UNIVERSITÉ DU QUÉBEC À MONTRÉAL

IMPLÉMENTATION HOMOGÈNE DE LA GÉNÉRICITÉ COVARIANTE DANS NIT, UN LANGAGE À OBJETS EN HÉRITAGE MULTIPLE

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR
ALEXANDRE TERRASA

FÉVRIER 2014

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.01-2006). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

REMERCIEMENTS

Je tiens à remercier tout particulièrement mon directeur de recherche, Jean Privat, professeur au département d'informatique de l'UQAM, pour toute son aide et sa patience. Tout au long de ma maîtrise, il m'a permis d'entrevoir le vaste monde que représente la compilation. Je le remercie d'avoir su me partager son goût pour la beauté des langages à objets et du beau code. Merci à lui pour la réalisation du langage NIT avec lequel j'ai eu le plaisir de m'amuser et de me battre depuis maintenant trois ans.

Je remercie également mes collègues et amis développeurs du langage Nit, qui ont eu l'amabilité de me partager leur savoir et leur talent ; Alexis Laferrière et Lucas Bajolet, ainsi qu'Étienne Gagnon, directeur de la maîtrise et du doctorat en informatique et professeur au département d'informatique de l'UQAM. Merci aussi à tous les professeurs du département dont j'ai eu la chance de suivre les cours.

Je ne peux oublier mes parents qui ont su me supporter, malgré l'éloignement, durant toute ma maîtrise à l'UQAM. Ainsi que mes amis ici au Québec ; Anthony, Benoît, Léo, Mathieu et ailleurs dans le monde ; Camille, Nathalie, Paul et tous les autres que je ne peux citer ici. Pour leur soutien et leur gentillesse durant les périodes de doutes, merci.

Je me dois aussi de remercier Anthony, Mathieu et Lucas pour leur relecture profonde de ce document. Ils ont su trouver de nombreuses fautes, coquilles et tournures de phrases incompréhensibles.

Enfin, merci à Eveline pour son soutien, sa présence et son amour durant les épisodes les plus difficiles et sombres de cette maîtrise. Merci à elle de partager ma vie.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

La généricité est une fonctionnalité importante des langages à objets, spécialement pour l'écriture de bibliothèques et de composants réutilisables. La généricité existe cependant sous plusieurs formes. D'abord par la politique de typage appliquée par le langage : invariante, covariante ou effacée. Mais aussi par l'implémentation qui en est faite : hétérogène, homogène ou effacée.

Dans cette étude, nous traitons le sujet de l'implémentation homogène de la généricité dans le langage de programmation NIT. D'après sa spécification, le langage NIT applique une politique de typage covariante. Or, les implémentations existantes sont soit effacées, soit dans un schéma de compilation globale avec représentation hétérogène, soit naïves et inefficaces.

La politique covariante autorise une plus grande expressivité dans l'utilisation des types génériques. En contrepartie, elle nécessite une mécanisme de résolution des types génériques ouverts et elle entraîne un problème d'insécurité dans les appels de méthodes utilisant des types génériques redéfinis de manière covariante.

L'implémentation homogène permet de partager la même implémentation du corps des méthodes entre les variations génériques d'une même classe. Le coût en mémoire de la généricité est alors réduit au strict minimum. En revanche, cette approche pose les problèmes de représentation des types génériques, d'implémentation du test de sous-typage et d'implémentation du mécanisme de résolution.

Nous présentons une solution basée sur l'utilisation de descripteurs de types et de tables de résolution et nous l'implémentons dans un compilateur pour le langage NIT. Nous comparons ses performances avec des moteurs d'exécution pour plusieurs langages grâce à des programmes artificiels et non-réalistes. Les résultats montrent que notre proposition est parmi les plus performantes. Nous utilisons ensuite de vrais programmes NIT pour comparer les performances de notre solution face à d'autres compilateurs NIT suivant d'autres politiques. Les résultats montrent que le sur-coût engendré par une politique covariante est négligeable.

Finalement, nous considérons qu'il est possible de fournir une implémentation homogène efficace de la généricité covariante en héritage multiple. Le sur-coût engendré par la covariance est négligeable comparé aux gains en termes de simplicité et d'expressivité du code.

Mots clés : généricité covariante, implémentation homogène, langages à objets.

INTRODUCTION

Introduite par ML en 1984 (MacQueen, 1984) puis popularisée par Ada en 1987 (Musser et Stepanov, 1987), la généricité est un style de programmation dans lequel les algorithmes sont écrits indépendamment des types de données sur lesquels ils s'appuient. La généricité est un moyen de factoriser du code tout en conservant le typage statique et la sécurité qu'il apporte. Les bibliothèques de collections de nombreux langages sont basées sur ce modèle.

La généricité est un concept fermement rattaché à la programmation par objets et aux langages à typage statique. Certains langages, comme Java, autorisent la définition de méthodes génériques. Cette pratique ne sera pas étudiée plus en détails ici.

Une *classe générique* est une classe qui peut être paramétrée par un ou plusieurs *paramètres de types générique* lors de sa déclaration. C'est le cas par exemple de la classe `List<E>` où `E` est un paramètre de type. Les paramètres de types peuvent être utilisés au sein de la classe en abstraction des types réels qui seront utilisés à l'exécution du code. Les classes génériques peuvent utiliser plusieurs paramètres de types dans la même déclaration. C'est une pratique très courante dans l'implémentation des dictionnaires (*map*), par exemple `Map<T, U>`.

Lors de l'instanciation de la classe générique, les paramètres de types sont remplacés par des *arguments de types* précisés par le développeur dans l'initialisation de l'objet. Par exemple `new List<Integer>` ou `new List<E>`. Les types génériques `List<E>` et `List<Integer>` sont appelés des *variations génériques* de la classe générique `List`.

Les types génériques contenant des références vers des paramètres de types comme `List<E>`, `Map<Integer, List<E>>` ou tout simplement `E` sont appelés des *types ouverts*.

À l'inverse, les types génériques ne contenant aucune référence vers un type ouvert comme `List<Integer>` sont appelés des *types résolus*.

À l'écriture de la classe générique, les paramètres de types peuvent être bornés (Cardelli et Wegner, 1985). En précisant une borne, le développeur s'assure que les arguments de types fournis à l'instanciation de la classe sont conformes à cette borne. Il est ainsi possible de créer des classes pouvant opérer de manière générique sur un sous-ensemble de types. Notons que dans la majorité des cas, ne pas fixer de borne à un type formel revient à borner ce dernier par *Object* ou tout autre racine de la hiérarchie d'héritage du langage considéré. Les bornes sont des informations de type statique. Elles permettent au compilateur de vérifier la cohérence des types utilisés lors de l'écriture du programme par le développeur.

Pour un langage de programmation à objets, la généricité est une fonctionnalité importante, spécialement pour l'écriture de bibliothèques et de composants réutilisables. De nombreux langages permettent de l'utiliser. On la retrouve sous le nom de *généricité* en ADA, EIFFEL, JAVA et C# ou *polymorphisme paramétrique* en ML, SCALA et HASKELL.

Contexte de l'étude : le langage Nit

Notre étude se place dans le contexte du développement du langage de programmation NIT[1]. NIT est une évolution du langage PRM, un langage dédié à l'expérimentation de différentes techniques d'implémentation et de compilation (Ducournau, Morandat et Privat, 2009; Privat et Ducournau, 2005). Il s'agit d'un langage à objets à typage statique et héritage multiple. Ses fonctionnalités incluent types nullables (Gélinas, Gagnon et Privat, 2009), types virtuels (Torgersen, 1998) et généricité.

La spécification de NIT précise que la politique de typage dynamique appliquée par le langage est covariante. Cependant, le compilateur principal `nitc` implémente la généricité selon une politique effacée non sûre, c'est à dire sans vérifications à l'exécution pour assurer la cohérence des types. Ce compilateur comporte encore aujourd'hui un grand

1. `http://nitlanguage.org`

nombre de *TODO* et *FIXME* dans son code source en rapport avec l'implémentation de la généricité.

Il existe deux autres moteurs utilisés dans un but d'enseignement. Le premier est l'interpréteur `nit`, un interpréteur naïf et peu performant. Le second est `nitg`, un compilateur global proposant de grosses optimisations globales et de la particularisation (*customization*). Ces deux moteurs implémentent la politique de typage covariante comme indiqué par la spécification mais seulement sur un sous-ensemble du langage.

Dans cette étude, nous souhaitons fournir un compilateur qui implémente efficacement la généricité en respectant la spécification du langage NIT et sa politique covariante.

Problèmes étudiés

La politique de typage appliquée aux types génériques peut varier selon les langages. Cette politique est choisie par les concepteurs du langage qui imposent aux développeurs des compilateurs et moteurs d'exécution de s'y conformer. L'implémentation de la généricité, quant à elle, est généralement à la charge de ces développeurs. Ainsi il existe aujourd'hui une multitude d'implémentations compatibles avec telle ou telle politique.

Comparaison des politiques de typage générique Le choix de la politique de typage générique définit les règles d'utilisation des types génériques. Elle peut entraîner des contraintes pour conserver un typage sûr. La politique invariante est sûre et ne nécessite pas de tests de sous-typage supplémentaires à l'exécution mais manque d'expressivité. La politique effacée supprime toute information générique à la compilation et ne permet pas de représenter la généricité à l'exécution mais implique des vérifications supplémentaires à l'exécution. La politique covariante qui nous concerne impose des validations supplémentaires pour assurer un typage sûr. Ces validations peuvent être établies statiquement, directement via la syntaxe du langage pour forcer la vérification par le compilateur, ou dynamiquement en vérifiant les types lors de l'exécution. De plus, les politiques non-effacées introduisent la notion de types ouverts. Ces types nécessitent de connaître le

receveur courant pour être traduits en types concrets et être utilisés dans les tests de sous-typage. Nous appelons ce problème de traduction la *résolution des types ouverts*. Mais quels sont les impacts de la politique covariante par rapport aux deux autres ? La politique covariante ralentit-elle les performances du programme à l'exécution à cause de vérifications supplémentaires ou de la relation de sous-typage plus complexe ?

Comparer objectivement des politiques de typage n'est pas une tâche facile puisque chacune apporte ses propres règles d'utilisation des types génériques. En effet, changer la politique de typage d'un langage implique de gros effets de bord car cela change le sémantique du langage. La sémantique du test de type lui même peut varier selon la politique et bouleverser le fonctionnement d'un programme conforme à une autre politique. Par exemple, des conversions de type (*casts*) implicites peuvent être retirées ou ajoutées à différents endroits pour assurer un typage sûr et ainsi briser le code déjà existant.

Dès lors, comparer des politiques de typage implique de pouvoir résoudre deux contraintes majeures. Trouver des programmes réels qui se comportent exactement de la même façon avec les différentes politiques. C'est la tâche la plus difficile. Trouver des compilateurs qui peuvent générer le même code machine excepté pour les parties relatives à la politique elle-même et à l'implémentation du test de type. Cette tâche est plus facile car nous pouvons implémenter ces compilateurs nous-mêmes.

Implémentation du test de type générique Les implémentations du mécanisme de sous-typage basées sur des tables en mémoire sont les plus populaires (Vitek, Horspool et Krall, 1997). En plus d'être compatibles avec l'héritage multiple, celles-ci sont constantes en temps et nécessitent un faible espace mémoire. Trois approches utilisant les tables existent : les matrices binaires, la coloration et le hachage parfait. Chacune de ces approches apporte ses avantages et ses inconvénients selon les schémas de compilation envisagés. Mais peut-on réutiliser ces mécanismes pour les types génériques dans le cadre d'une compilation séparée ? Avec une édition de liens globale ? Avec le chargement dynamique ?

Implémentation de la généricité L'implémentation des mécanismes de la généricité varie elle aussi entre les différents langages et moteurs d'exécution. Dans le cadre d'une implémentation effacée, les types génériques n'existent plus après la compilation. Dans l'implémentation hétérogène chaque variation générique est considérée comme une classe distincte et possède son propre code compilé. Avec une implémentation homogène, les variations génériques d'une classe partagent le même code compilé. Ainsi ces trois approches impliquent des mécanismes de représentations des types génériques et du test de sous typage différents. Mais parmi ces trois approches, lesquelles sont compatibles avec une politique de typage générique covariante ? Et laquelle permet de l'implémenter le plus efficacement possible ?

Finalement, toutes ces problématiques peuvent être réunies en une question : « Peut-on implémenter efficacement la généricité covariante dans un langage à héritage multiple ? »

Contributions

Nos contributions au domaine sont les suivantes :

Comparaison des politiques de typage générique Nous comparons les politiques invariante, covariante et effacée. Pour chacune d'elles nous présentons ses impacts sur la sémantique du test de sous-typage, les règles qu'elle définit concernant l'usage des types génériques, la sécurité du typage à l'exécution ainsi que la nature et le nombre de tests de sous-typage requis pour garantir cette sécurité.

Ingénierie inverse des moteurs EIFFELSTUDIO et MONO C# Dans notre présentation des implémentations homogènes de la généricité, nous listons le compilateur EIFFELSTUDIO pour le langage EIFFEL ainsi que la machine virtuelle MONO pour le langage C#. Ces deux moteurs sont peu ou pas documentés sur leurs mécanismes de généricité. Nous proposons une description de leur implémentation grâce à la rétro-ingénierie du code source des compilateurs, machines virtuelles et de programmes générés.

Implémentation homogène de la généricité covariante en héritage multiple Nous proposons une solution pour une implémentation efficace de la politique covariante avec héritage multiple. Nous présentons deux nouvelles structures permettant d'implémenter la généricité de manière homogène. Les descripteurs de types permettent de représenter les types génériques en un minimum d'espace mémoire. Les tables de résolution permettent de fournir un mécanisme de résolution et un test de sous-typage en temps constant. Notre solution est compatible avec l'héritage multiple et les schémas de compilation : global, séparé avec édition de liens globale et séparé avec chargement dynamique.

Implémentation d'un compilateur pour le langage NIT Nous implémentons notre solution dans un compilateur séparé avec édition de liens globale pour le langage NIT. Ce compilateur, `nitg-s`, est compatible avec les types nullables et les types virtuels tout en respectant la politique covariante du langage.

Étude pratique des implémentations existantes Nous étudions les performances des implémentations de la généricité de différents compilateurs et machines virtuelles. Nous utilisons des hiérarchies de types générées pour comparer des moteurs d'exécution pour les langages C++, JAVA, SCALA, EIFFEL, C# et NIT. Nous montrons que les implémentations du mécanisme de sous-typage basées sur des tables sont constantes en temps et figurent parmi les plus performantes.

Comparaison pratique des politiques génériques Nous dérivons notre implémentation de `nitg-s` pour produire un compilateur utilisant une implémentation effacée nommé `nitg-e`. Grâce à ces deux compilateurs supportant de vrais programmes écrits en NIT, nous pouvons fournir une comparaison pratique de la politique covariante et effacée. De plus, nous comparons ces résultats avec les performances de deux autres dérivations de notre compilateur simulant une politique sûre, c'est à dire sans vérifications supplémentaires à l'exécution. Nous montrons que le coût de la politique covariante est faible sinon négligeable comparé à ses avantages.

Comparaison pratique des mécanismes d'implémentation du test de type générique Toujours grâce à notre compilateur `nitg-s`, nous comparons les performances des matrices binaires, de la coloration et du hachage parfait appliqués aux mécanismes de résolution des types génériques et de sous-typage. Nous prouvons ainsi que notre solution est compatible avec ces trois approches, la coloration étant plus adaptée à un schéma de compilation séparé avec édition de liens globale, le hachage parfait étant plus adapté au chargement dynamique.

Structure du document

Ce mémoire est structuré comme suit. Le chapitre 1 présente et compare théoriquement les impacts des politiques invariante, covariante et effacée. Le chapitre 2 explique le principe de l'implémentation des mécanismes objets par tables et présente les approches par matrices binaires, coloration et hachage parfait. Le chapitre 3 décrit les implémentations de la généricité effacée, hétérogène et homogène existantes. Le chapitre 4 présente notre solution et son implémentation dans le langage NIT. Enfin, nous étudions les performances de notre implémentation par rapport à d'autres moteurs dans le chapitre 5. Le dernier chapitre présente nos conclusions et perspectives d'avenir.

CHAPITRE I

POLITIQUES DE TYPAGE GÉNÉRIQUE

Dans les langages de programmation à objets, la politique de typage générique spécifie les règles concernant l'utilisation des types génériques. Dans ce chapitre, nous expliquons tout d'abord les impacts de la politique choisie par un langage sur la sémantique du test de sous-typage. Dans certains cas la politique de typage dynamique, appliquée à l'exécution, peut différer de la politique de typage statique imposée par le compilateur. Nous montrons ensuite que cette politique a un effet sur la sécurité des types et qu'elle peut nécessiter des vérifications supplémentaires à l'exécution.

Lors de l'exécution, nous pouvons regrouper les comportements des langages en trois familles de politiques à savoir invariante, covariante et effacée. Chacune de ces politiques impose une sémantique particulière au test de sous-typage.

Avec la **généricité invariante**, il n'existe pas de relation de sous-typage entre les variations génériques d'une classe :

$$\forall A, B, G\langle A\rangle, G\langle B\rangle : G\langle A\rangle <: G\langle B\rangle \Leftrightarrow A = B$$

Avec la **généricité covariante**, une relation de sous-typage entre deux variations génériques est possible en fonction de la relation des arguments de types :

$$\forall A, B, G\langle A\rangle, G\langle B\rangle : G\langle A\rangle <: G\langle B\rangle \Leftrightarrow A <: B$$

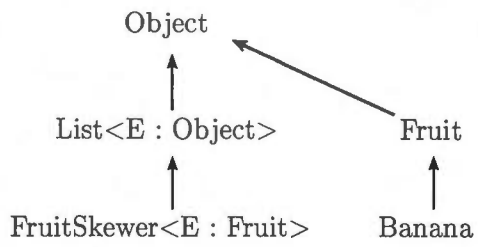

Figure 1.1 Exemple de hiérarchie de classes. Le paramètre de type `E` de la classe générique `List` est borné par `Object`. Elle accepte donc n'importe quel type comme argument de type générique. Dans sa sous-classe, `FruitSkewer`, `E` est borné par `Fruit`. L'argument de type générique utilisé ne peut être que du type `Fruit` ou un sous-type.

Avec la **généricité effacée**, les arguments de types génériques ne sont aucunement considérés et toutes les variations génériques d'une classe sont sous-types les unes des autres :

$$\forall G\langle A\rangle, G\langle B\rangle : G\langle A\rangle <: G\langle B\rangle$$

La figure 1.1 présente un exemple de hiérarchie de classes. Pour chacune des politiques invariante, covariante et effacée, les hiérarchies de types résultantes sont données en figure 1.2. La figure 1.3 donne quelques exemples de résultats de tests de sous-typage en fonction de ces trois politiques.

La politique de typage dynamique utilisée par le langage affecte aussi le nombre de tests de sous-typage qui seront réalisés lors de l'exécution. Pour assurer la sûreté du système de type, chaque politique implique des vérifications additionnelles lors de l'exécution qui ne sont pas demandées explicitement par le développeur.

La figure 1.4 donne un exemple de programme statiquement correct dans les trois politiques mais dont le comportement à l'exécution doit être vérifié en fonction de la politique choisie.

