Production de code, langages à héritage multiple

Martin Odersky

23 janvier 2007
version 1.2

Plan du cours

1. Héritage multiple
2. Techniques de dispatching
   - Trampolines
   - Tableaux de déplacement de lignes
   - Améliorer les performances
3. Les compilateurs JIT

Héritage multiple

Le schéma de *dispatching* par table de méthodes virtuelles (VMT) est prédominant dans les situations d'héritage simple :

- Simula, Modula-3, Ada 95, Object Oberon, Beta...

Mais la plupart des langages OO sont plus complexes.

Langages avec héritage multiple ou *mixins* :

- Scala, Eiffel, gbeta, C++...

Langages avec sous-typage structurel :

- Smalltalk, Cecil, Self, Pict...

Langages hybrides avec héritage simple et interfaces :

- Java, Objective-C...

Plusieurs techniques existent pour implanter l'héritage multiple, les *mixins* ou l'héritage hybride :

- trampolines,
- tableaux de déplacement de lignes (*row-displacement tables*),
- antémémoire en ligne (*inline caching*).

Exemple : Des classes en Drei à héritage multiple

```
class Point {
  val x: Int;
  def position(): Int = {
    return this.x
  }
  def copy(delta: Int): Point = {
    return new Point(this.position() + delta);
  }
}
class Colored {
  val c: Color;
  def color(): Color = {
    return this.c
  }
}
class ColoredPoint extends Colored, Point {
  def copy(delta: Int): ColoredPoint = {
    return new ColoredPoint(color(), this.position() + delta);
  }
}
```


Trampolines

L'idée est d'avoir des points d'entrée multiples pour les références, un par classe de base.

- Chaque point d'entrée a un champ entête (*header*) qui pointe vers une table de méthodes virtuelles.
- Quand on passe d'une sous-classe à une super-classe on met à jour le pointeur de l'objet pour qu'il pointe vers le point d'entrée correct.
- La redéfinition d'une méthode rend nécessaire de se déplacer d'un point d'entrée au début de l'objet englobant.

Cela est réalisé par une méthode trampoline qui, une fois appelée, retourne la référence de l'objet englobant en soustrayant une valeur connue du point d'entrée.

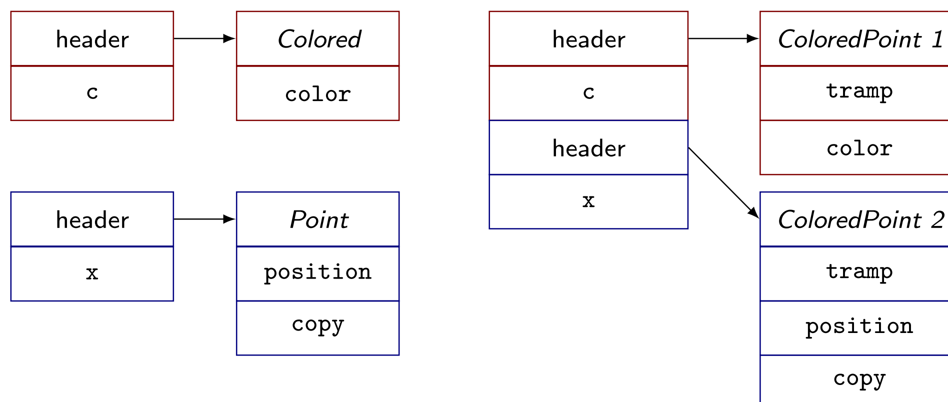
Cette technique a été utilisée pour gbeta et C++.

Avantages de la technique par trampolines :

- Des performances raisonnables même dans le pire des cas.
- Les champs et les méthodes peuvent être hérités de façon multiple.

Désavantages :

- Surcoût des méthodes trampoline.
- les structures de données covariantes ne sont pas supportées : `ColoredPoint[] <: Point[]` ne peut pas fonctionner car il faudrait alors mettre à jour chaque pointeur dans le tableau.



Les méthodes trampolines sont, pour *ColoredPoint 1* : `tramp(p) = p`, pour *ColoredPoint 2* : `tramp(p) = p - 8`.

Pour deux variables `p: Point` et `cp: ColoredPoint` on a :

- `p = cp` devient `p = cp + 8`,
- `cp = p` devient `cp = p.tramp(p)`.

Tableaux de déplacement de lignes

Une autre technique de dispatching est le tableau de déplacement de lignes (*row-displacement table*).

