 1204, Springer, p. 67-81, 1997.

Czech Z. J., Havas G., Majewski B. S., « Perfect hashing », *Theor. Comput. Sci.*, vol. 182, n° 1-2, p. 1-143, 1997.

Dixon R., McKee T., Schweitzer P., Vaughan M., « A fast method dispatcher for compiled languages with multiple inheritance », *Proc. OOPSLA'89*, ACM Press, 1989.

Driesen K., « Selector table indexing and sparse arrays », *Proc. OOPSLA'93*, ACM Press, 1993.

Driesen K., *Efficient Polymorphic Calls*, Kluwer Academic Publisher, 2001.

Ducournau R., Implementing Statically Typed Object-Oriented Programming Languages, Technical Report n° 02-174, L.I.R.M.M., Université Montpellier 2, 2002a. (submitted).

Ducournau R., « "Real World" as an Argument for Covariant Specialization in Programming and Modeling », *in* J.-M. Bruel, Z. Bellahsène (eds), *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, LNCS 2426, Springer, p. 3-12, 2002b.

Ducournau R., Perfect hashing as an almost perfect subtype test, Technical Report n° 05-058, L.I.R.M.M., Université Montpellier 2, 2005. (submitted).

Ducournau R., Coloring, a Versatile Technique for Implementing Object-Oriented Languages, Technical Report n° 06-001, L.I.R.M.M., Université Montpellier 2, 2006. (submitted).

Ellis M., Stroustrup B., *The Annotated C++ Reference Manual*, Addison-Wesley, Reading (MA), USA, 1990.

Fall A., « The foundations of taxonomic encoding », *Computational Intelligence*, vol. 14, p. 598-642, 1998.

Gagnon E. M., Hendren L., « SableVM : A Research Framework for the Efficient Execution of Java Bytecode », *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

Kiczales G., des Rivières J., Bobrow D., *The Art of the Meta-Object Protocol*, MIT Press, 1991.

Klefstad R., Krishna A., Schmidt D., « Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications », *Proc. of the 4th International Symposium on Distributed Objects and Applications*, OMG, 2002.

Knuth D. E., *The art of computer programming, Sorting and Searching*, vol. 3, Addison-Wesley, 1973.

Krall A., Grafl R., « CACAO - A 64 bits JavaVM Just-in-Time Compiler », *Concurrency : Practice ans Experience*, vol. 9, n° 11, p. 1017-1030, 1997.

Lippman S., *Inside the C++ Object Model*, Addison-Wesley, New York (NY), USA, 1996.

Liskov B., Curtis D., Day M., Ghemawat S., Gruber R., Johnson P., Myers A. C., THETA Reference Manual, Technical report, MIT, 1995.

Mehlhorn K., Tsakalidis A., « Data structures », *in* Van Leeuwen (1990), chapter 6, p. 301-341, 1990.

Meyer B., *Object-Oriented Software Construction*, The Object-Oriented Series, second edn, Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

OOP, *Proceedings of the Twelfth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*, SIGPLAN Notices, 32(10), ACM Press, 1997.

OOP, *Proceedings of the Sixteenth ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'01*, SIGPLAN Notices, 36(10), ACM Press, 2001.

Pfister B. H. C., Templ J., Oberon technical notes, Technical Report n° 156, Eidgenossische Techniscle Hochschule Zurich–Departement Informatik, 1991.

Privat J., Ducournau R., « Link-time Static Analysis for Efficient Separate Compilation of Object-Oriented Languages », *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, 2005.

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

Queinnec C., « Fast and Compact Dispatching for Dynamic Object-Oriented languages », *Information Processing Letters*, 1997.

Schubert L., Papalaskaris M., Taugher J., « Determining type, part, color and time relationship », *Computer*, vol. 16, p. 53-60, 1983.

Sprugnoli R., « Perfect hashing functions : a single probe retrieving method for static sets », *Communications of the ACM*, vol. 20, n° 11, p. 841-850, November, 1977.

Van Leeuwen J. (ed.), *Algorithms and Complexity*, vol. 1 of *Handbook of Theoretical Computer Science*, Elsevier, Amsterdam, 1990.

Vitek J., Horspool R., Krall A., « Efficient Type Inclusion Tests », *in* OOP (1997), p. 142-157, 1997.

Vitter J. S., Flajolet P., « Average-case analysis of algorithms and data structures », *in* Van Leeuwen (1990), chapter 9, p. 431-524, 1990.

Zendra O., Colnet D., Collin S., « Efficient Dynamic Dispatch without Virtual Function Tables : The SmallEiffel Compiler », *in* OOP (1997), p. 125-141, 1997.

Zibin Y., Gil J., « Efficient Subtyping Tests with PQ-Encoding », *in* OOP (2001), p. 96-107, 2001.

Zibin Y., Gil J., « Incremental Algorithms for Dispatching in Dynamically Typed Languages », *Proc. of ACM Conf. on Principles of Programming Languages (POPL'03)*, p. 126-138, 2003.

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LES ACTES :
   LMO 2006

2. AUTEURS :
   Roland Ducournau

3. TITRE DE L'ARTICLE :
   Le hachage parfait fait-il un parfait test de sous-typage ?

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Hachage parfait et test de sous-typage

5. DATE DE CETTE VERSION :
   20 janvier 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :
   L.I.R.M.M. – CNRS et Université Montpellier 2
   161, rue Ada – 34392 Montpellier cedex 5
   Roland.Ducournau@lirmm.fr
   – téléphone : +33(0)4 67 41 86 01
   – télécopie : +33(0)4 67 41 85 00
   – e-mail : Roland.Ducournau@lirmm.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.22 du 04/10/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com