Lors de l'exécution, tous les types manipulés sont résolus. Ainsi, un type dynamique est toujours résolu et les tests de sous-typage manipulent des types résolus. La question

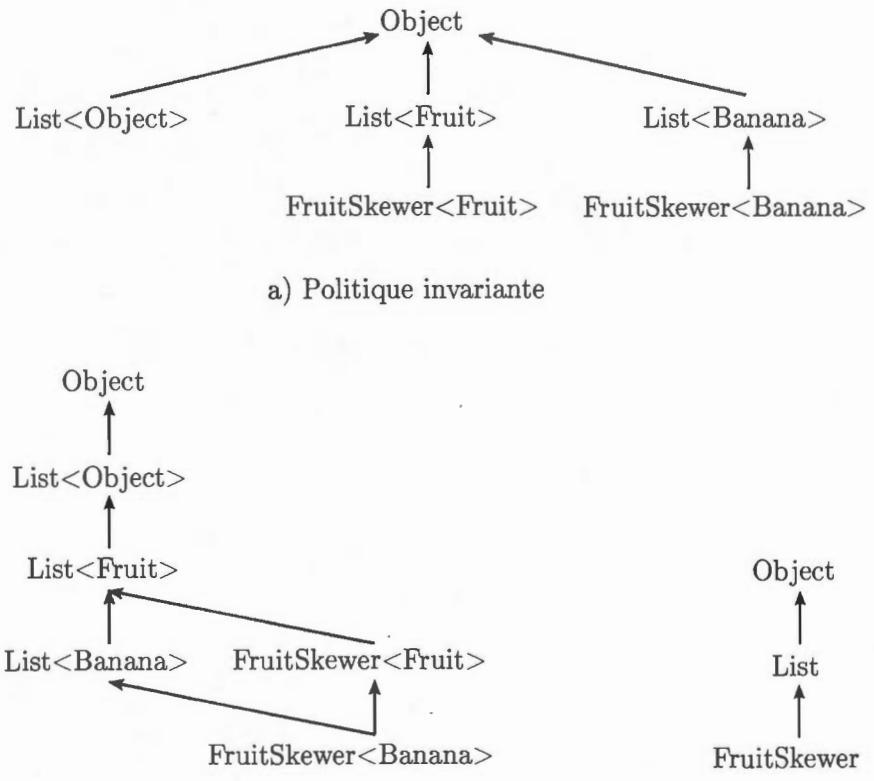

Figure 1.2 Hiérarchie des types génériques selon les politiques de typage dynamique a) invariante, b) covariante et c) effacée.

Test de sous-typage	Invariance	Covariance	Effacement
`List<Fruit> <: Object`	√	√	√
`FruitSkewer<Fruit> <: List<Fruit>`	√	√	√
`List<Banana> <: List<Fruit>`		√	√
`List<Integer> <: List<Fruit>`			√
`List<Fruit> <: FruitSkewer<Fruit>`			

Figure 1.3 Résultats de tests de sous-typage selon la politique de typage générique appliquée. Le choix d'une politique de typage dynamique affecte la sémantique du test de sous-typage. Le résultat d'un même test peut varier en fonction de la politique appliquée.

est alors : « Comment la politique de typage gère-t-elle les références vers des types ouverts comme `new List<E>` ou `instanceof E` ? » Ces types ouverts dépendent du type dynamique du receveur courant (appelé `this`, `self` ou encore `Current` en fonction des langages) comme dans le cas l'envoi de message (liaison tardive). C'est ce que nous appelons le mécanisme de *résolution*.

Dans la plupart des langages, la seule construction généralement interdite utilisant des types ouverts est `new E`, où `E` est un paramètre de type. Puisque le type dynamique de `E` dépend du type dynamique du receveur, il ne peut être statiquement garanti qu'un constructeur existe effectivement pour ce type car les constructeurs se sont pas hérités.

Le langage C# propose une contrainte statique particulière pour autoriser l'instanciation de paramètres de types. Les paramètres de types peuvent être bornés grâce à une borne `where E: new()`. En spécifiant cette borne, le développeur s'assure que l'argument de type utilisé fournit obligatoirement un constructeur public et sans paramètre ce qui permet d'écrire `new E()`.

1.1 Généricité invariante

La politique de typage générique invariante garantit que le typage statique est suffisant pour garantir la sécurité des types à l'exécution. Ainsi, aucun test de sous-typage sup-

```
1  class List<E> {
2          public void push(E e) { ... }
3          public E pop() { ... }
4  }
5
6  List<Fruit> lf;
7  List<Banana> lb = new List<Banana>;
8  Object o = lb;
9
10 lf = (List<Fruit>)o; // La politique invariante lève une exception ici:
11                      // Attendu `List<Fruit>`, obtenu `List<Banana>`
12 lf.push(new Coconut()); // La politique covariante lève une exception
13                         // à l'entrée de la méthode `push()`:
14                         // Attendu `Banana`, obtenu `Coconut`
15 Banana b = lb.pop(); // La politique effacée lève une exception ici:
16                      // Attendu `Banana`, obtenu `Coconut`
```

Figure 1.4 Comportement à l'exécution d'une programme affecté par la politique générique du langage. Le comportement réel du programme à l'exécution est déterminé par la sémantique du test de sous-typage et les contraintes additionnelles nécessaires au typage sûr qui seront ajoutées par le compilateur. À moins d'indication contraire, nous utilisons une syntaxe proche de celle de Java.

```
1  class FruitSkewer<E extends Fruit> {
2      public void addFruits(List<E> fruits) { ... }
3  }
4
5  List<Banana> bananas = new List<Banana>();
6  FruitSkewer<Fruit> skewer = new FruitSkewer<Fruit>();
7  skewer.addFruits(bananas); // Erreur de compilation en politique
       invariante:
8                             // Attendu `List<Fruit>`, obtenu `List<
                                  Banana>`
```

Figure 1.5 Exemple du manque d'expressivité de la généricité invariante. La méthode `addFruits` n'accepte pas de paramètre de type dynamique `List<Banana>` bien que `Banana` soit un sous-type de `Fruit`.

plémentaire n'est requis lors de l'exécution. C'est la politique utilisée pour les *templates* en C++ et la politique appliquée aux classes génériques en C#.

En revanche, la politique invariante souffre d'un manque d'expressivité comme le montre la figure 1.5. Pour reproduire le même comportement, un programme écrit dans un langage à politique invariante doit utiliser des solutions de contournement. La documentation MSDN de C# donne quelques exemples de ce genre de patrons [1].

Plusieurs concepteurs de langages de programmation considèrent que la spécification invariante manque trop d'expressivité pour être utilisée efficacement et que les solutions de contournement utilisées sont peu élégantes (Ducournau, 2002; Liskov et Wing, 1994). Ce constat a mené à la création de la spécification covariante.

1.2 Généricité covariante

Le concept de généricité covariante découle naturellement du raisonnement suivant : « Si une banane est un fruit alors, une liste de bananes doit être une liste de fruits. » Cette

1. `http://msdn.microsoft.com/en-US/library/ms228359(v=vs.90).aspx`

logique a mené à la création de la politique covariante utilisée par EIFFEL et NIT. C'est aussi la politique appliquée aux interfaces covariantes dans C# et celle des tableaux primitifs dans JAVA.

La politique covariante est une forme de généricité variante mais elle n'est pas la seule. Le langage C# par exemple, autorise une généricité contravariante sur les interfaces tel que :

$$\forall A, B, G\langle A\rangle, G\langle B\rangle; G\langle A\rangle <: G\langle B\rangle \Leftrightarrow B <: A$$

Cette forme de variance est considérée par certains comme contre-intuitive et peu applicable au monde réel (Ducournau, 2002; Day et al., 1995). La bivariance a aussi été envisagée (Altidor, Reichenbach et Smaragdakis, 2012; Igarashi et Viroli, 2006). Le choix de l'une ou l'autre de ces formes de variances revient au spécificateur du langage. Dans notre cas, NIT a choisi une politique covariante. Nous nous concentrons donc sur la covariance, les solutions d'implémentation présentées dans ce mémoire pouvant s'appliquer à d'autres formes de variance.

Puisque `List<Banana>` est un sous-type de `List<Fruit>`, l'exemple refusé par une politique invariante donné en figure 1.5 est tout à fait correct en suivant une politique covariante. En revanche, certaines vérifications additionnelles sont nécessaires lors de l'exécution pour assurer la cohérence des types.

1.2.1 Sûreté des types

Au sein d'une classe générique, les méthodes utilisant des paramètres de types dans leur signature peuvent être non-sûres lorsqu'utilisées avec la covariance. Comme montré en figure 1.4, un appel à une méthode définie dans une classe générique peut s'avérer non-sûr statiquement. Le type des arguments passés à une méthode dépend du type dynamique du receveur. C'est le genre de polymorphisme introduit par la généricité covariante.

Selon la théorie du typage sûr (Cardelli et Wegner, 1985), pour que les appels de méthodes soient statiquement sûrs, les types des paramètres de la méthode ne devraient

```
1  List<Fruit> l = (List<Fruit>)new List<Banana>(); // Statiquement refusé
       : Cannot convert type `List<Banana>` to `List<Fruit>`
```

Figure 1.6 Exemple d'invariance des classes génériques en C#. Avec la politique invariante appliquée par défaut, la conversion du type `List<Banana>` vers `List<Fruit>` est interdite.

évoluer que de manière contravariante. Les types de retour des méthodes peuvent quand à eux évoluer de manière covariante.

Pour assurer la sécurité du typage, les langages à objets peuvent avoir recours à deux approches différentes. Le première consiste à introduire des règles statiques particulières permettant de vérifier la cohérence des types lors de la compilation. C'est ce que nous appelons une *approche statique*. La seconde délègue ces vérifications à l'exécution et relève d'une *approche dynamique*.

1.2.2 Approche statique

Le but de cette approche est de s'assurer statiquement qu'aucune utilisation de la covariance ne peut mener à une insécurité lors de l'exécution. C'est l'approche utilisée par C#.

Par défaut, toute utilisation variante d'un type générique est interdite comme illustré en figure 1.6 pour le langage C#.

L'activation de la variance sur les types génériques se fait grâce aux interfaces, en préfixant les paramètres de types à l'aide des modificateurs génériques `out` et `in`.

L'utilisation du modificateur `out` indique au compilateur que le paramètre de type va être utilisée de manière covariante. Il ne peut donc être utilisé qu'en tant que type de retour de méthode afin que le code écrit soit statiquement sûr. Toute utilisation en position de paramètre de méthode est statiquement rejetée par le compilateur. La politique de

```
1  interface IListOut<out E> {
2      // void push(E e); // Statiquement refusé: The covariant type
           parameter `E` must be contravariantly valid on `IStackOut<E>.
           push(E)`
3      E pop();
4  }
5
6  class List1<E>: IListOut<E> {
7      public void push(E e) { ... }
8      public E pop() { ... }
9  }
10
11 IListOut<Fruit> lo = (IListOut<Fruit>)new List1<Banana>(); // ok
12 // lo.push(new Banana()); // Statiquement refusé: Type `IListOut<Fruit
       >' does not contain a definition for `push'
13 Fruit f = lo.pop(); // ok
```

Figure 1.7 Utilisation du modificateur out en C#. Grâce à ce modificateur, le compilateur s'assure que les seules utilisations variantes des paramètres de types se font en position contravariante. L'assignation peut alors se faire de manière covariante.

```
1  interface IListIn<in E> {
2      void push(E e);
3      // E pop(); // Statiquement refusé: The contravariant type
           parameter `E` must be covariantly valid on `IStackIn<E>.pop()`
4  }
5
6  class List2<E>: IListIn<E> {
7      public void push(E e) { ... }
8      public E pop() { ... }
9  }
10
11 IListIn<Fruit> li = (IListIn<Fruit>)new List2<Banana>(); // Erreur
       dynamique: Cannot cast from source type to destination type.
12 IListIn<Banana> lb = (IListIn<Banana>)new List2<Fruit>(); // ok
```

Figure 1.8 Utilisation du modificateur *in* en C#. Grâce à ce modificateur, le compilateur s'assure que les seules utilisations variantes des paramètres de types se font en position covariante. L'assignation peut alors se faire seulement de manière contravariante.

typage dynamique utilisée pour cette interface est alors covariante comme le montre la figure 1.7.

L'utilisation du modificateur `in` indique quant à lui l'utilisation du paramètre de type en position contravariante. L'utilisation du paramètre de type `E` est alors autorisée en tant que paramètre de méthode mais interdite en tant que type de retour. La politique de typage dynamique utilisée pour cette interface est alors contravariante comme le montre la figure 1.8.

Réunir les comportements `in` et `out` dans une même interface est possible mais impose l'invariance comme le montre la figure 1.9.

Le cas des attributs typés par des paramètres de types est un peu particulier. Puisqu'accessibles à la fois en lecture et en écriture, les paramètres de types sont utilisés à la fois

```
1  interface IList<E>: IListOut<E>, IListIn<E> {}
2
3  class List3<E>: IList<E> {
4      public void push(E e) { ... }
5      public E pop(E e) { ... }
6  }
7
8  IList<Fruit> l = (IList<Fruit>)new List3<Banana>(); // Erreur dynamique
       : Cannot cast from source type to destination type.
```

Figure 1.9 Exemple d'interface invariante en C#. Statiquement, l'assignation covariante est autorisée mais mène à une erreur d'exécution car `List<Banana>` ne peut être converti en `IList<Fruit>` de par l'invariance.

en position de paramètre (*setter* d'attribut) et en type de retour (*getter* d'attribut). Afin d'assurer un typage statique sûr, ils doivent donc être invariants. Pour cette raison, C# n'autorise la variance que sur les interfaces, puisqu'elles sont dépourvues d'attributs.

L'activation de la variance sur une interface a un impact sur le résultat des tests de sous-typage dans lesquels cette interface est impliquée. Le modificateur `out` autorise une relation de sous-typage covariante entre les interfaces. Le modificateur `in` autorise quant à lui une relation de sous-typage contravariante. Réunir ces deux modificateurs au sein d'une même interface revient à créer une relation invariante.

Bien que statiquement sûre, l'approche proposée par C# est verbeuse et, finalement, très restrictive. De plus, l'activation des relations de sous-typage covariantes ou contravariantes en fonction de l'utilisation des mots clés `in` et `out` s'avère peu intuitive pour le développeur. Malgré son manque d'expressivité et le besoin d'implémenter un grand nombre d'interfaces, elle permet d'éliminer la nécessité de tests additionnels à l'exécution pour assurer la cohérence des types.

1.2.3 Approche dynamique

Une autre approche consiste à déléguer les vérifications de covariance à l'exécution. Les constructions qui peuvent être traitées statiquement sont vérifiées à la compilation. Les autres cas sont vérifiés à l'exécution au moment où le type dynamique de l'instance est connu. C'est la politique spécifiée par EIFFEL et NIT.

Pour assurer un typage sûr, le compilateur ajoute un *cast* à l'entrée des méthodes utilisées en covariance. Nous appelons ce genre de conversions des *casts de covariance*. Le type réel utilisé pour le *cast* est déterminé en fonction du type dynamique du receveur.

Malgré les tests additionnels requis à l'exécution, cette approche se veut plus intuitive et en adéquation avec les attentes du développeur dans un monde réel tel qu'expliqué par Ducournau (Ducournau, 2002) et illustré par la figure 1.5.

1.3 Généricité effacée

Avec la généricité effacée : 1- tous les paramètres de types utilisés dans une classe sont remplacés par leurs bornes statiques lors de la compilation ; 2- tous les arguments de types sont supprimés. Ainsi, l'information générique n'est plus disponible à l'exécution. Ce processus est appelé *effacement* (*erasure*) en JAVA et SCALA.

Puisque les arguments de types n'ont pas d'importance à l'exécution, `new List<Fruit>`, `new List<GtkButton>` et `new List<E>` (`E` étant un paramètre de type) ont exactement le même comportement et retournent tous une instance du type dynamique `List`.

Le langage JAVA a choisi une spécification effacée pour maintenir une rétro-compatibilité avec les machines virtuelles existantes. Ainsi, la création d'instances génériques sans arguments de type comme `new List` est autorisée. Dans ce cas particulier, le type statique utilisé par le compilateur est basé sur les bornes des paramètres de types.

À cause de l'effacement, toutes les instances de la classe `List<E>` sont du même type dynamique `List`. Un *cast* vers un type générique `(List<Banana>)expr` est en fait considéré comme un *cast* vers le type de la classe `List` et vérifie que `expr` est une liste ou

```
1  List lf;
2  List lb = new List; // Instancie une nouvelle liste de `Object`
3  Object o = lb;
4  lf = (List)o;
5  lf.push(new Coconut());
6  Banana b = (Banana)lb.pop(); // Le type attendu est `Banana`,
7                               // `pop` retourne statiquement un `Object`
8                               // un cast est donc nécessaire
9                               // et ce cast va échouer.
10 b.peel();
```

Figure 1.10 Simulation de la généricité sans classe générique.

une de ses sous-classes. Préciser les arguments de type est encouragé pour une meilleure analyse statique des types.

Les tests de sous-typage fonctionnent sur le même comportement, la partie générique du type n'étant pas considérée non plus. JAVA autorise seulement la syntaxe utilisant les jokers (*wildcards*) `List<?>` ou pas de partie générique du tout.

Avec l'effacement, toute utilisation d'un paramètre de type est en fait remplacé par sa borne lors de la compilation. Ainsi, `(E)expr` est en fait considéré comme `(Bound)expr` à l'exécution. `expr instanceof E` pourrait être considéré comme `expr instanceof Bound` mais est interdit en JAVA.

L'effacement peut en fait être vu comme un moyen de faire de la généricité sans classes génériques. Le compilateur ajoute automatiquement des *casts* pour changer le types des arguments des méthodes et de leurs valeurs de retour en accord avec les bornes statiques spécifiées.

La figure 1.10 modifie notre exemple de base (figure 1.4) pour retirer tout usage de la généricité et les remplacer par des *casts* explicites.

Avec l'effacement, ces *casts* sont ajoutés pour assurer la cohérence du type de retour des méthodes. Nous les appelons *casts d'effacement* et ils ne sont nécessaires que dans le cadre d'une politique effacée.

```
1  /* Liste de fruits spécialisée */
2  class FruitList extends List {
3      @Override
4      /* Attention: peut être utilisée seulement avec des fruits */
5      void push(Object o) { // La signature originale prend un Object
6          Fruit fruit = (Fruit) o; // un cast est ajouté pour s'assurer
               que `o` est un `Fruit`
7          ...
```

Figure 1.11 Exemple de spécialisation d'une classe non générique avec covariance sur le paramètre de la méthode push. Un *cast* explicite est ajouté au début de la méthode pour s'assurer d'un type correct.

```
1  class FruitList<F extends Fruit> extends List<F> { ... }
```

Figure 1.12 Évolution de la borne statique d'un paramètre de type en politique effacée.

Ces mêmes *casts* explicites peuvent être utilisés pour assurer la cohérence des types à l'entrée d'une méthode redéfinie de manière covariante comme le montre la figure 1.11.

Comme avec la politique covariante, des *casts de covariance* sont nécessaires pour assurer la sûreté du typage dans les redéfinitions covariantes des bornes des paramètres de types. En revanche, la plupart de ces *casts* peuvent être évités. En réalité, les *casts* de covariance dans une politique effacée sont nécessaires uniquement dans les méthodes surchargées si la borne statique change comme le montre la figure 1.12.

C'est aussi vrai dans le cas où le paramètre de type est fixé dans la relation d'héritage comme montré dans la figure 1.13.

Grâce à l'effacement, les redéfinitions covariantes utilisant des types ouverts comme `List<E>` n'ont pas besoins de *casts* puisque toutes les versions de `List` auront en fait le même type dynamique `List` même si la borne change.

Alors que l'effacement permet d'éviter quelques *casts de covariance*, les *casts d'effacement* représentent un sur-coût nécessaire seulement avec une politique covariante.

```
1  class FruitList extends List<Fruit> { ... }
```

Figure 1.13 Fixation de la borne statique d'un paramètre de type dans une relation d'héritage.

La politique effacée, plus simple à implémenter dans un compilateur, pose un problème d'expressivité à l'instar de la politique invariante. Diverses approches ont été expérimentées pour contourner ce problème.

Le langage JAVA fournit une solution permettant de gérer les appels covariants sur des types génériques grâce aux jokers. Les jokers peuvent, entre autres, être utilisés dans n'importe quelle déclaration de méthode pour spécifier que le paramètre de type attendu par la méthode est inconnu et que le compilateur doit utiliser `Object` à la place. Les jokers peuvent aussi être bornés pour remplacer `Object` par un type plus précis. Les jokers permettent ainsi de vérifier statiquement les appels de méthodes covariants. Les *casts d'effacement* restent nécessaires pour vérifier les types de retour des méthodes.

Le langage SCALA offre un compromis entre la politique effacée de JAVA et les modificateurs génériques utilisés par C# pour assurer un typage variant statiquement sûr. À l'aide des modificateurs de type génériques `+E` et `-E` correspondant respectivement à `out E` et `in E` C#, il est possible d'activer la covariance tout en évitant le sur-coût des *casts* de covariance à l'exécution. En revanche, les *casts d'effacement* sont toujours nécessaires pour protéger les retours de méthodes.

1.4 Conclusion

Ce chapitre discute des différentes politiques de typage dynamique appliquées à la généricité indépendamment de l'implémentation sous-jacente. Ces politiques peuvent être regroupées en trois grandes familles : invariantes, covariantes et effacées.

L'expressivité fournie par la généricité dépend de la politique choisie. La politique invariante est la plus restrictive, la généricité covariante apporte une expressivité demandée

par plusieurs concepteurs de langages et programmeurs. La politique effacée présente des limites d'expressivité qui, à cause de l'effacement ne sont pas présentes dans une politique covariante.

La politique choisie par le langage change les relations de sous-typage entre types génériques et impacte ainsi la sémantique du test de sous-typage. Un test de sous-typage entre deux types ne donne pas le même résultat en fonction de la politique appliquée.

La politique choisie impose aussi des vérifications supplémentaires lors de l'exécution pour assurer la cohérence des types. Alors que la politique invariante est suffisante pour assurer un typage statique sûr, les politiques covariantes (par approche dynamique) et effacée nécessitent des *casts de covariance* pour offrir un typage sûr. La généricité effacée permet d'éviter certains *casts de covariance* mais ajoute la nécessité de *casts d'effacement.*

Finalement, l'implémentation de la généricité et les performances de cette implémentation dépendent essentiellement de l'implémentation faite du test de sous-typage. Le chapitre suivant présente donc les différentes implémentations connues du test de sous-typage.

CHAPITRE II

IMPLÉMENTATIONS DU TEST DE SOUS-TYPAGE

Le chapitre précédent montre que la généricité peut engendrer des *casts de covariance* et des *casts d'effacement* pour assurer un typage sûr. Ces *casts* représentent un surcoût induit par la politique sur le test de sous-typage. Les performances à l'exécution des programmes utilisant la généricité dépendent donc de l'implémentation du test de sous-typage. La relation de sous-typage, quant à elle, diffère en fonction de la politique appliquée.

Dans ce chapitre, nous discutons des principales implémentations des mécanismes objets en nous concentrant sur l'implémentation du test de sous-typage. Le cas particulier de la généricité n'est pas considéré dans ce chapitre, il est étudié au chapitre 3 de ce mémoire. Ainsi, dans ce chapitre, il est considéré que les relations de sous-typage s'appuient uniquement sur les relations de spécialisation entre classes et donc qu'un type équivaut à une classe.

Le test de sous-typage en héritage simple a déjà été longuement discuté et n'a pas vu d'avancée significative depuis la numérotation de Cohen (Cohen, 1991). De plus, la plupart des langages objets modernes doivent gérer un sous-typage multiple. Même certains langages, comme JAVA ou C# qui spécifient un héritage de classe simple, autorisent un héritage multiple d'interfaces. De ce fait, nous dédions notre étude aux approches du test de sous-typage applicables à la fois à l'héritage simple et à l'héritage multiple.

L'usage de tables en mémoire pour représenter les mécanismes objets est la stratégie d'implémentation communément utilisée dans les langages. Les représentations par tables sont généralement utilisées en liaison avec des descripteurs de classes et d'instances (Proebsting et Townsend, 1997).

Un *descripteur de classe* contient les informations partagées par toutes les instances de cette classe : identifiant unique de la classe, valeurs des attributs de classes, pointeurs de méthodes (table de fonctions virtuelles) et table de sous-typage. Les descripteurs sont représentés en mémoire sous la forme de tableaux ou de structures, un descripteur pour chaque classe.

Un *descripteur d'instance* contient les informations spécifiques à une instance. Un descripteur d'instance contient au moins les valeurs des attributs d'instance et un pointeur vers le descripteur de sa classe. D'autres informations spécifiques à une instance peuvent aussi y être stockées comme la valeur de hachage unique de l'objet.

C++ (Ellis et Stroustrup, 1990) utilise des tables plus complexes nommées sous-objets basées sur des ajustements de pointeurs en fonction du type statique des références. Tous les mécanismes objets ne sont pas basés sur des tables. SmartEiffel (Zendra, Colnet et Collin, 1997), par exemple, propose une implémentation basée sur des arbres (*Binary Tree Dispatch* ou *BTD*). Les *BTD*, grâce à leur organisation en arbres binaires de recherche, peuvent profiter d'une meilleure prédiction de branchement.

Les sections suivantes traitent des implémentations basées sur des tables et des algorithmes de sous-typage correspondants, sans considération de la généricité. Nous présentons d'abord l'approche par numérotation de Cohen utilisée en héritage simple. Ensuite, l'approche par matrice binaire (*Binary Matrix*). Cette approche, de par sa simplicité, est populaire dans l'implémentation des langages en héritage multiple. Enfin, nous présentons ensuite deux méthodes permettant de compresser ce type de matrice : la coloration et le hachage parfait.

2.1 Numérotation de Cohen

La numérotation de Cohen (Cohen, 1991) est considérée comme la première implémentation performante du test de sous-typage. Particulièrement performante pour les langages à héritage simple, cette approche n'est toutefois pas adaptée à l'héritage multiple.

À chaque classe C est associé un indice correspondant à sa profondeur dans la hiérarchie d'héritage $C.pos$. Chaque classe possède aussi une table de sous-typage $C.st$ contenant toutes ses super-classes à leurs indices respectifs.

Le test de sous-typage `x instanceof C` consiste alors à accéder à la table de sous-typage st puis à vérifier si la table contient l'identifiant de $C.id$ à la position $C.pos$ (Morandat, 2010; Ducournau, 2011b).

$$\texttt{x instanceof C} \Leftrightarrow C.pos < x.class.st.length \;\&\&\; x.class.st[C.pos] == C.id$$

Le test de sous-typage peut être réalisé en temps constant $O(1)$. La vérification sur la taille de la table peut-être évitée en utilisant des tables de taille uniforme au prix d'un espace mémoire supplémentaire.

Cette approche est utilisée pour implémenter le test de sous-typage dans les machines virtuelles HotSpot et Jalapeno pour Java et dans mono pour C#.

2.2 Matrice binaire

Pour représenter la relation entre classes, l'approche la plus simple consiste à associer chaque type avec tous les autres types du systèmes au sein d'une matrice. Chaque ligne de la matrice représente une table de sous-typage pour une classe et est stockée dans son descripteur de classe.

À chaque classe C est assignée une position invariante unique notée $C.pos$. Soit la relation de sous-typage entre deux classes A et B telle que $A <: B$. Le bit à la position $B.pos$ dans la table de sous-typage $A.st$ est positionné à 1. Une absence de relation est représentée par un bit positionné à 0.

Le test de sous-typage `x instanceof C` consiste alors à accéder à la table de sous-typage *st* depuis le descripteur d'instance de x au travers de son descripteur de classe puis à vérifier si la table *st* contient un bit positionné à 1 à la position $C.pos$.

$$\texttt{x instanceof C} \Leftrightarrow x.class.st[C.pos] == 1$$

Le coût théorique du test de sous-typage correspond à une indirection depuis le descripteur d'instance de x pour récupérer son descripteur de classe, à une seconde indirection pour obtenir la table de sous-typage dans le descripteur de classe et enfin à un accès à la table de sous-typage. Les matrices binaires permettent ainsi un test de sous-typage en temps constant $O(1)$.

L'espace mémoire nécessaire au stockage de la matrice est quadratique sur le nombre de classe, $O(n^2)$ bits. Gagnon (Gagnon et Hendren, 2001) montre que ces matrices binaires sont généralement clairsemées dans le cas de l'héritage multiple d'interfaces en JAVA.

Une optimisation simple consiste à retirer la partie finale de la table ne contenant que des 0. Avec ce type d'optimisation les tables peuvent avoir des tailles variables et un test supplémentaire sur la taille de la table doit être ajouté avant le test de sous-typage.

$$\texttt{x instanceof C} \Leftrightarrow C.pos < x.class.st.length \;\&\&\; x.class.st[C.pos] == 1$$

De par leur simplicité, les matrices binaires sont fréquemment utilisées pour gérer l'héritage multiple. Elles sont compatibles aussi bien avec un mode de compilation global que séparé et avec le chargement dynamique. La matrice binaire est utilisée pour la résolution des tests de types dans le compilateur EIFFELSTUDIO. Elles sont aussi appliquées au cas de l'héritage multiple des interfaces JAVA par Alpern (Alpern, Cocchi et Grove, 2001), Krall (Krall et Grafl, 1997) et Gagnon (Gagnon et Hendren, 2001).

2.3 Coloration

En théorie des graphes, la coloration consiste à assigner une couleur (un numéro) à chaque nœud d'un graphe tel que deux nœuds directement connectés par une arête ont des couleurs différentes. Vitek (Vitek, Horspool et Krall, 1997) et Ducournau (Ducournau, 2011a) appliquent la coloration aux hiérarchies de classes afin de minimiser le nombre de colonnes utilisées dans les tables de sous-typage. L'idée générale est de faire partager la même colonne, représentée par une couleur, par plusieurs classes incomparables.

À chaque classe C est assignée un identifiant $C.id$ et une position invariante $C.color$ tel que deux classes ayant des sous-classes communes ne peuvent partager la même couleur. Ceci peut être calculé simplement en créant un graphe de conflit sur la hiérarchie tel que décrit par Ducournau (Ducournau, 2011a).

Puisque plusieurs classes peuvent partager la même couleur, et donc la même colonne dans une table de sous-typage, un seul bit n'est pas suffisant pour différencier deux classes. Ainsi, si on considère une relation de sous-typage entre deux classes A et B tel que $A <: B$, la position $B.color$ dans $A.st$ contient un entier $B.id$ représentant l'identifiant unique de B.

Le test de sous-typage `x instanceof C` consiste alors à accéder à la table de sous-typage st depuis le descripteur d'instance de `x` au travers de son descripteur de classe puis à vérifier si la table st à la colonne $C.color$ contient l'identifiant de C, $C.id$.

$$\texttt{x instanceof C} \Leftrightarrow C.color < x.class.st.length \;\&\&\; x.class.st[C.color] == C.id$$

La coloration permet un test de sous-typage en temps constant tout comme les matrices binaires, $O(1)$. La complexité spatiale est quant à elle quadratique $O(n^2)$ mais seulement dans le pire cas (Ducournau, 2011a; Ducournau, Morandat et Privat, 2009).

De par la complexité du calcul des tables de sous-typage, la coloration est préférablement utilisée dans un monde fermé en compilation globale plutôt qu'avec du chargement

dynamique. Privat (Privat et Morandat, 2008) applique la coloration à la compilation séparée avec édition de lien globale dans le langage PRM. C'est aussi l'implémentation utilisée dans le compilateur NITC du langage NIT. Des versions incrémentales adaptées au chargement dynamique sont proposées pour SMALLTALK (Amiel, Gruber et Simon, 1994) et JAVA (Palacz et Vitek, 2003) mais restent relativement inefficaces pour l'héritage multiple (Ducournau, 2011a).

2.4 Hachage parfait

Le hachage parfait de classe tel que proposé par Ducournau (Ducournau et Morandat, 2011) est aussi un moyen de compresser les tables de sous-typage. L'idée est de créer une table de hachage sans collision, dite parfaite, qui représente la table de sous-typage.

Le hachage parfait est possible si l'on connaît au moment de la création de la table toutes les valeurs qu'elle doit contenir. Par contre, contrairement à la coloration, il n'est pas nécessaire de connaître l'ensemble des tables pour effectuer les calculs, ce qui rend cette technique plus adaptée au chargement dynamique.

À chaque classe C est associée un identifiant unique $C.id$ qui sera utilisé par la fonction de hachage. Puis pour chaque classe est calculé un paramètre de hachage $C.hash$ qui autorise une table de sous-typage sans collision pour toutes les super-classes de C. Soit une relation de sous-typage entre deux classes A et B tel que $A <: B$, le hachage de $B.id$ par $A.hash$ noté h donne la position invariante de B dans $A.st$ et contient l'entier $B.id$.

Pour réduire l'espace total requis par les tables, la meilleure solution est d'utiliser des tables de longueurs variables. Cela implique que le paramètre de hachage doit être calculé pour chaque classe et non globalement. Par ailleurs, utiliser un paramètre de hachage global empêcherait tout chargement dynamique efficace.

Le test de sous-typage `x instanceof C` consiste alors à accéder à la table de sous-typage st depuis le descripteur d'instance de `x` au travers de son descripteur de classe puis à

vérifier si la table st à la position h contient l'identifiant de C, $C.id$.

$$h = hash(x.class.hash, C.id)$$

$$\texttt{x instanceof C} \Leftrightarrow h < x.class.st.length \;\&\&\; x.class.st[h] == C.id$$

La complexité temporelle et spatiale de cette méthode est liée à la fonction de hachage utilisée. Plusieurs fonctions de hachage ont été considérées. Les expérimentations menées par Ducournau (Ducournau et Morandat, 2011) donne un avantage à l'opérateur ET bit à bit.

La complexité théorique reste constante $O(1)$ mais est en pratique plus lente que pour la matrice binaire ou la coloration à cause du calcul de la position par hachage. La complexité spatiale dépend elle aussi de la fonction de hachage mais reste quadratique $O(n^2)$ dans le pire des cas (Ducournau et Morandat, 2011; Ducournau, Morandat et Privat, 2009).

Le hachage parfait est moins adapté à un mode de compilation global car moins performant que la coloration mais il propose un bon compromis entre temps et espace dans le cadre du chargement dynamique.

2.5 Conclusion

Ce chapitre explique les techniques d'implémentation des mécanismes de sous-typage par tables. La plupart des mécanismes objets sont basés sur des structures stockées en mémoire nommées descripteurs. Les descripteurs d'instances permettent de représenter les instances et leurs attributs. Les descripteurs de classes permettent de représenter les classes et les méthodes qu'elles contiennent.

Dans les implémentations par tables, les relations de sous-typage correspondant à une classe sont représentées par une table nommée table de type. Cette table de type est stockée dans le descripteur de classe qui est accédée au moyen d'une indirection depuis le descripteur d'instance.

Il existe trois approches notables permettant d'implémenter les tables de type et le test de sous-typage dans un contexte d'héritage multiple. Toutes permettent d'obtenir un résultat en temps constant mais toutes ne sont pas compatibles avec les mêmes schémas de compilation.

La plus simple utilise des matrices binaires faisant correspondre chaque classe à toutes les autres classes du système. Son implémentation est compatible avec les modes de compilation global et séparé mais nécessite un espace mémoire conséquent.

L'implémentation par coloration permet de compresser efficacement les tables de sous-typage au prix d'un coût de calcul initial important. De ce fait, cette approche est destinée à un schéma de compilation global, le coût de recalcul des couleurs suite à un chargement dynamique étant un facteur discriminant.

Le hachage parfait permet lui aussi de compresser les tables de sous-typage et ce de manière compatible avec le chargement dynamique. Le temps de calcul est théoriquement identique aux deux premières approches mais en pratique plus lent car nécessitant une fonction de hachage. Le hachage parfait représente un compromis entre temps et espace pour les schémas de compilation séparés avec chargement dynamique.

La présentation des méthodes de sous-typage que nous avons faite dans ce chapitre ne prend aucunement en compte le cas de la généricité. La généricité introduit une catégorie supplémentaire de types que sont les variations génériques d'une même classe. Ainsi, l'information offerte par le descripteur de classe ne suffit plus à représenter les différents types qui peuvent exister dans un programme à l'exécution et les relations de sous-typage qui en découlent.

CHAPITRE III

IMPLÉMENTATIONS DE LA GÉNÉRICITÉ

La généricité nécessite des mécanismes additionnels à ceux présentés dans le chapitre précédent pour fonctionner. Comme montré au chapitre 1, le type dynamique du receveur doit être pris en compte pour déterminer le type exact des types génériques ouverts. Ce principe s'approche du mécanisme de la liaison tardive utilisé pour le polymorphisme dans l'envoi de message.

Ce chapitre explique les différentes implémentations connues de la généricité dans les langages à objets. Ces implémentations peuvent être regroupées en trois principales familles d'implémentations : l'implémentation effacée, l'implémentation hétérogène et l'implémentation homogène.

Les implémentations hétérogènes et homogènes peuvent être appliquées aux politiques invariantes et covariantes sans discernement, seul le contenu des tables de sous-typage varie. La différence principale entre ces deux implémentations est de savoir si les variations génériques d'une classe partagent ou non la même implémentation. Cette considération a un impact majeur sur la structure des descripteurs utilisés et l'implémentation du test de sous-typage.

L'implémentation effacée proposée par JAVA se comporte exactement comme une implémentation sans généricité. Son existence s'explique par un besoin de rétrocompatibilité avec les précédentes machines virtuelles. Cette implémentation n'est compatible qu'avec une politique effacée et c'est ce qui justifie l'existence d'une telle politique.

Les sections suivantes expliquent ces trois implémentations, leur fonctionnement puis leurs avantages et inconvénients.

3.1 Implémentation effacée

La généricité effacée trouve ses origines dans l'implémentation de la machine virtuelle JAVA (*JVM*). Le support de la généricité pour le langage JAVA a été introduit dans la version 5.0, avant cela, JAVA ne supportait pas la généricité. Afin de préserver une rétro-compatibilité avec les machines virtuelles écrites pour les versions précédentes, JAVA a opté pour une méthode permettant de représenter la généricité dans un code intermédiaire qui n'était pas fait pour la représenter.

À la compilation, toutes les informations génériques associées aux classes et aux instances sont supprimées. Les références vers des paramètres de types au sein d'une classe générique sont remplacées par leurs bornes statiques ou `Object` si aucune n'a été spécifiée par le développeur. De la même façon, tous les types ouverts utilisés sont remplacés par des types résolus basés sur les bornes. Les *casts* de covariance nécessaires sont remplacés par des *casts* vers les bornes. Ce processus est appelé l'effacement (*erasure*).

Puisque les classes génériques compilées ne contiennent plus d'informations basées sur le type dynamique du receveur, la même classe générique peut représenter toutes les instances de cette classe. Les descripteurs de classes et le code compilé des méthodes sont partagés par toutes les variations génériques d'une même classe.

Puisque les informations de type générique sont effacées lors de la compilation, elles sont donc inaccessibles à l'exécution. Les tests de sous-typage se comportent alors comme des tests de sous-typage non génériques, seul le type de la classe est pris en compte. Les tests de sous-typage par matrice binaire, coloration ou hachage parfait peuvent être appliqués aux implémentations effacées de la généricité. Ce genre d'implémentation fonctionne aussi très bien dans un contexte de compilation séparée avec chargement dynamique. C'est d'ailleurs l'implémentation utilisée par JAVA et SCALA (Dragos et Odersky, 2009). C'est aussi l'implémentation utilisée par le compilateur PRMC (Privat, 2006).

Plusieurs travaux se sont concentrés sur la possibilité de maintenir une information sur les types génériques dans une implémentation effacée sans modifier la JVM (Viroli et

Natali, 2000; Cartwright, Steele et L., 1998; Bracha et al., 1998; Odersky et Wadler, 1997; Agesen, Freund et Mitchell, 1997; Bank, Myers et Liskov, 1997). Dans toutes ces extensions, il s'agit de conserver les types génériques exacts dans des structures de données additionnelles pour pallier le processus d'effacement. Ces approches souffrent toutes d'inconvénients majeurs comparé au gain obtenu. Ainsi, la syntaxe utilisée pour activer le polymorphisme générique peut être complexe et peu intuitive comme dans GJ et NEXTGEN. Le processus de compilation est complexifié avec NEXTGEN et PIZZA. Ou peut engendrer une perte de performance notable dans le cas de POLYJ et PIZZA. Au final, toutes ces approches ne présentent que des solutions de contournement appliquées à une implémentation effacée et inutiles dans une implémentation de type hétérogène ou homogène (Kennedy et Syme, 2001).

3.2 Implémentation hétérogène

Avec une implémentation hétérogène, le compilateur génère une version adaptée du code pour chaque variation générique d'une classe. Soit la classe générique `List<E>`; si le compilateur détermine que cette liste est utilisée comme une `List<Fruit>` et une `List<Banana>`, il crée deux versions compilées de cette classe, une adaptée à `Fruit` et une autre adaptée à `Banana`. Dans chaque version adaptée d'une classe, les références vers des types ouverts sont remplacés par les types résolus spécifiques à la version compilée.

La création d'une instance de type `List<Fruit>` revient à créer une instance de la classe `List` adaptée pour `Fruit`. Chaque variation générique compilée possède alors son propre descripteur de classe et sa propre propre table de sous-typage. Les tests de sous-typage peuvent donc être résolus de la même façon qu'avec le test de sous-typage utilisé pour des classes tel que nous l'avons présenté au chapitre 2.

En plus d'augmenter la taille des exécutables et l'espace mémoire utilisé, cette approche peut poser un problème de récursivité infinie dans la génération des versions adaptées avec certaines constructions. Par exemple, si le code de la classe générique `List<E>` fait un `new List<List<E>>`, la version adaptée pour `List<Int>` à besoin de compiler

la version adaptée pour `List<List<Int>>`. Cette nouvelle version adaptée doit alors utiliser une `List<List<List<Int>>>` et ainsi de suite. C'est une limitation connue des compilateurs hétérogènes comme *g++* avec les *templates*.

Fixer un nombre limité d'imbrication des types génériques peut être une solution pour éviter ce problème d'adaptation infinie mais elle interdit du même coup toute utilisation récursive de classe générique. Cette solution est choisie dans la spécification du langage C++, le nombre limite d'imbrications étant laissé à l'implémentation du compilateur (ISO/IEC, 2007). C'est aussi la solution choisie par le compilateur SmartEiffel.

Le processus de compilation de C++ considère en fait les *templates* comme des macros. Pendant la compilation d'un fichier source, chaque nouvelle instanciation d'un *template* va déclencher la compilation d'une nouvelle version adaptée du *template*. Il est alors impossible de compiler séparément un *template* puisque le compilateur doit connaître les types avec lesquels il est instancié pour créer les adaptations correspondantes. C'est une autre limitation de l'implémentation hétérogène. Le code source du *template* doit être fourni au développeur des applications clientes afin qu'il puisse compiler ses propres versions adaptées du *template*.

En C++, puisque chaque adaptation est compilée séparément pour chaque unité de compilation, un *template* donné peut être compilé autant de fois qu'il existe d'unités de compilation l'utilisant. Le compilateur choisira une de ces versions au moment de l'édition de liens. Cela peut ralentir considérablement le temps de compilation des projets où de nombreuses adaptations sont nécessaires.

La dernière version de C++ (version 11) introduit le concept de *template* externe. En utilisant un *template* externe, le développeur indique au compilateur de ne pas générer d'adaptation pour ce *template*. C'est alors au développeur de fournir lui même la version adaptée du *template* lors de l'édition de liens. Cette technique complexe et peu intuitive ne change pas la sémantique ni l'implémentation des génériques. Elle permet uniquement au développeur de s'épargner le sur-coût en terme de temps de compilation et d'espace mémoire induit par l'implémentation hétérogène.

La généricité est implémentée de manière hétérogène dans les compilateurs SmartEiffel pour le langage Eiffel, le compilateur EiffelStudio en mode *finalize*, *g++* et *clang++* pour le langage C++ et enfin le compilateur global *nitg* pour le langage Nit.

3.3 Implémentation homogène

Avec une implémentation homogène, toutes les variations génériques d'une classe partagent le même code compilé des méthodes. Puisque le code des méthodes est partagé entre plusieurs variations génériques d'une même classe, l'information sur le type générique de l'instance ne peut plus être stockée directement dans le code de la méthode contrairement à une implémentation hétérogène. Partager l'implémentation des méthodes présente trois problèmes que l'approche homogène doit résoudre pour fonctionner.

Représentation des types génériques à l'exécution Contrairement aux implémentations effacées et hétérogènes, l'implémentation homogène pose un problème de représentation des types génériques à l'exécution.

Dans le cadre d'une implémentation homogène, toutes les variations génériques d'une classe dépendent de la même implémentation, celle compilée pour la classe générique, indépendamment de ses variations. Il faut donc trouver un moyen de différencier deux variations comme `List<Banana>` et `List<Object>` puis être capable, pour toute instance générique de la classe `List`, de retrouver quel argument de type générique à été utilisé lors de l'instanciation.

Mécanisme de résolution de type Comme il est présenté dans le chapitre 1, les spécifications invariantes et covariantes dépendent du type dynamique du receveur. Le type dynamique du receveur est utilisé pour trouver le type exact à l'exécution d'un type générique tel que `List<E>` où `E` est un paramètre de type. Nous appelons *mécanisme de résolution de type* le processus qui permet de transformer `List<E>` en `List<Argument>` où `Argument` est l'argument de type utilisé pour l'instanciation de la variation générique.

Puisque le type dynamique n'est pas présent dans l'implémentation des méthodes partagées, l'implémentation homogène nécessite que toutes les références vers des types ouverts dans le corps d'une méthode soient résolues lors de l'exécution. Il faut donc fournir un mécanisme pour obtenir le type exact à partir d'un type ouvert depuis le corps de la méthode.

Test de sous-typage Avec l'implémentation homogène, la nécessité de représenter les relations de sous-typage entre les variations génériques s'ajoute aux problèmes de résolution des types ouverts et de représentation des variations génériques. Les approches présentées dans le chapitre 2 doivent donc être étendues pour prendre en compte ce genre de relations de sous-typage.

Dans la suite de cette section, nous présentons les solutions utilisées par deux implémentations homogènes de la généricité : l'implémentation du compilateur EIFFELSTUDIO et l'implémentation de la plate-forme MONO [1] pour C#.

Puisqu'aucune documentation expliquant l'implémentation de ces deux moteurs n'existe, la description que nous faisons ici est déduite de l'analyse du code source et par ré-ingénierie inverse.

3.3.1 Implémentation dans EiffelStudio

Cette section décrit l'implémentation utilisée par le compilateur EIFFELSTUDIO version 7 pour le langage EIFFEL dans les modes incrémentaux *melt* et *freeze*. Ces deux modes de compilation génèrent une représentation homogène des classes génériques.

Représentation des types génériques à l'exécution Comme présenté en figure 3.1, EIFFELSTUDIO utilise une combinaison de descripteurs pour représenter les types génériques à l'exécution.

Le descripteur de classe (*class type descriptor* dans EIFFELSTUDIO) contient les données de la classe ainsi que la table de sous-typage tel que présenté au chapitre 2 sans aucune information sur la généricité. Les types génériques disposent d'un descripteur supplémentaire, le descripteur de type générique (*generic type descriptor*). Ce descripteur permet de représenter un type générique tel qu'il est connu à la compilation. Il contient l'identifiant unique du type générique et le nombre de paramètres de type sous la forme d'entiers ainsi qu'un pointeur vers la liste des paramètres de type génériques.

---

1. MONO est une implémentation libre de .NET.

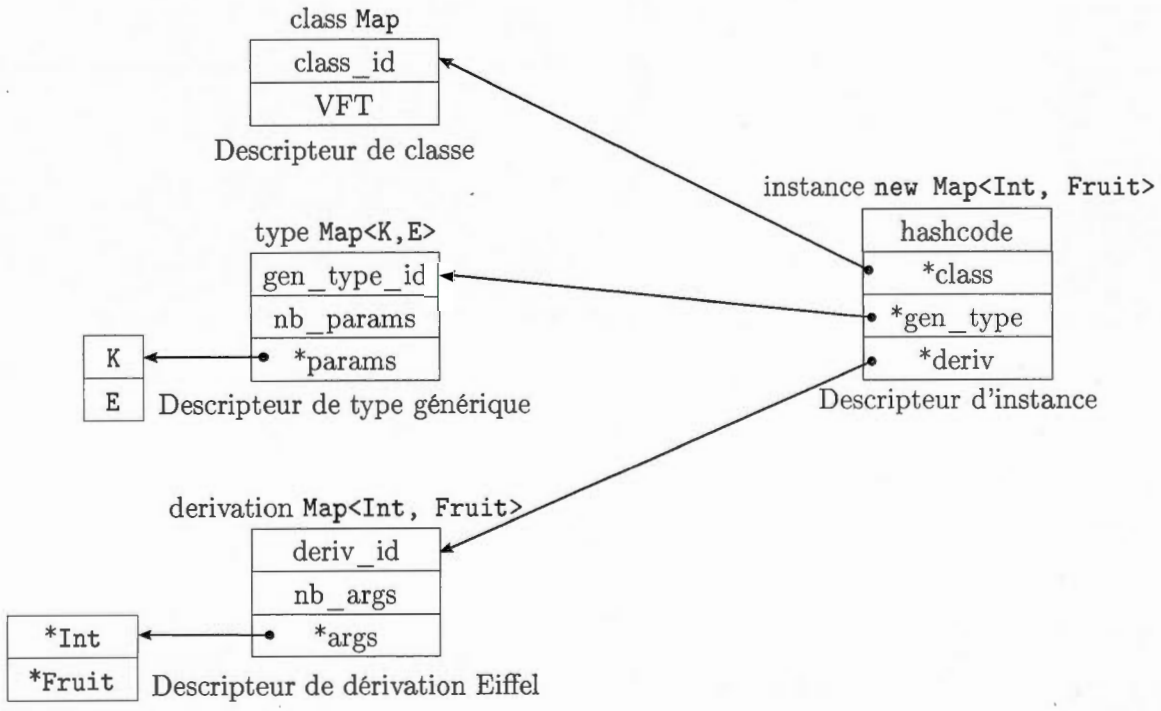

Figure 3.1 Représentation des types génériques à l'exécution avec EIFFELSTUDIO. L'instance créée par l'instruction `new Map<Int, Fruit>` est représentée par un descripteur d'instance pointant vers trois descripteurs : le descripteur de classe, le descripteur de type générique et le descripteur de dérivation Eiffel.

Mécanisme de résolution de type Pendant l'exécution, quand un nouvel objet générique est instancié, le descripteur de type générique est évalué pour calculer le descripteur de dérivation Eiffel (*eiffel derivation descriptor*). Le descripteur de dérivation Eiffel permet de représenter un type résolu. Il contient l'identifiant unique de la dérivation et le nombre de paramètres génériques du type résolu sous la formes d'entiers ainsi qu'un pointeur vers la liste des arguments de type correspondants. Le descripteur de dérivation est référencé dans le descripteur d'instance. Deux types génériques identiques partagent le même descripteur de dérivation.

La liste des arguments de types est construite à partir du descripteur de type générique. Pour la résolution, les identifiants des paramètres de types contenus dans le descripteur de type générique sont copiés dans le descripteur de dérivation. Si le paramètre de type rencontré dans le descripteur de type générique est ouvert, il est d'abord évalué grâce au receveur courant avant d'être ajouté au descripteur de dérivation.

La transformation d'un paramètre de type ouvert en un argument de type résolu se fait à l'aide d'un algorithme récursif. Chaque type générique pouvant contenir d'autres types génériques, ce processus peut devenir coûteux en temps pour la création d'un descripteur de dérivation d'un type générique complexe.

Test de sous-typage À la compilation, les relations entre les types génériques qui peuvent être déterminées statiquement sont stockées dans une matrice binaire. Les autres relations seront découvertes au cours de l'exécution.

Pour chaque nouveau type découvert à l'exécution, un identifiant unique est généré pour ce type et deux tables binaires sont créées. La première table est utilisée pour déterminer si oui ou non le résultat d'un test de sous-typage a déjà été calculé pour chacun des autres types existants. La seconde table est utilisée pour mettre en cache le résultat du test de sous-typage. Les tests de sous-typage restants seront alors calculés de manière paresseuse, au moment où le programme en a besoin.

Pour chaque nouveau test de sous-typage impliquant un type ouvert, le résultat est calculé à l'aide d'une méthode, non constante en temps, qui parcourt l'arbre d'héritage des types pour le placer en cache dans la seconde table. Les tests de sous-typage suivants seront ensuite réalisés en temps constant grâce à cette table de cache mais à travers une fonction.

Même si la complexité temporelle du test de sous-typage impliquant des types ouverts peut être considérée comme amortie, cette approche nécessite d'élargir les tables à chaque nouveau type générique rencontré lors de l'exécution. Cette opération peut s'avérer coûteuse en temps de calcul.

3.3.2 Implémentation dans Mono C#

À l'instar de JAVA, le langage C# fonctionne au travers d'une machine virtuelle. Le code source C# est tout d'abord compilé dans un langage intermédiaire indépendant des processeurs appelé *Common Imtermediate Langage* (*CIL*). Ce code intermédiaire est ensuite compilé en langage machine par la machine virtuelle de la plate-forme .NET nommée *Common Language Runtime* (*CLR*). Suivant le principe de l'implémentation homogène de la généricité, le code compilé en langage machine par le *CLR* est partagé par toutes les instances d'une même classe.

Le code intermédiaire d'une application est découvert au fil de l'exécution par le *CLR*. Le code intermédiaire des classes nécessaires au fonctionnement du programme est chargé de manière paresseuse au moment où le programme en a besoin. C'est ce qui s'appelle le chargement dynamique. Le chargement dynamique d'une classe se fait grâce à un *loader*.

Contrairement à JAVA, la généricité n'est pas effacée lors de la compilation du code C# vers le code intermédiaire. Le *CLR* est donc capable de déterminer si une classe est générique et quels sont les arguments de type générique utilisés pour instancier un type.

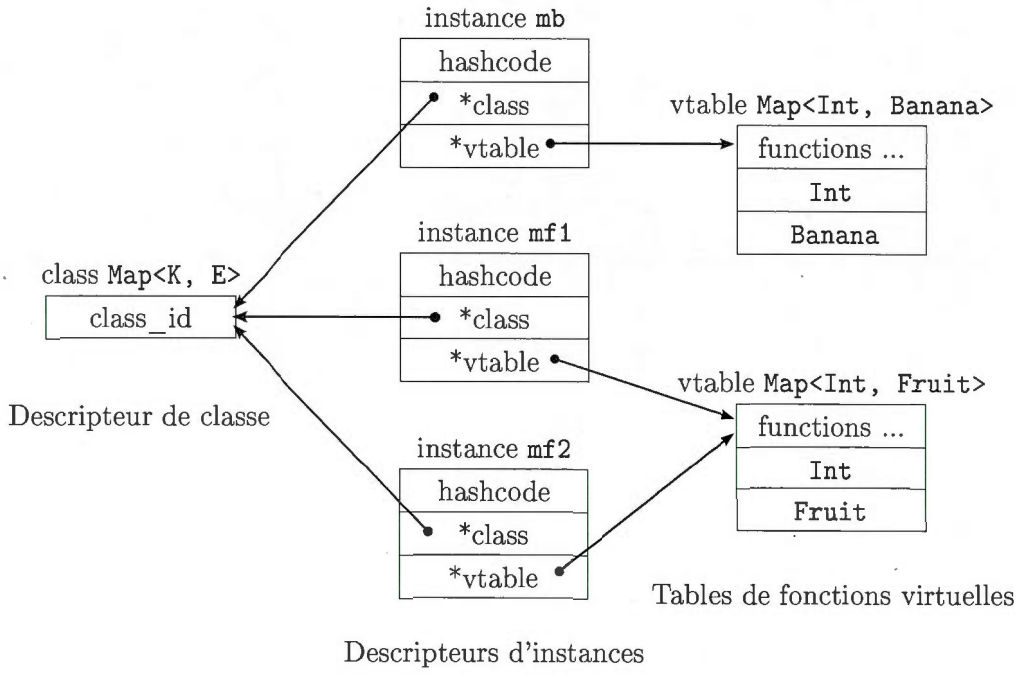

Figure 3.2 Représentation des types génériques à l'exécution avec MONO C#. Le descripteur d'instance `mb` représente un objet de type `Map<Int, Banana>`. Les descripteurs d'instances `mf1` et `mf2` représentent des objets de types `Map<Int, Fruit>`. Le descripteur de classe `Map<K, E>` est partagé entre les trois instances de la classe. La table de fonction virtuelle descripteur de type `Map<Int, Fruit>` est partagé entre `mf1` et `mf2`.

Représentation des types génériques à l'exécution Dans l'implémentation proposée par MONO telle que décrite en figure 3.2, l'implémentation des méthodes d'une classe générique est partagée par ses variations génériques au travers des tables de fonctions virtuelles (Kennedy et Syme, 2001). Chaque variation générique d'une classe possède sa propre table de fonctions virtuelles. Toutes les instances d'une variation générique peuvent quant à elles partager cette table. Deux variations génériques sont différentes si leurs tables de fonctions sont différentes.

À l'exécution, lorsqu'un type générique est instancié, le *loader* vérifie si une table de fonctions virtuelles pour ce type existe déjà afin de la réutiliser. Dans le cas contraire il

crée une nouvelle table, laquelle sera alors partagée par toutes les prochaines instances de ce type générique.

La création d'une table de fonctions pour un type consiste à importer la table de la super-classe sur laquelle il est basé, puis à ajouter les méthodes et paramètres de types introduits par sa classe.

Pour chaque méthode dans la table, le *loader* insère un pointeur vers une souche d'implémentation (*stub*) à sa position. Le corps de la méthode ne sera compilé en langage machine par le compilateur juste à temps (*JIT compiler*) que lors de sa première invocation. Les invocations suivantes bénéficieront du code déjà compilé.

Les arguments de types génériques sont ajoutés à la suite des méthodes dans la table de fonctions virtuelles. Lors de l'instanciation du type générique, le *loader* détermine les arguments de types utilisés et leur assigne une position invariante dans la table.

Mécanisme de résolution de type Il reste à gérer le cas des types ouverts qui ne sont pas des paramètres de types. Ces types ouverts sont eux-aussi stockés dans la table de fonctions virtuelles. Tout d'abord, les positions des types ouverts utilisés dans les super-classes sont importées via la copie de la table de fonctions virtuelles de la super-classe. Le *loader* détermine ensuite les types ouverts utilisés dans la classe générique du type instancié et assigne à chaque type ouvert une position invariante dans la table. La structure de la table est donc déterminée une fois pour toute au moment du chargement de la classe par le *loader*.

Le contenu de la table aux positions des types ouverts représente les types résolus pour la variation générique et est rempli de manière paresseuse. Lors du premier accès à la case d'un type ouvert, le type résolu est déterminé à l'aide d'une technique lente en prenant en compte les arguments de types puis le résultat est stocké dans la table pour les prochains accès.

Test de sous-typage C# est un langage à héritage simple de classe mais à héritage multiple d'interface. Deux mécanismes de sous-typage différents sont utilisés pour réaliser les tests de sous-typage selon qu'il s'agisse d'un test de sous-typage avec une classe ou avec une interface.

Le test de sous-typage pour les classes est implémenté à l'aide du test de Cohen tel que présenté au chapitre 2. Comme présenté au chapitre 1, C# n'autorise pas la covariance sur les classes, le support de l'héritage multiple n'est pas nécessaire à ce niveau.

Le test de sous-typage pour les interfaces est, quant à lui, réalisé à l'aide de matrices binaires telles que présentées au chapitre 2.

Implémentation semi-hétérogène C# est considéré dans certains travaux (Sallenave et Ducournau, 2012; Morandat, 2010) comme une implémentation semi-hétérogène de la généricité. En effet, pour des raisons de performances, C# utilise à la fois une implémentation homogène et une implémentation hétérogène. Chaque instance générique utilisant des types primitifs comme `int` ou `char` en tant qu'argument de type (par exemple `List<int>` ou encore `List<char>`) bénéficie d'une implémentation spécifique. Les types génériques basés sur des arguments de types non-primitifs comme `List<Object>` ou `List<Banana>` partagent quant à eux la même implémentation.

Finalement, la solution proposée pour .NET permet de représenter des types génériques exacts dans une implémentation homogène tout en conservant un mécanisme de résolution en temps constant. Le code des méthodes peut être partagé afin de préserver de l'espace sans nuire aux performances à l'exécution. L'utilisation d'une machine virtuelle associée à la compilation paresseuse permet à .NET de ne pas compiler du code qui ne sera jamais utilisé.

En contrepartie, il est plus difficile d'optimiser le code machine généré pour une méthode puisque ce code peut être partagé avec d'autres instances qui ne sont pas encore connues au moment de la compilation du corps des méthodes par le JIT. De plus, l'utilisation d'index incrémentiels pour les positions des types ouverts dans les tables de fonctions virtuelles est incompatible avec l'héritage multiple.

3.4 Conclusion

Ce chapitre présente trois formes d'implémentation de la généricité dans les langages à objets : l'implémentation effacée, l'implémentation hétérogène et l'implémentation homogène.

L'implémentation effacée peut être considérée comme une implémentation permettant de faire de la généricité avec un langage machine non-prévu pour le faire. Il s'agit d'une solution de contournement imaginée pour JAVA afin de préserver la compatibilité avec les machines virtuelles existantes.

L'implémentation hétérogène représente chaque variation générique comme une classe à part entière. Chacune possède son propre descripteur et sa propre implémentation en langage machine. L'implémentation hétérogène, en plus de nécessiter plus de mémoire pour stocker toutes ces implémentations, pose un problème d'expressivité puisqu'elle interdit toute définition récursive de types génériques. En termes de génie logiciel, cette implémentation empêche toute compilation indépendante des classes génériques et donc le partage de bibliothèques pré-compilées. En revanche, utilisée dans le cadre d'une compilation globale, elle permet d'appliquer des optimisations dans le code machine généré pour chaque variation générique.

L'implémentation homogène est la plus élégante. L'implémentation du corps des méthodes n'a pas besoin d'être dupliquée pour chaque variation et peut donc être partagée. Le coût en mémoire de la généricité est de ce fait réduit au strict minimum. Du point de vue des performances, la solution proposée par .NET montre qu'il est possible d'obtenir un mécanisme de résolution en temps constant.

Malgré ses avantages, la solution proposée par .NET a été imaginée dans un contexte d'héritage simple de classe. Dans ce mémoire nous nous intéressons au cas des langages à héritage multiple. De par ses performances, la solution présentée par EIFFELSTUDIO n'est pas concevable pour l'implémentation en NIT. Le prochain chapitre s'intéresse donc à la possibilité d'appliquer la solution utilisée par .NET dans un contexte d'héritage multiple sans perte de performances.

CHAPITRE IV

IMPLÉMENTATION HOMOGÈNE EFFICACE DE LA GÉNÉRICITÉ COVARIANTE EN HÉRITAGE MULTIPLE

Le chapitre précédent présente deux implémentations homogènes pour une généricité covariante dans un langage à objets. La solution appliquée à EIFFELSTUDIO est certes compatible avec l'héritage multiple mais souffre d'une implémentation complexe qui ne permet pas de bonnes performances. La solution implémentée dans C# propose en revanche de bonnes performances mais n'est pas conçue pour supporter l'héritage multiple de classes.

Le mécanisme de résolution des types ouverts utilisé par C# est intéressant car il permet une résolution en temps constant contrairement à EIFFELSTUDIO. Dans ce chapitre, nous nous proposons d'adapter l'approche de C# à un héritage multiple et de l'implémenter dans le langage NIT.

Dans la solution imaginée pour C#, la résolution des types ouverts se fait grâce à la table de fonctions virtuelles. La table est alors spécifique à un type. Cependant les pointeurs de fonctions d'une même classe sont identiques entre tous ses types génériques. Ceux-ci pourraient alors être factorisés au niveau du descripteur de classe. C'est là un premier problème que nous nous proposons de résoudre ici grâce à une structure supplémentaire, le descripteur de type.

Le second problème tient à la construction de la table par héritage de la table de la super-classe. Cette approche, efficace en héritage simple, pose un problème bien connu

de conflits lorsqu'appliquée à l'héritage multiple. Le lecteur trouvera une explication complète des problèmes de construction de tables en héritage multiple dans d'autres travaux (Ducournau, Morandat et Privat, 2009). Nous proposons ici une solution basée sur les implémentations existantes des mécanismes objets adaptés à l'héritage multiple tels que présentés dans le chapitre 2.

Les sections suivantes présentent notre proposition pour une implémentation homogène de la généricité. Il est d'abord décrit la technique employée pour représenter les types génériques de manière homogène, puis l'implémentation utilisée pour le test de sous-typage, la technique de résolution des types ouverts et enfin l'implémentation faite de cette solution en NIT.

4.1 Représentation homogène des types génériques

Avec une implémentation homogène toutes les variations génériques d'une classe partagent la même implémentation. Il est donc possible de leur faire partager la même table de fonctions virtuelles. Un envoi de message sur une instance générique se comporte dès lors comme un envoi de message sur toute autre instance. Il faut d'abord accéder au descripteur de classe depuis le descripteur de l'instance puis accéder à la case correspondante dans la table de fonctions virtuelles.

Soit une instance x de la classe X et une méthode foo stockée à la position pos_{foo} dans la table de fonctions virtuelles du descripteur de classe de X. L'envoi du message $x.foo()$ correspondra alors à :

$$x.class.vft[pos_{foo}]()$$

Si le descripteur de classe peut-être partagé entre toutes les variations génériques de la classe, la table de sous-typage, quant à elle, ne peut plus être stockée dans ce descripteur. En effet, chaque variation générique possède ses propres relations de sous-typage qui ne sont pas toujours partagées avec les autres variations génériques.

Pour stocker la table de sous-typage, nous ajoutons une nouvelle structure spécifique aux types : le *descripteur de type*. Toute instance, générique ou non, est alors associée

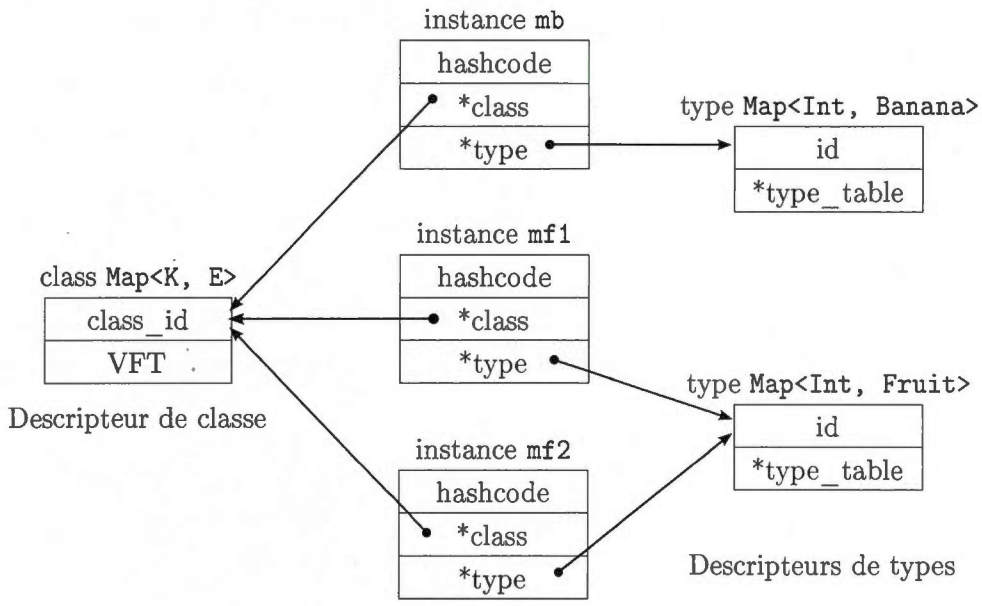

Figure 4.1 Exemple de descripteurs de types. Le descripteur d'instance `mb` représente un objet de type `Map<Int, Banana>`. Les descripteurs d'instances `mf1` et `mf2` représentent des objets de types `Map<Int, Fruit>`. Le descripteur de classe `Map<K, E>` est partagé entre les trois instances de la classe. Le descripteur de type `Map<Int, Fruit>` est partagé entre `mf1` et `mf2`.

à un descripteur de type. Ce descripteur de type est accédé depuis le descripteur de l'instance via un pointeur. Ainsi, au prix d'une indirection supplémentaire, toutes les instances d'un même type peuvent partager le même descripteur. La figure 4.1 donne un exemple d'utilisation des descripteurs de types.

Cette approche permet une représentation homogène des types génériques dans un contexte d'héritage multiple en limitant au maximum la duplication des descripteurs.

4.2 Implémentation du test de sous-typage

La représentation proposée est compatible avec les mécanismes existants de sous-typage multiple tel que nous les avons présentés dans la section 2. Le descripteur de type

contient dès lors l'identifiant unique du type, sa table de sous-typage et toute information nécessaire au mécanisme de sous-typage. La couleur utilisée pour la coloration ou le paramètre de hachage pour le hachage parfait sont ainsi stockés dans le descripteur de type.

Considérant le test de sous-typage `x instanceof T` où `T` est un type résolu comme `Integer` ou `Array<Fruit>`. Le test de sous-typage consiste alors à accéder au descripteur de type de `x` puis à accéder à la table de sous-typage *st* depuis le descripteur de type. Par exemple, en utilisant la coloration :

$$T.color < x.type.st.length \;\&\&\; x.type.st[T.color] == T.id$$

Cette approche permet la résolution des tests de sous-typage impliquant des types résolus en temps constant. La résolution des types ouverts comme `E` ou `Array<E>` ne peut quant à elle être réalisée en utilisant uniquement le descripteur de type. Nous présentons la structure utilisée pour gérer le cas des types ouverts dans la prochaine section.

4.3 Mécanisme de résolution des types génériques ouverts

Cette section présente le mécanisme de résolution des types ouverts en types résolus dans notre solution. Il est d'abord présenté la table de résolution et le mécanisme de résolution. Puis il est expliqué comment est construite cette table de résolution en fonction du schéma de compilation envisagé.

4.3.1 Table de résolution

Le mécanisme de résolution est réalisé grâce à la même structure que C#. Nous utilisons un dictionnaire pour associer chaque type dynamique ouvert à un type résolu. Contrairement à C#, le dictionnaire n'est pas stocké dans la table de fonctions virtuelles mais dans le descripteur de type. Il est ainsi partagé par toutes les instances d'un même type. Nous appelons ce dictionnaire une table de résolution.

À chaque classe C, est associée la liste des types ouverts utilisés dans son corps. Cette liste est notée $C.ots$. Les paramètres de types de C ainsi que tous les types ouverts utilisés dans ses super-classes (directes ou indirectes) sont compris dans $C.ots$.

À chaque type ouvert OT présent dans $C.ots$ est assigné une position unique notée $OT.pos$. Pour chaque type T instancié dans la classe C, on calcule le type résolu RT correspondant à OT dans C.

RT est alors stocké dans la table de résolution du type T notée $T.rt$ à la position $OT.pos$. Ainsi, chaque type ouvert OT instancié dans la classe C est associé à un type résolu RT dans la table de résolution de T.

4.3.2 Test de sous-typage

Soit `self` le receveur où apparaît le test de sous-typage vers un type ouvert. Considérant un test de type `x instanceof UT` dans `self`. Il faut d'abord accéder à la table de résolution depuis le descripteur de type de `self`. Puis obtenir le type résolu RT à la position $OT.pos$ dans la table de résolution. Le test de sous-typage peut alors être résolu par les moyens traditionnels en utilisant RT :

$$RT = self.type.rt[OT.pos]$$

$$RT.pos < x.type.st.length \;\&\&\; x.type.st[RT.pos] == 1$$

La complexité temporelle pour résoudre un test de sous-typage avec un type ouvert est donc en temps constant $O(1)$. Les tables de résolution peuvent être compressées de la même manière que les tables de sous-typage en appliquant la coloration ou le hachage parfait.

La liste des types ouverts utilisés au sein d'une classe peut être déterminée à la compilation séparée du module. Il suffit de repérer dans le code de la classe les références vers des types étant ou contenant des paramètres de types. La construction de la table de résolution va quant à elle dépendre du schéma de compilation utilisé.

4.3.3 Compilation séparée et édition de liens globale

Dans le cadre de la compilation séparée avec édition de liens globale, nous ré-exploitons l'implémentation des mécanismes objets compatible avec l'héritage multiple à savoir la coloration.

La liste des types ouverts $C.ots$ utilisés dans la classe C est construite à la compilation séparée du module.

À chaque type ouvert OT est associé un identifiant unique $OT.id$ quelque soit la classe dans laquelle il est utilisé et une position invariante (couleur) notée $OT.color$ grâce à un algorithme de coloration.

L'algorithme utilisé est proche de celui proposé par Ducournau pour la coloration des classes à la différence que le graphe de conflit ne contient que des types ouverts et qu'il est basé sur le contenu des tables de résolution. Deux types ouverts sont considérés en conflits s'ils sont utilisés par la même classe. Cela signifie qu'ils seront présents dans la même table de résolution et donc qu'ils ne peuvent partager la même position.

Le graphe de conflit et la coloration sont calculés lors de l'édition de liens globale. Les meilleurs résultats en termes de compacité des tables sont obtenus en colorant les types selon une extension linéaire descendante, du super-type au sous-type (Ducournau, 2011a). C'est ce qu'on appelle l'ordre de linéarisation. L'algorithme de coloration utilisé est donné en figure 4.2.

Chaque type résolu RT dans T correspondant à un type ouvert OT dans C est ensuite ajouté dans la table de résolution $T.rt$ à la position $OT.color$. Le pointeur vers le type résolu RT se trouve alors à la position $OT.color$ dans la table de résolution :

$$RT = self.type.rt[OT.color]$$

Contrairement au test de sous-typage, nous avons la certitude que tous les accès aux tables de résolution sont corrects. Il n'y pas besoin de rajouter de test sur la longueur

```
  // Calculs préliminaires
1 foreach type ouvert OT dans l'ensemble des types ouverts utilisés do
2 |   calculer de OT.id, l'identifiant unique de OT;
3 end
  // Calcul du graphe de conflit des types ouverts
4 foreach classe C dans l'ordre de linéarisation do
5 |   foreach type ouvert OT dans C.ots do
6 |   |   rajouter au graphe de conflit les éléments du produit OT × C.ots qui
  |   |   n'y sont pas déjà;
7 |   end
8 end
  // Coloration des types ouverts
9 foreach classe C dans l'ordre de linéarisation do
10|   foreach type ouvert OT dans C.ots do
11|   |   affecter à OT l'une des couleurs libres minimales;
12|   |   propager la couleur de OT à tous les types ouverts en conflit avec OT;
13|   end
14 end
```

Figure 4.2 Algorithme de coloration des tables de résolution.

de la table. La complexité temporelle pour transformer un type ouvert en un type résolu est donc en temps constant $O(1)$.

Comme dans le cas du sous-typage, la coloration donne de bons résultats dans la compression des tables de résolution mais nécessite un long traitement pour le calcul des couleurs. De ce fait, elle est à considérer dans un schéma de compilation avec édition de liens globale.

4.3.4 Compilation séparée et chargement dynamique

Pour le chargement dynamique, la coloration est une approche trop lente pour pouvoir calculer les tables de résolution à chaque nouveau chargement de classe. L'implémentation par matrice binaire peut être un bon candidat mais est généralement inefficace pour la compression des tables. Nous préférons donc l'implémentation proposée par Ducournau basée sur le hachage parfait.

Le hachage parfait est possible puisque l'on connaît au moment de la création de la table de résolution (au chargement de la classe en fait) toutes les valeurs qu'elle doit contenir.

À chaque type ouvert OT est associé un identifiant unique $OT.id$ qui sera utilisé par la fonction de hachage. Puis, pour chaque type T et sa table de résolution $T.t$, nous calculons une table de résolution sans collision $T.rt$. Le paramètre de hachage $T.rt\_hash$ est alors stocké dans le descripteur de type de T. Soit une association entre le type ouvert OT et le type résolu RT dans C, le hachage de $OT.id$ par $T.rt\_hash$ donne la position invariante de RT dans $T.rt$:

$$RT = self.type.rt[phash(OT.id, self.type.rt\_hash)]$$

La complexité temporelle théorique pour transformer un type ouvert en type résolu reste constante $O(1)$ mais est en pratique plus lente que pour la coloration à cause du calcul de la position par hachage.

En revanche, l'instanciation d'un nouveau type dans le cadre du chargement dynamique nécessite la création de la table de résolution à l'exécution puis son remplissage. Puisque remplir la table peut-être une opération coûteuse en fonction du nombre de types à résoudre, nous préférons ici une approche amortie basée sur un calcul paresseux des résolutions en fonction des besoins.

4.4 Implémentation en Nit

Nous implémentons notre solution dans un compilateur pour le langage NIT. Pour notre implémentation, plutôt que de travailler à partir du compilateur original du langage `nitc`, nous avons choisi de partir de `nitg`. `nitg` utilise un nouveau méta-modèle, plus robuste et avec une API plus simple. À partir de ce compilateur, nous avons dérivé un nouveau compilateur : `nitg-s` qui applique une politique de typage dynamique covariante.

Ce compilateur a été développé pour fournir un maximum de flexibilité dans le test de différentes implémentations et propose de nombreuses options de configuration. Considérant la politique de typage dynamique, les *casts* de covariance peuvent être activés ou désactivés. Considérant l'implémentation du mécanisme de sous-typage et du mécanisme de résolution, les approches par matrice binaire, coloration ou hachage parfait peuvent être choisies alternativement. Des métriques sur l'utilisation des tests de sous-typage peuvent être collectées à la compilation ainsi qu'au cours de l'exécution des programmes compilés. Les deux autres mécanismes objets que sont l'accès aux attributs et l'envoi de message sont implémentés en utilisant la coloration (Ducournau, 2011a). Le ramasse miettes utilisé est celui décrit par Boehm (Boehm et Spertus, 2009).

Le schéma de compilation utilisé par `nitg-s` est réellement séparé avec édition de liens globale. *Réellement séparé* car chaque module NIT est compilé en un fichier objet (.o) indépendamment du programme NIT qui l'utilise. Lors de la phase d'édition de liens globale, les tables et autres services globaux sont calculés et stockés dans un autre fichier objet, celui du programme. Les positions dans les différentes tables sont associées à des symboles qui sont utilisés dans les modules séparés puis calculés lors de la phase d'édition de liens globale dans le fichier objet du programme. L'éditeur de liens standard est suffisant pour produire l'exécutable final. Aucune optimisation globale n'est appliquée au code machine généré. Les seules optimisations faites concernent la construction des différentes tables.

Dans la pratique, les compilateurs `nitg` et `nitg-s` sont combinés dans un seul et même exécutable appelé `nitg`. L'activation du compilateur séparé est faite à l'aide de l'option `-separate`.

Les tables de résolution sont initialisées grâce à une analyse RTA (*Rapid Type Analysis*) (Bacon et Sweeny, 1996) durant la phase d'édition de liens globale. L'analyse RTA peut être considérée comme excessive dans ce contexte mais elle garantit que tous les types et toutes les variations génériques nécessaires à l'exécution sont déjà connus.

Néanmoins, certaines constructions permettent de générer un nombre non-borné de types génériques. C.est le cas, si le code de la classe générique `List<E>` utilise par exemple un `new List<List<E>>`. Ce genre de définition récursive peut générer un nombre infini de types génériques et empêche l'analyse RTA de terminer. À l'instar de l'implémentation hétérogène de C++, nous sommes contraints d'interdire toute définition récursive des types génériques et c'est là un inconvénient majeur de notre approche.

4.4.1 Types nullables

Les types nullables (Gélinas, Gagnon et Privat, 2009), présents dans NIT, permettent de garantir statiquement l'absence d'erreurs de dé-référencement de références nulles en annotant chaque type en fonction qu'il soit nullable ou non. Les types nullables permettent de nouvelles constructions dans le cas du test de sous-typage. Ainsi, il est possible de demander si `x instanceof nullable T` où `T` est un type. Les types `T` et `nullable T` peuvent exister en même temps et il est important de 1) pouvoir différencier ces types à l'exécution et 2) pouvoir résoudre le test de sous-typage correctement.

Pour gérer ce genre de cas, nous rajoutons dans le descripteur du type T un booléen $T.is\_nullable$ permettant de spécifier si T est nullable ou non. `T` et `nullable T` posséderont chacun leur descripteur avec des identifiants différents.

Ainsi le test `x instanceof nullable T` où `x = null` retournera vrai si et seulement si :

$$x == null \;\&\&\; T.is\_nullable$$

4.4.2 Types virtuels

Le langage NIT utilise des types virtuels. Introduits par Torgersen (Torgersen, 1998), les types virtuels combinent généricité et covariance. Les types virtuels peuvent être considérés comme des attributs de classes représentant des types. Dans une déclaration de classe, chaque type virtuel est borné par un type. La borne d'un type virtuel peut être ensuite redéfinie dans les sous-classes de manière covariante.

Utilisés avec la généricité, les types virtuels peuvent être bornés par des paramètres de types ou des types ouverts. Ainsi, la borne du type virtuel dépend elle aussi du receveur courant et ne peut pas être stockée dans le descripteur de classe.

Dans notre implémentation pour NIT, les types virtuels sont considérés comme des types ouverts et utilisent les mêmes mécanismes de résolution et de sous-typage.

4.5 Conclusion

Puisqu'aucune implémentation homogène de la généricité covariante n'est satisfaisante en termes de performances, nous avons proposé d'adapter la solution proposée par C# à l'héritage multiple. Cette implémentation permet de résoudre deux problèmes rencontrés par l'implémentation de C#.

Tout d'abord, les tables de fonctions virtuelles peuvent être factorisées par classe et non plus par type. Les tables de sous-typage peuvent quant à elles être factorisées par type. Ces deux améliorations sont possibles grâce à un nouveau descripteur que nous nommons descripteur de type. Ce descripteur permet de représenter les attributs communs à plusieurs variations génériques d'une même classe. Le descripteur de type est accessible depuis un pointeur dans le descripteur d'instance. Il contient la table de sous-typage permettant de résoudre les tests de sous-typage en héritage multiple en temps constant.

Le descripteur contient aussi une table que nous nommons table de résolution. Cette table permet de transformer un type ouvert en un type résolu en fonction du type du receveur. La résolution se fait en temps constant et permet de récupérer un pointeur

vers le type résolu à partir d'un type ouvert. Ce type résolu peut ensuite être utilisé dans un test de sous-typage comme tout autre type. Le test de sous-typage contre un type ouvert se fait donc lui aussi en temps constant.

La construction de la table de résolution en héritage multiple se fait en réutilisant les mécanismes objets adaptés à ce type d'héritage. Dans le cadre d'une compilation séparée avec édition de liens globale, on préférera la coloration. Dans le cadre d'une compilation séparée avec chargement dynamique, on préférera le hachage parfait. Les matrices binaires peuvent quant à elles être utilisées dans les deux configurations.

Cette solution est implémentée dans le compilateur séparé `nitg-s` pour le langage NIT. Ce compilateur permet une implémentation homogène de la généricité dans un langage à typage statique et en héritage multiple en suivant un politique de typage covariante et en supportant les types virtuels et les types nullables. Dans le prochain chapitre, nous étudions les performances de ce compilateur grâce à des simulations et des programmes concrets.

CHAPITRE V

EXPÉRIMENTATIONS ET VALIDATION

Ce chapitre présente une validation de notre solution au travers de plusieurs expérimentations sur des programmes tant artificiels que réels. Le compilateur `nitg-s` permet une implémentation homogène de la généricité selon une politique covariante, nous souhaitons maintenant vérifier ses performances. Ainsi, nos expérimentations suivent deux objectifs.

Tout d'abord, nous voulons comparer la vitesse d'exécution des programmes compilés selon notre solution par rapport à d'autres langages et compilateurs. Dans la première partie de notre expérimentation, nous utilisons des hiérarchies de classes générées pour évaluer les performances de 10 compilateurs et 2 machines virtuelles. Les programmes générés sont conçus pour utiliser l'implémentation des mécanismes de généricité au maximum. L'objectif est de montrer que notre implémentation offre des performances dans le même ordre de grandeur que les solutions existantes.

La seconde partie de l'expérimentation consiste à évaluer la vitesse d'exécution sur de vrais programmes NIT. Basé sur des programmes existants utilisant la généricité, nous comparons les performances du compilateur homogène `nitg-s` par rapport à un compilateur pratiquant l'effacement. Nous montrons ici que la covariance entraîne un faible sur-coût par rapport à la politique effacée ou invariante.

Enfin, toujours en utilisant des vrais programmes NIT, nous comparons les trois mécanismes de sous-typage présentés au chapitre 2, la matrice binaire, la coloration et le hachage parfait. Nous vérifions ainsi la compatibilité de notre solution avec ces trois mécanismes et comparons leurs performances.

5.1 Simulations sur programmes artificiels

Dans cette section, nous comparons les comportements de plusieurs moteurs d'exécution (10 compilateurs, 2 machines virtuelles) pour les langages C++, C#, EIFFEL, JAVA, SCALA et NIT sur des programmes artificiels et non-réalistes.

Contrairement à l'étude proposée par Garcia (Garcia et al., 2007) qui présente une comparaison des fonctionnalités génériques offertes par les langages, cette évaluation vise à comparer les implémentations du test de sous-typage générique, tous les autres mécanismes étant différents.

Ainsi, cette partie de l'expérimentation vise à répondre aux questions suivantes : i) Est-ce que nos estimations formulées dans le chapitre 4 sont correctes et représentent la réalité ? ii) Quelles sont les différences de performances entre une grande variété de politiques, d'implémentations et de compilateurs ? iii) Comment une implémentation répond-elle à un stress sur ses mécanismes de généricité ?

5.1.1 Moteurs comparés

Nous comparons les compilateurs et machines virtuelles suivantes :

g++ propose une implémentation hétérogène de la politique invariante de C++. La version utilisée est la *GNU gcc g++ (Debian 4.7.2-5) 4.7.2 - linux-x86-64 with GNU.*

clang++ propose lui aussi une une implémentation hétérogène de la politique invariante de C++. La version utilisée est la *Debian clang version 3.0-6.2 (tags/RELEASE_30/final) (based on LLVM 3.0) - linux-x86-64.*

Les compilateurs *g++* et *clang++* utilisent tous les deux la même implémentation du test de sous-typage partagée dans *libstdc++* version *4.7.2-5 Debian GNU linux-x86-64.* Tous deux utilisent le niveau d'optimisation `-O2`.

javac version *1.7.0_03* est utilisé pour compiler les programmes JAVA. Ces programmes sont lancés sur la machine virtuelle *OpenJDK Runtime (IcedTea7 2.1.2) (7u3-2.1.2-2)* suivant la politique et l'implémentation effacée de JAVA.

gcj, le compilateur GNU pour JAVA, version *4.7.2* est utilisé pour compiler ces mêmes programmes JAVA.

scalac version *2.9.1*. Les programmes SCALA sont lancés sur la même machine virtuelle que JAVA. La politique utilisée est covariante mais utilise une implémentation effacée.

gmcs est le compilateur GNU de MONO C# version *2.10.8.1*. Les programmes C# sont exécutés sur le MONO JIT COMPILER version *2.10.8.1*. MONO C# propose une implémentation homogène de la politique covariante du langage C#.

es est le compilateur EIFFEL STUDIO. Il propose une implémentation homogène de la politique covariante du langage EIFFEL. La version utilisée est la *ISE EiffelStudio 7.1.8.8986 GPL Edition - linux-x86-64*. Les programmes EIFFEL sont compilés à l'aide du niveau d'optimisation `-finalize`.

se est le compilateur SMARTEIFFEL. Il propose quant à lui une implémentation hétérogène pour EIFFEL. La version utilisée est la version *Release 2.4 (svn snaphsot 9308) - linux-x86-64*. Les programmes sont compilés avec le niveau d'optimisation `-no_check`.

nitg est le compilateur hétérogène pour la politique covariante de NIT utilisant un processus d'optimisation et de particularisation en compilation globale.

nitg-s est le compilateur implémenté selon notre solution. Il propose une représentation homogène de la politique covariante de NIT utilisant une compilation séparée avec une édition de liens globale.

Le code C généré par les deux compilateurs NIT est compilé par *gcc* version *GNU gcc g++ (Debian 4.7.2-5) 4.7.2 - linux-x86-64 with GNU* avec le niveau d'optimisation `-O2`. Le numéro du commit Git est `7e44894eb766bf5d4d58c2a45356c6344410248f`.

5.1.2 Micro-benchmarks

L'objectif des micro-benchmarks est de mesurer les performances du mécanisme de sous-typage générique. Pour ce faire, les programmes choisis pour les mesures doivent, pendant

leur exécution, solliciter fortement le mécanisme de sous-typage mais utiliser les autres mécanismes du langage au minimum.

Les programmes choisis doivent aussi être équivalents autant que possible entre les différents langages. Pour cette raison, les programmes doivent utiliser les plus petits dénominateurs communs entre les politiques et faire un usage minimum des bibliothèques ou des mécanismes spécifiques.

Les programmes de tests sont générés automatiquement par des scripts. La génération est déterministe et le même programme est généré pour tous les langages. Chaque programme généré comprend trois parties :

- une hiérarchie de classe linéaire. La hauteur de la hiérarchie est déterminée par un paramètre du script.
- une boucle de 2.500.000.000 itérations.
- un test de sous-typage dans chaque itération.

Protocole de test Les micro-benchmarks sont lancés sur un ordinateur portable Intel i7-2640M avec un processeur @2.80GHz x86_64 installé avec Debian GNU/Linux.

Le protocole de test est le suivant : 6 exécutions consécutives de chaque configuration sont faites, la première est éliminée et les temps minimum, maximum et moyen des cinq exécutions sont conservés. Le temps utilisateur est mesuré avec la commande GNU `time`.

La boucle La boucle est faite au sein d'une fonction divisée en deux boucles `for` imbriquées de 50.000 itérations chacune. La partie `else` du test n'est pas exécutée mais évite les optimisations de boucles. Le code de la boucle pour le langage JAVA est donné en figure 5.1.

La figure 5.2 donne les performances de chaque moteur d'exécution avec une boucle vide : aucun test de sous-typage n'est fait dans la boucle. L'étiquette `TYPE_TEST` est remplacée par `true`. Nous avons tenté de mettre tous les moteurs au même niveau mais la JVM applique des optimisations imprévisibles. Les autres moteurs donnent sensiblement les mêmes performances sur cette même boucle. Ces résultats mesurent uniquement le temps utilisé par la boucle à vide et peuvent être utilisés pour estimer le temps consommé par les tests de sous-typage dans les prochains résultats.

```
1  static public void test(Root a, Root b, int loops, int start) {
2    int x = start;
3    for(int i = 0; i < loops; i++) {
4      for(int j = 0; j < loops; j++) {
5        if(TYPE_TEST && x >= 0) {
6        } else { x = x - i + j; a = b;}
7      }
8    }
9    System.out.println(x);
10 }
```

Figure 5.1 Exemple de boucle principale pour le langage JAVA.

	g++	clang++	java	gcj	scala	gmcs	es	se	nitg	nitg-s
Time (s)	1.44	1.44	2.21	1.74	2.38	1.51	1.46	1.45	1.44	1.44

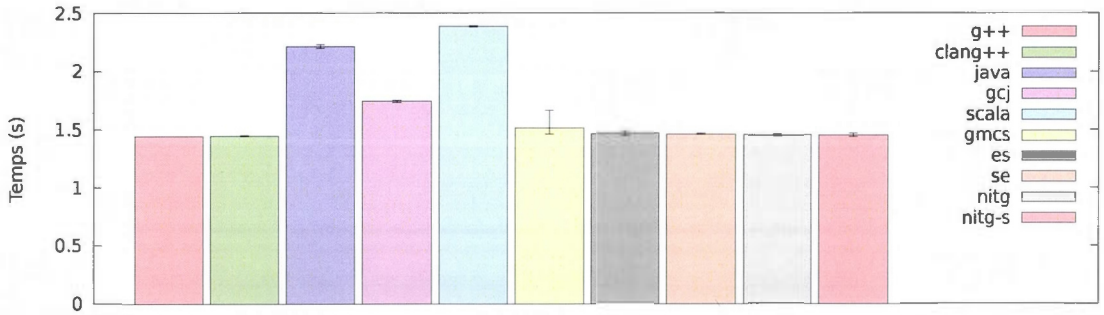

Figure 5.2 Temps d'exécution moyens des différents moteurs sur la boucle à vide. L'étiquette `TYPE_TEST` est remplacée par `true`. Les temps sont donnés en secondes.

Le test de sous-typage Trois types de hiérarchies sont générées en fonction des objectifs du micro-benchmark. Les hiérarchies générées sont en héritage simple. Le paramètre h du script détermine la hauteur de la hiérarchie. Plus précisément, le script génère : Une classe racine `R`, sans attributs. Un total de h classes génériques notées C_i où i représente la profondeur de la classe dans l'arbre d'héritage depuis `R`. Chaque classe générique prend seulement un paramètre de type. Trois configurations de hauteur sont utilisées pour les expérimentations : h=2 ; h=4 ; h=8.

Dans la boucle, l'étiquette `TYPE_TEST` est remplacée par un test entre une instance du type le plus profond de la hiérarchie et un type situé au milieu de la hiérarchie.

Les résultats donnés en figure 5.3 montrent les performances du test de type en fonction de la taille de la hiérarchie :

$$R :> C_1\langle R\rangle :> C_2\langle R\rangle :> \cdots :> C_h\langle R\rangle$$

La même hiérarchie est utilisée pour mesurer les performances sur des tests de sous-typage négatifs. Dans la boucle, l'étiquette `TYPE_TEST` est remplacée par la négation du test entre l'avant-dernier type de la hiérarchie et le dernier. Les résultats sont présentés en figure 5.4.

Les résultats donnés en figure 5.5 montrent les performances des moteurs considérant des tests de sous-typage avec covariance générique :

$$R :> C_1\langle C_1\langle R\rangle\rangle :> C_1\langle C_2\langle R\rangle\rangle :> \cdots :> C_1\langle C_h\langle R\rangle\rangle$$

Les résultats donnés en figure 5.6 montre les performances des différents moteurs considérant des types génériques imbriqués :

$$R :> C_1\langle R\rangle :> C_1\langle C_1\langle R\rangle\rangle :> \cdots :> C_1\langle C_1\langle C_1\langle \cdots\rangle\rangle\rangle$$

Note : Pour C++, les classes générées sont purement virtuelles : les méthodes et la spécialisation sont déclarées à l'aide du mot-clé `virtual`. En JAVA et *C#*, les tests de sous-typage sont fait avec des interfaces au lieu de classes pour s'assurer que la machine virtuelle utilise l'implémentation du test de sous-typage pour l'héritage multiple. La même chose est faite avec les traits en *Scala*.

	g++	clang++	java	gcj	scala	gmcs	es	se	nitg	nitg-s
$h = 2$	104.82	105.44	3.97	12.94	4.10	11.67	20.19	3.62	2.21	2.88
$h = 4$	188.23	213.00	4.02	12.98	4.12	11.38	28.88	3.62	2.19	2.88
$h = 8$	299.21	299.21	3.96	12.96	4.12	11.55	28.92	3.61	2.18	2.88

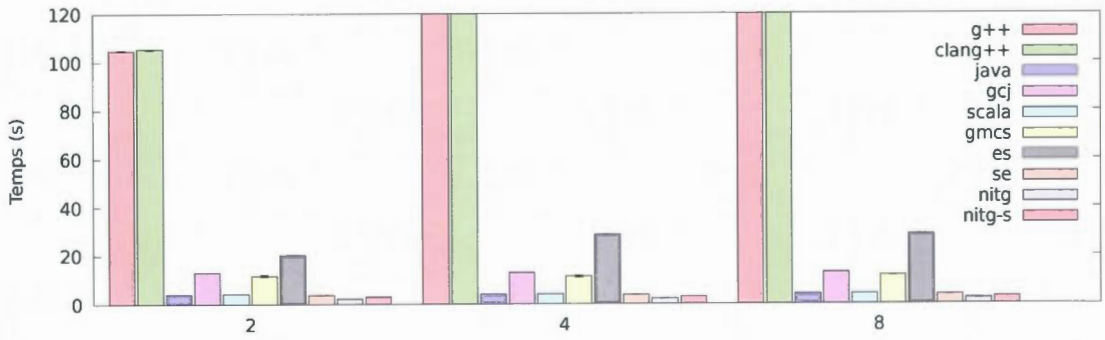

Figure 5.3 Temps d'exécution moyens des différents moteurs considérant une variation dans la hauteur de la hiérarchie de classes. L'étiquette `TYPE_TEST` est remplacée par un test de sous-typage réussi. Les temps sont donnés en secondes.

	g++	clang++	java	gcj	scala	gmcs	es	se	nitg	nitg-s
$h = 2$	69.13	86.88	48.11	12.34	47.87	11.54	30.43	3.81	2.17	2.16
$h = 4$	258.95	237.06	51.35	12.43	50.95	11.16	30.32	3.75	2.16	2.16
$h = 8$	299.21	299.21	56.76	12.33	56.83	11.50	33.61	3.67	2.16	2.16

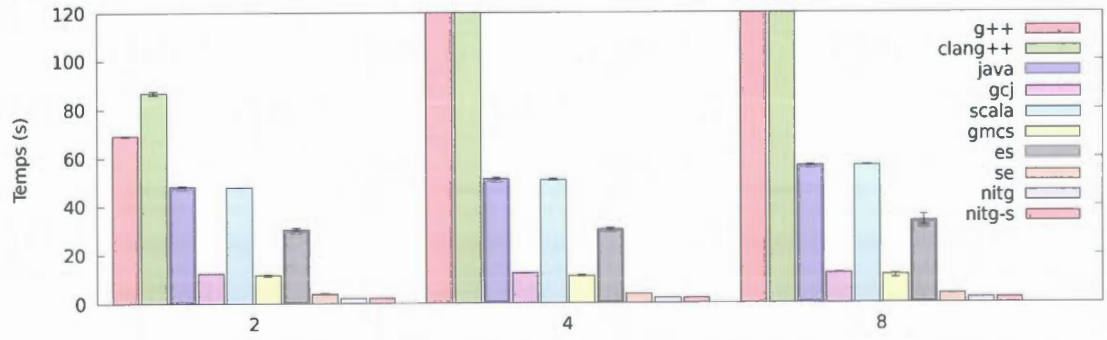

Figure 5.4 Temps d'exécution moyens des différents moteurs considérant un test de sous-typage échoué. L'étiquette `TYPE_TEST` est remplacée par la négation d'un test de sous-typage échoué. Les temps sont donnés en secondes.

	gmcs	es	se	nitg	nitg-s
$h = 2$	11.46	20.56	3.77	2.19	2.90
$h = 4$	11.55	29.41	3.72	2.19	2.89
$h = 8$	11.11	29.41	4.41	2.18	2.90

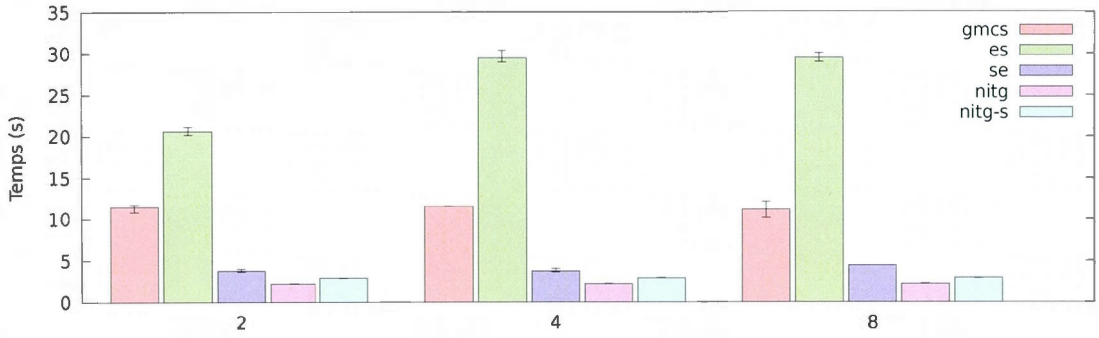

Figure 5.5 Temps d'exécution moyens des différents moteurs considérant un test de sous-typage avec covariance générique. Les temps sont donnés en secondes.

	gmcs	es	se	nitg	nitg-s
$h = 2$	11.27	28.88	2.17	2.88	2.92
$h = 4$	11.26	29.13	3.66	2.20	2.90
$h = 8$	11.55	30.47	3.70	2.16	2.88

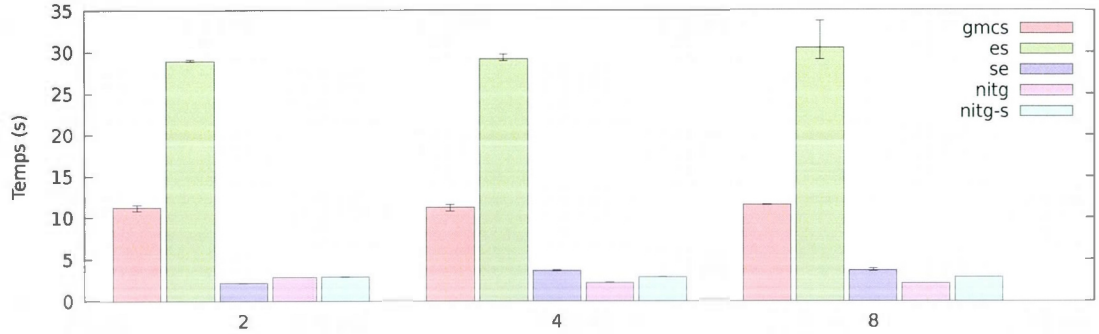

Figure 5.6 Temps d'exécution des différents moteurs considérant des types génériques imbriqués. Les temps sont donnés en secondes.

5.1.3 Discussion

Les implémentations basées sur des appels de fonctions sont les plus lentes. Les fonctions du test de sous-typage utilisées par *g++*, *clang++* et EIFFELSTUDIO sont moins efficaces que les *switches* utilisés par *nitg* et SMARTEIFFEL ou que les tables utilisées par *nitg-s*.

Toutes les implémentations du test de sous-typage ne sont pas en temps constant. Les performances de *g++*, *clang++* et EIFFELSTUDIO varient en fonction de la hauteur de la hiérarchie d'héritage. Les performances d'EIFFELSTUDIO varient aussi en fonction du niveau d'imbrication des types génériques. Plus surprenant, la JAVA JVM est non-constante entre tests de sous-typages réussis ou échoués.

Finalement, les compilateurs NIT donnent les meilleures performances sur le test de sous-typage. L'implémentation homogène utilisant notre solution dans *nitg-s* apparaît comme étant en temps constant dans tous les cas, même avec covariance. L'implémentation hétérogène optimisée de *nitg* reste plus rapide que celle de *nitg-s* grâce à la particularisation.

5.2 Comparaison des politiques en Nit

Cette partie de l'expérimentation nous permet de déterminer le coût réel d'une politique covariante par rapport à une politique effacée. Ainsi, nous comparons ces deux politiques et leurs implémentations dans les compilateurs NIT.

Comparer objectivement des politiques de typage n'est pas une tâche facile puisque chaque politique apporte ses propres règles d'utilisation des types génériques. En effet, changer la politique de typage d'un langage implique de gros effets de bord car cela change la sémantique du langage. Par exemple, des *casts* implicites peuvent être retirés ou ajoutés à différents endroits pour assurer un typage sûr et ainsi briser le code déjà existant. La sémantique du test de type lui même peut aussi varier selon la politique et bouleverser le fonctionnement d'un programme conforme à une autre politique.

La spécification de NIT précise que la politique de typage dynamique appliquée par le langage est covariante. Cependant, le compilateur principal, `nitc`, implémente la généricité selon une politique effacée non sûre, c'est à dire sans *casts* d'effacement pour assurer la cohérence des types. Le second compilateur, `nitg`, implémente la politique de typage covariante comme indiqué par la spécification mais seulement sur un sous-ensemble du langage.

En conséquence de ces deux moteurs, la plupart des programmes écrits en NIT se comportent exactement de la même façon dans les politiques invariantes et effacées. Dans ces circonstances, NIT représente un excellent cadre d'étude pour comparer les politiques covariante et effacée.

5.2.1 Compilateurs Nit

Pour cette expérimentation, nous dérivons un nouveau compilateur à partir de `nitg-s`. Ce compilateur nommé `nitg-e` applique une politique d'effacement des types génériques. Lors de la compilation d'un module, les paramètres et arguments de types génériques sont effacés par `nitg-e`. Les types virtuels sont quant à eux conservés selon une politique covariante. L'activation du compilateur effacé est faite à l'aide de l'option `-erasure`.

En plus de `nitg-s` et `nitg-e`, nous avons développé deux autres variations du compilateur nommées `nitg-su` et `nitg-eu` dans lesquelles les vérifications de types effectuées à l'exécution sont désactivées.

Le premier correspond à `nitg-s` avec l'option additionnelle `-no-check-covariance`. Si cette option est activée, le compilateur produit un programme possiblement non-sûr dans lequel les tests de covariance ne sont pas ajoutés lors de la compilation. `nitg-su` permet de mesurer le sur-coût engendré par les tests de covariance.

Le second correspond à `nitg-e` avec les options additionnelles `-no-check-covariance` et `-no-check-erasure-cast`. Ces deux options permettent de désactiver à la fois les tests de covariance et les tests d'effacement. Le programme produit est alors lui aussi

possiblement non-sûr et peut se comporter de manière indéterminée. `nitg-eu` permet de mesurer le sur-coût engendré par les tests de covariance et les tests d'effacement.

De plus, `nitg-su` peut être considéré comme la plus proche approximation d'un compilateur appliquant une politique sûre sur des programmes NIT. *Proche* car toute l'information générique est conservée à l'exécution (pas d'effacement) et qu'aucun test de type implicite n'est ajouté par le compilateur. Une *approximation* car les règles de covariance s'appliquent toujours au test de sous-typage : (i) la sémantique des tests de sous-typage et des *casts* ne peut être changée sinon les programmes NIT ne se comporteraient plus correctement ; (ii) la taille de la relation de sous-typage est plus grande qu'en invariance, par exemple, des listes de bananes restent des listes de fruits ; (iii) les programmes existants gardent leur expressivité et les adapter à la politique sûre impliquerait une réécriture complexe du code, l'implémentation d'un bon nombre de solutions de contournement et l'ajout de nombreux *casts* explicites.

5.2.2 Corpus de test

Le corpus de test consiste en cinq différents programmes : `nitg`, `nit`, `shoot`, `bintrees` and `pep8analysis`. La plupart de ces programmes sont disponibles dans le dépôt du projet NIT.

nitg est le compilateur utilisé pour les implémentations des politiques effacées et covariantes. Ce programme est compilé une fois mais exécuté de deux manières différentes.

La première, `nitg nit_metrics.nit`, lance une compilation globale du programme `nit_metrics` (un programme qui calcule des métriques à partir du code source de programmes NIT). La seconde, `nitg -separate nitg.nit`, est une compilation séparée de `nitg` lui-même utilisant une politique covariante et une implémentation homogène.

Note : les deux exécutions partagent une même partie du moteur (*parsing*, construction du modèle, vérification sémantique), mais les traitements communs nécessitent moins de 15% du temps total de compilation. Le programme est le même mais les deux exécutions sont différentes.

nit est un interpréteur naïf pour le langage exécuté avec `nit - test_parser.nit -n rapid_type_analysis.nit`. Le premier argument, `test_parser.nit`, est un petit programme de test qui analyse un fichier source NIT puis construit et affiche un AST. Le fichier source utilisé ici est un module NIT implémentant l'algorithme RTA. Note : `-n` est utilisé pour `test_parser` et désactive l'affichage.

shoot est un jeu minimaliste de type *shoot'em up* avec une logique de jeux orientée objet dans un environnement 2D. Ce programme utilise énormément l'arithmétique à virgule flottante et des collections imbriquées pour stocker les éléments de jeu. La version utilisée est exécutée sans affichage et n'a pas de limitation de *frame rate*. Un total de 300.000 *frames* sont calculés.

bintrees est une adaptation simpliste de GCBench [1] par Hans Boehm, à la différence que les arbres sont représentés par des classes génériques.

pep8analyzer [2] est un analyseur statique qui détecte les erreurs et les bugs dans des programmes PEP/8 [3]. Les arguments passés à l'analyseur sont un ensemble de programmes écrits par des étudiants d'un cours d'assembleur.

5.2.3 Impacts de la politique générique

Nous nous intéressons d'abord à l'impact de la politique générique choisie sur le nombre de tests de sous-typage requis à l'exécution. Pour chaque programme du corpus, nous comptons d'abord le nombre de tests compilés en fonction de la politique, incluant tests implicites et explicites. Puis nous exécutons le programme compilé et comptons le nombre et la nature des tests de sous-typage réalisés pendant l'exécution. La figure 5.7

---

1. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/applet/GCBench.java`

2. `http://github.com/xymus/pep8analysis`

3. PEP/8 est un processeur CISC virtuel conçu pour enseigner l'architecture des ordinateurs et les principes de la programmation en assembleur. `https://code.google.com/p/pep8-1/`

donne le nombre de tests et de *casts* effectués à l'exécution de chaque programme du corpus.

Avec `nitg-s`, une importante proportion des tests sont générés par le compilateur pour assurer la cohésion de types dans les appels génériques covariants. Puisque la spécification de NIT n'autorise pas la covariance dans les paramètres des méthodes, il s'agit de tests de sous-typage contre des types ouverts.

`nitg-e` nécessite plus de tests de sous-typage que `nitg-s` à cause des *casts* d'effacement. Les programmes faisant plus de lectures que d'écritures dans les collections, ce résultat était attendu. Seuls quelques tests contre des types ouverts sont requis, ces tests sont présents uniquement pour les types virtuels, tous les autres types étant effacés.

Les métriques pour `nitg-su` et `nitg-eu` peuvent être dérivés des résultats en retirant tous les tests implicites et en conservant seulement les tests explicites.

Le protocole expérimental utilisé est le même que pour les micro-benchmarks (Section 5.1.2). Comme montré en figure 5.8, sans considération de la politique générique, les performances sont comparables entre tous les moteurs. `nitg-e` donne de meilleures performances que `nitg-s` dans presque tous les cas, mais seulement avec un maximum de 5.8% pour `nit`. Ce comportement peut être expliqué parce que la majorité des tests de sous-typage de `nitg-e` sont faits contre des types résolus. Cependant, quand les *casts* d'effacement sont plus nombreux comme dans `shoot`, `nitg-e` est 9,5% plus lent que `nitg-s`.

Pour les versions non-sûres de `nitg-e` et `nitg-s`, nous pouvons conclure que le coût des tests ajoutés ne représente pas un sur-coût important. Pour `nitg-su` cela va de 1.4% dans `shoot` à 9.1% dans `bintrees`. Puisque ce dernier est le moins réaliste des programmes du corpus, cela n'est pas préoccupant.

	`nitg`		`nitg-s`		`nit`	
	rés.	ouvert	rés.	ouvert	rés.	ouvert
explicite	153M	10M	269M	15M	54M	14M
covariant	0	149M	0	201M	0	156M
explicite	159M	0.7M	280M	0.3M	64M	3.1M
covariant	66M	0.08M	70M	0.1M	53M	0.08M
effacement	956M	0.4M	131M	0.3M	140M	0.06M
	shoot		bintree		pep8analysis	
	rés.	ouvert	rés.	ouvert	rés.	ouvert
explicite	38M	0	143	0	4.5M	17M
covariant	0	5.8M	0	10M	0	98M
explicite	38M	0	143	0	14M	379
covariant	5.4M	0	70M	40M	53M	0.4M
effacement	134M	0	50	0	220M	0

Figure 5.7 Nombre de tests exécutés. Les deux premières lignes correspondent à `nitg-s` et les trois dernières à `nitg-e`. *explicite* correspond au nombre de tests de sous-typage et *casts* explicites requis par le développeur ; *covariance* représente les *casts* introduits par le compilateur pour assurer la sécurité des types dans les appels génériques covariants ; *effacement* correspond aux *casts* ajoutés par le processus d'effacement ; *rés.* correspond aux tests de sous-typage contre des tests résolus tels que `List<Integer>` ou `Map<String, Integer>` ; *ouvert* correspond aux tests contre des types ouverts tels que `List<E>` or `E`.

	nitg-s	nitg-e	nitg-su	nitg-eu
nitg	9.63	9.19	9.09	8.87
nitg-s	7.89	7.49	7.30	7.05
nit	6.87	6.47	6.45	5.84
shoot	7.80	8.55	7.69	7.78
bintree	11.57	11.47	10.51	10.48
pep8analysis	10.97	11.01	10.74	10.71

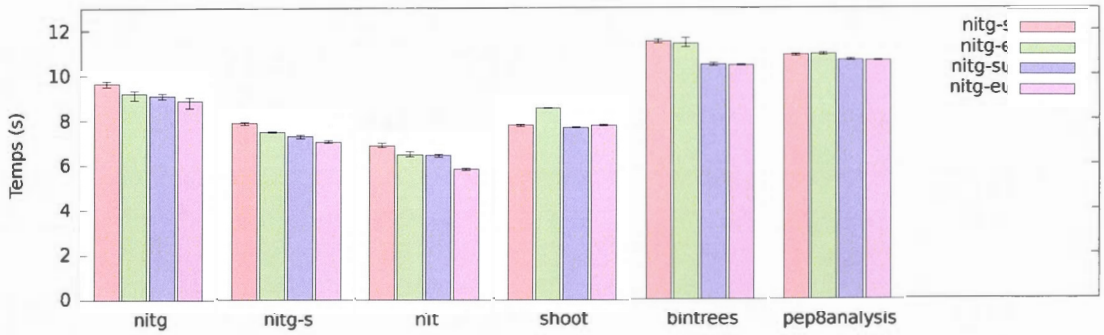

Figure 5.8 Temps d'exécution moyens des compilateurs NIT sur de vrais programmes en fonction des compilateurs et de la politique appliquée. Les temps sont donnés en secondes.

	nitg	nit	shoot	bintree	pep8analysis
CL	24.92%	14.24%	4.88%	10.45%	16.39%
BM	99.42%	99.32%	87.7%	77.79%	96.78%
PH	83.68%	75.78%	72.62%	56.31%	67.78%

Figure 5.9 Comparaison du taux de trous en fonction des implémentations des mécanismes de sous-typage et de résolution : coloration (CL), matrices binaires (BM), hachage parfait (PH). Les valeurs correspondent au pourcentage de trous (cases vides) cumulé dans les tables sous-typage et de résolution.

5.3 Comparaison des techniques d'implémentation

Nous réutilisons le protocole d'expérimentation de la section précédente pour évaluer les techniques d'implémentation. Ainsi, nous comparons les matrices binaires, la coloration et le hachage parfait (opérateur ET bit à bit) sur notre corpus de programmes NIT. Ces trois techniques d'implémentation sont appliquées sur les tests de sous-typage et la résolution des types ouverts. L'envoi de message et l'accès aux attributs restent basés sur la coloration.

La figure 5.9 présente le taux de trous[4] dans les tables de sous-typage et de résolution en fonction de l'implémentation choisie. La coloration, en accord avec nos espérances, offre la meilleure compression des tables avec une moyenne de 14% de trous. Le hachage parfait donne des résultats discutables avec une moyenne de 71% de trous. Mais il reste meilleur que l'approche par matrice binaire avec une moyenne de 92% de trous dans les tables.

La figure 5.10 donne les performances obtenues sur les programmes du corpus pour `nitg-s`. Les autres moteurs donnent des performances comparables. Les résultats sont cohérents avec nos espérances. L'implémentation par matrice binaire est légèrement plus lente à cause des effets de cache du processeur, les tables étant très grandes. Le hachage parfait coûte un peu plus en temps (de 1% dans `shoot` à 4% dans `nitg-s`). La coloration reste la solution la plus performante dans un contexte d'édition de liens globale.

4. Un trou représente une case vide dans la table et donc un espace alloué inutilement.

	CL	BM	PH
nitg	9.55	9.77	9.78
nitg-s	7.81	7.95	8.24
nit	6.79	7.14	7.18
shoot	8.04	8.10	8.12
bintree	11.32	11.43	11.99
pep8analysis	10.77	10.59	11.04

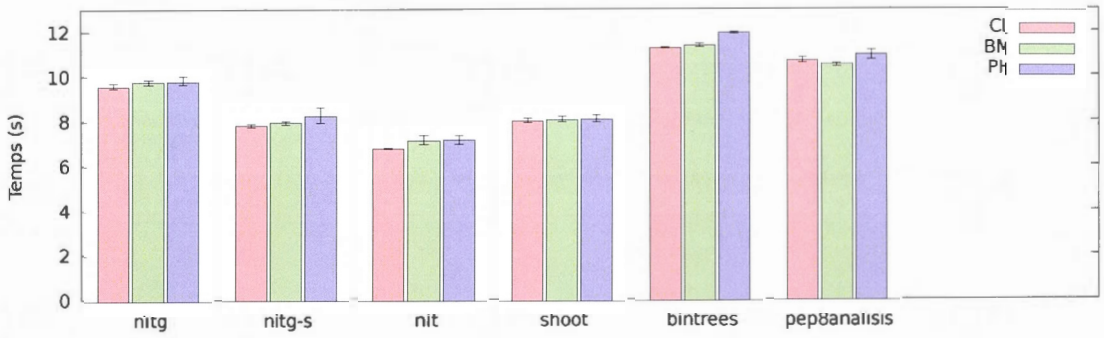

Figure 5.10 Temps d'exécution des compilateurs NIT sur de vrais programmes en fonction de la méthode d'implémentation du mécanisme de sous-typage : coloration (CL), matrices binaires (BM), hachage parfait (PH). Les temps sont donnés en secondes.

5.4 Conclusion

Ce chapitre présente la validation de notre solution telle qu'implémentée dans le compilateur `nitg-s`. Les expérimentations sont menées dans un premier temps sur des programmes générés forçant l'usage de la généricité, puis dans un second temps sur de vrais programmes NIT.

La première partie de l'expérimentation compare les performances de `nitg-s` par rapport à d'autres solutions et implémentations existantes. Ainsi, des micro-benchmarks sont effectués sur des moteurs d'exécution pour les langages C++, C#, EIFFEL, JAVA, SCALA et NIT grâce à des programmes artificiels et non-réalistes. Les résultats confirment que le compilateur séparé et homogène `nitg-s` ainsi que le compilateur global et hétérogène `nitg` sont constants en temps. `nitg` et `nitg-s` sont les plus performants. De plus, les résultats montrent que les implémentations du test de sous-typage basées sur des tables ou des *switches* sont plus rapides que celles basées sur des appels de fonctions.

La seconde partie permet de déterminer les performances de notre solution sur un corpus de vrais programmes NIT. Cette expérimentation permet une comparaison entre `nitg-s` et le compilateur effacé `nitg-e` afin de mettre en évidence le réel coût de la covariance. `nitg-e` donne de meilleurs résultats que `nitg-s` dans presque tous les cas, mais seulement avec un maximum de 5.8%. Le coût de la covariance paraît donc négligeable.

Enfin, le même corpus d'applications NIT est utilisé pour comparer les techniques d'implémentation du test de sous-typage par matrice binaire, coloration ou hachage parfait. Les résultats montrent que notre solution est compatible avec ces trois implémentations. La coloration reste le meilleur choix pour le compilateur `nitg-s` et son schéma de compilation séparée avec édition de liens globale.

CONCLUSION

L'objectif de cette maîtrise est d'implémenter efficacement la généricité covariante et non-effacée dans le langage de programmation NIT. Les implémentations existantes étant soit effacées, soit incomplètes, soit complètement naïves et inefficaces. Une solution permettant d'atteindre cet objectif a été décrite dans le chapitre 4 et les performances de cette solution ont été discutées dans le chapitre 5.

La réalisation principale de notre étude consistait en l'implémentation du compilateur séparé `nitg-s`. Ce compilateur respecte la spécification covariante de la généricité du langage NIT en suivant une implémentation homogène. Ce compilateur est aussi compatible avec les spécificités du langage impactant la généricité telles que l'héritage multiple, les types virtuels et les types nullables.

Nous avons profité de la réalisation du nouveau compilateur pour en écrire un second, `nitg-e`, suivant une spécification effacée et deux autres versions non-sûres, `nitg-su` et `nitg-eu` simulant des spécifications statiquement vérifiées de la généricité.

L'ensemble de ces quatre nouveaux compilateurs nous a permis de comparer l'impact des différentes politiques de typage générique sur les performances des applications compilées. Comparer des politiques est un problème difficile à résoudre car changer la politique d'un langage revient à changer son comportement. Il faut alors trouver des programmes qui se comportent de la même manière quelle que soit la politique. Ainsi, nous utilisons ces quatre compilateurs pour comparer les performances de vrais programmes NIT selon les politiques utilisées.

Les compilateurs `nitg-s` et `nitg-e` nous ont d'abord permis de comparer les politiques covariante et effacée. Nous avons montré que la politique covariante ne représente qu'un

sur-coût négligeable de performances comparé au gain qu'elle représente en termes d'expressivité et d'intuitivité par rapport à la politique effacée.

Les deux versions non-sures de ces compilateurs, `nitg-su` et `nitg-eu`, nous ont permis de constater que le coût d'une généricité covariante vérifiée à l'exécution est négligeable comparé à celle vérifiée à la compilation. Il est ainsi possible de se passer de la syntaxe verbeuse utilisée par certains langages pour assurer un typage statique sûr. Ces deux résultats représentent notre principale contribution au domaine.

L'implémentation du compilateur en lui même nous a elle aussi amené à relever quelques défis intéressants. Tout d'abord l'implémentation homogène de la généricité covariante a nécessité un mécanisme objet additionnel que nous avons nommé le mécanisme de résolution des types ouverts. L'implémentation de ce mécanisme de résolution nous a amené à étudier les implémentations existantes de la généricité dans les moteurs EIFFELSTUDIO et MONO C#. Par manque de documentation sur l'implémentation de la généricité dans ces moteurs, nous avons dû les analyser par rétro-ingénierie de leur code source. L'étude de ces deux implémentations représente une autre contribution de notre étude.

Ne trouvant aucune solution efficace et compatible avec l'héritage multiple dans les moteurs existants, nous avons proposé notre propre implémentation homogène de la généricité. Inspirée par l'implémentation de MONO C#, notre solution utilise des tables stockées en mémoire que nous nommons tables de résolution. Ces tables permettent d'assurer une résolution des types ouverts et un test de sous-typage en temps constant compatible avec l'héritage multiple, la généricité covariante, les types virtuels et les types nullables. La construction des tables de résolution peut être faite en utilisant les implémentations traditionnelles des mécanismes objets en héritage multiple telles que les matrices binaires, la coloration ou le hachage parfait. Grâce à ces trois techniques, notre solution est compatible avec les schémas de compilation séparés et globaux.

Une fois le compilateur implémenté, il nous fallait vérifier l'efficacité de notre solution, le problème étant de s'assurer que les performances de notre solution était dans le même

ordre de grandeur que celles des autres solutions existantes. Mais comment comparer l'implémentation de la généricité entre différents langages, tous les autres mécanismes objets étant différents ? Pour répondre à cette question, nous avons créé un protocole d'évaluation capable de comparer les performances de différents moteurs d'exécution sur des programmes artificiels utilisant la généricité. Ces programmes utilisent au maximum les mécanismes de sous-typage des moteurs pour en éprouver les limites.

Le protocole d'évaluation mis en place pour réaliser cette comparaison représente une autre contribution de notre étude. Il nous a permis de comparer les performances de notre implémentation contre des moteurs d'exécution pour les langages C++, C#, EIFFEL, JAVA, SCALA et NIT. Nous avons ainsi été capable de démontrer que notre solution reste constante en temps même dans les cas les plus poussés et qu'elle offre de meilleures performances que la majorité des moteurs.

Finalement, à la lumière de cette étude, nous considérons qu'il est possible de fournir une implémentation homogène efficace de la généricité covariante en héritage multiple. Le sur-coût engendré par la covariance est négligeable comparé aux gains en termes de simplicité et d'expressivité du code.

Notre principale préoccupation pour l'avenir concerne la question du chargement dynamique dans une machine virtuelle. Bien qu'il n'apparaisse pas d'incompatibilité notable entre notre proposition et le chargement dynamique, notre implémentation se base actuellement sur un schéma de compilation séparé avec édition de liens globale et un algorithme RTA. Ainsi, nos possibles avenues de recherche peuvent impliquer un algorithme RTA incrémental pendant le chargement dynamique ou une évaluation paresseuse avec mise en cache.

Un autre problème soulevé par le chargement dynamique est l'impact de l'apparition de nouveaux types dans le système. Là où les autres politiques assurent que la hiérarchie des types ne sera jamais modifiée autrement que par l'ajout de sous-types aux types déjà chargés, la politique covariante permet l'apparition de nouveaux super-types et entraîne donc le recalcul de l'ensemble de la hiérarchie.

BIBLIOGRAPHIE

(Agesen, Freund et Mitchell, 1997) Agesen, O., S. Freund et J. Mitchell. 1997. « Adding type parameterization to the Java language ». In *Proceedings of the 12th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, p. 49–65, Atlanta, Georgia, USA. ACM.

(Alpern, Cocchi et Grove, 2001) Alpern, B., A. Cocchi et D. Grove. 2001. « Dynamic type checking in jalapeno ». In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM'01)*, p. 4–16, Monterey, California. USENIX Association.

(Altidor, Reichenbach et Smaragdakis, 2012) Altidor, J., C. Reichenbach et Y. Smaragdakis. 2012. « Java wildcards meet definition-site variance ». In *Proceedings of the 26th European conference on Object-Oriented Programming (ECOOP'12)*, p. 509–534, Beijing, China. Springer-Verlag.

(Amiel, Gruber et Simon, 1994) Amiel, E., O. Gruber et E. Simon. 1994. « Optimizing multi-method dispatch using compressed dispatch tables ». In *Proceedings of the 9th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'94)*, p. 244–258, Portland, Oregon, USA. ACM.

(Bacon et Sweeny, 1996) Bacon, D. F., et P. F. Sweeny. 1996. « Fast static analysis of C++ virtual function calls ». In *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, p. 324–341, San Jose, California, USA. ACM.

(Bank, Myers et Liskov, 1997) Bank, J. A., A. C. Myers et B. Liskov. 1997. « Parameterized types for Java ». In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '97)*, p. 132–145, Paris, France. ACM.

(Boehm et Spertus, 2009) Boehm, H.-J., et M. Spertus. 2009. « Garbage collection in the next C++ standard ». In *Proceedings of the 2009 International Symposium on Memory Management (ISMM'09)*, p. 30–38, Dublin, Ireland. ACM.

(Bracha et al., 1998) Bracha, G., M. Odersky, D. Stoutamire et P. Wadler. 1998. « Making the future safe for the past : adding genericity to the Java programming language ». In *Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, p. 183–200, Vancouver, British Columbia, Canada. ACM.

(Cardelli et Wegner, 1985) Cardelli, L., et P. Wegner. 1985. « On understanding types, data abstraction, and polymorphism ». *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, p. 471–523.

(Cartwright, Steele et L., 1998) Cartwright, R., J. Steele et G. L. 1998. « Compatible genericity with run-time types for the Java programming language ». In *Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, p. 201–215, Vancouver, British Columbia, Canada. ACM.

(Cohen, 1991) Cohen, N. H. 1991. « Type-extension type test can be performed in constant time ». *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, p. 626–629.

(Day et al., 1995) Day, M., R. Gruber, B. Liskov et A. C. Myers. 1995. « Subtypes vs. Where clauses : Constraining parametric polymorphism ». In *Proceedings of the 10th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*, p. 156–168, Austin, Texas, United States. ACM.

(Dragos et Odersky, 2009) Dragos, I., et M. Odersky. 2009. « Compiling generics through user-directed type specialization ». In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS'09)*, p. 42–47, Genova, Italy. ACM.

(Ducournau, 2002) Ducournau, R. 2002. « "Real World" as an argument for covariant specialization in programming and modeling ». In *Proceedings of the Workshops on Advances in Object-Oriented Information Systems (OOIS'02)*, p. 3–12. Springer-Verlag.

(Ducournau, 2011a) ———. 2011a. « Coloring, a versatile technique for implementing object-oriented languages ». *Software : Practice and Experience*, vol. 41, no. 6, p. 627–659.

(Ducournau, 2011b) ———. 2011b. « Implementing statically typed object-oriented programming languages ». *ACM Computing Surveys*, vol. 42, no. 3, p. 1–48.

(Ducournau et Morandat, 2011) Ducournau, R., et F. Morandat. 2011. « Perfect class hashing and numbering for object-oriented implementation ». *Software : Practice and Experience*, vol. 41, no. 6, p. 661–694.

(Ducournau, Morandat et Privat, 2009) Ducournau, R., F. Morandat et J. Privat. 2009. « Empirical assessment of object-oriented implementations with multiple inheritance and static typing ». In *Proceedings of the 24th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, p. 41–60, Orlando, Florida, USA. ACM.

(Ellis et Stroustrup, 1990) Ellis, M. A., et B. Stroustrup. 1990. *The annotated C++ reference manual.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

(Gagnon et Hendren, 2001) Gagnon, E. M., et L. J. Hendren. 2001. « SableVM : A research framework for the efficient execution of Java bytecode ». In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium (JVM'01)*, p. 3–3, Monterey, California. USENIX Association.

(Garcia et al., 2007) Garcia, R., J. J. Arvi, A. Lumsdaine, J. Siek et J. Willcock. 2007. « An extended comparative study of language support for generic programming ». *Journal of Functional Programming*, vol. 17, no. 2, p. 145–205.

(Gélinas, Gagnon et Privat, 2009) Gélinas, J., E. Gagnon et J. Privat. 2009. « Prévention de déréférencement de références nulles dans un langage à objets ». *Langages et Modèles à Objets*, vol. L-3, p. 5–16.

(Igarashi et Viroli, 2006) Igarashi, A., et M. Viroli. 2006. « On variance-based subtyping for parametric types ». In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, p. 441–469. Springer-Verlag.

(ISO/IEC, 2007) ISO/IEC. 2007. C++11 Technical Report 1 (TR1) - ISO/IEC TR 19768, C++ Library Extensions.

(Kennedy et Syme, 2001) Kennedy, A., et D. Syme. 2001. « Design and implementation of generics for the .NET Common Language Runtime ». In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Coll. « PLDI '01 », p. 1–12, Snowbird, Utah, USA. ACM.

(Krall et Grafl, 1997) Krall, A., et R. Grafl. 1997. « CACAO - A 64-bit JavaVM just-in-time compiler ». *Concurrency : Practice and Experience*, vol. 9, no. 11, p. 1017–1030.

(Liskov et Wing, 1994) Liskov, B., et J. Wing. 1994. « A behavioral notion of subtyping ». *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, p. 1811–1841.

(MacQueen, 1984) MacQueen, D. 1984. « Modules for standard ML ». In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP'84)*, p. 198–207, Austin, Texas, USA. ACM.

(Morandat, 2010) Morandat, F. 2010. « Contribution à l'efficacité de la programmation par objets. Evaluation des implémentations de l'héritage multiple en typage statique. ». Thèse de Doctorat, Université de Montpellier.

(Musser et Stepanov, 1987) Musser, D. R., et A. A. Stepanov. 1987. « A library of generic algorithms in Ada ». In *Proceedings of the 1987 annual ACM SIGAda international conference on Ada (SIGAda'87)*, p. 216–225, Boston, Massachusetts, USA. ACM.

(Odersky et Wadler, 1997) Odersky, M., et P. Wadler. 1997. « Pizza into Java : translating theory into practice ». In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '97)*, p. 146–159, Paris, France. ACM.

(Palacz et Vitek, 2003) Palacz, K., et J. Vitek. 2003. « Java subtype tests in real-time ». In *Proceedings of the 15th European conference on Object-Oriented Programming (ECOOP'03)*, p. 378–404. Springer-Verlag.

(Privat, 2006) Privat, J. 2006. « De l'expressivité à l'efficacité, une approche modulaire des langages à objets. Le langage PRM et le compilateur prmc ». Thèse de Doctorat, Université de Montpellier II.

(Privat et Ducournau, 2005) Privat, J., et R. Ducournau. 2005. « Raffinement de classes dans les langages à objets statiquement typés ». *Langages et Modèles à Objets (LMO'05)*, vol. 11, no. 1-2, p. 17–32.

(Privat et Morandat, 2008) Privat, J., et F. Morandat. 2008. « Coloring for shared object-oriented libraries ». In *Proceedings of the 3th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming*

Systems (ICOOOLPS'08), p. 8, Nancy, France. ACM.

(Proebsting et Townsend, 1997) Proebsting, T., et G. Townsend. 1997. « Toba : Java for applications : A way ahead of time (WAT) compiler ». In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS'97)*, p. 41–54, Portland, Oregon. USENIX Association.

(Sallenave et Ducournau, 2012) Sallenave, O., et R. Ducournau. 2012. « Lightweight generics in embedded systems through static analysis ». *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'12)*, p. 11–20.

(Torgersen, 1998) Torgersen, M. 1998. « Virtual types are statically safe ». In *5th Workshop on Foundations of Object-Oriented Languages*, p. 1–9, San Diego, California, USA. K. Bruce.

(Viroli et Natali, 2000) Viroli, M., et A. Natali. 2000. « Parametric polymorphism in Java : an approach to translation based on reflective features ». In *Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, p. 146–165, Minneapolis, Minnesota, USA. ACM.

(Vitek, Horspool et Krall, 1997) Vitek, J., R. N. Horspool et A. Krall. 1997. « Efficient type inclusion tests ». In *Proceedings of the 12th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, p. 142–157, Atlanta, Georgia, USA. ACM.

(Zendra, Colnet et Collin, 1997) Zendra, O., D. Colnet et S. Collin. 1997. « Efficient dynamic dispatch without virtual function tables : The SmallEiffel compiler ». In *Proceedings of the 12th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, p. 125–141, Atlanta, Georgia, USA. ACM.