Le problème du dispatching dynamique est le suivant :

- Étant donné un ensemble de classes et de méthodes, trouver le code correspondant à une classe et à une méthode données.

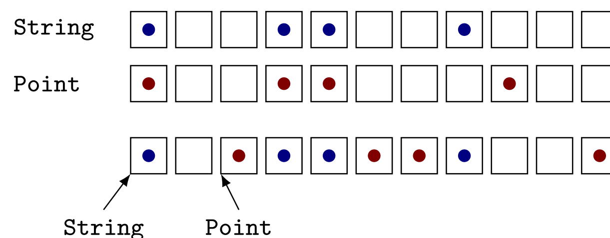
Si l'on énumère les classes et les méthodes, cette tâche se réduit à une opération d'indexage dans un tableau bi-dimensionnel :

	`equals`	`append`	`main`
`Object`	•		
`String`	•	•	
`Main`	•		•

Le tableau de déplacement de lignes devient vite énorme :

Une application de 500 classes et 2'000 noms de méthodes uniques crée une table de 1'000'000 d'entrées.

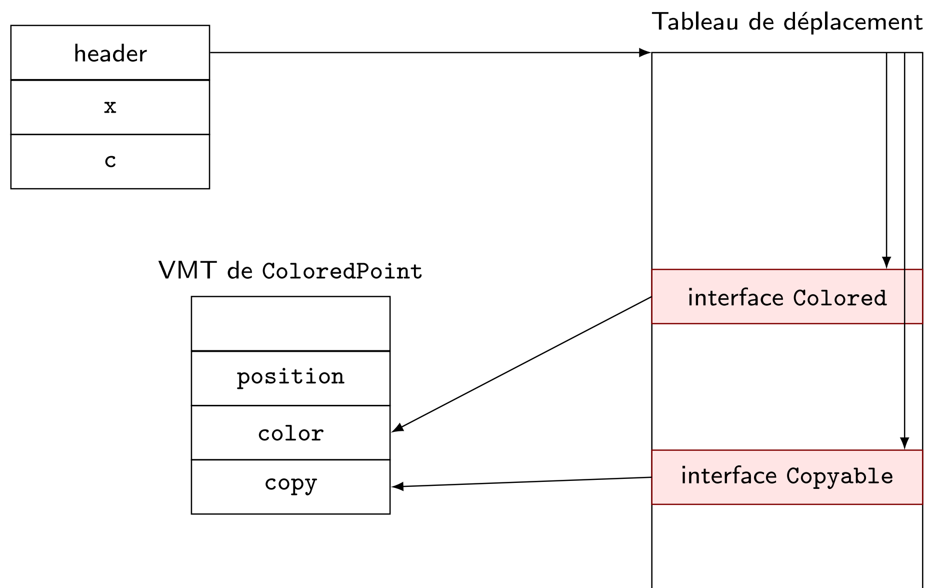
- Ce tableau à deux dimensions est occupé de façon clairsemée car chaque classe n'implante qu'un petit sous-ensemble de toutes les méthodes.
- On peut obtenir une meilleure utilisation de l'espace en imbriquant les lignes successives comme un ensemble de peignes :



En Java, on peut obtenir une bonne utilisation de l'espace de la façon suivante :

- Indexer le tableau avec les classes et les interfaces plutôt qu'avec les classes et les méthodes.
- Une entrée du tableau pointe sur l'endroit dans une VMT où la méthode de l'interface est implantée.

Cette technique a été utilisée dans certaines implantations très rapides de Java.



Le code de dispatching pour l'appel : `obj: I; ... obj.meth()` est `obj.header(I.number).meth()`.

Question : Comment savons-nous que l'entrée du tableau est utilisée pour la classe courante ?

Réponse : Nous n'avons pas besoin de le savoir, car Java est statiquement typé !

L'avantage du *dispatching* avec tableau à lignes (*row table*) est sa bonne performance avec le cas moyen = au pire des cas.

Considérations de pipelining

Le *dispatching* par VMT et celui par tableau à lignes introduisent des bulles dans le pipeline.

- On ne peut aller chercher de nouvelles instructions qu'après avoir calculé l'adresse dynamique de la méthode.
- Dans les processeurs modernes avec des *pipelines* profonds, cela peut s'avérer très coûteux.

Un processeur de type *Pentium 4* utilise un *pipeline* d'une profondeur pouvant aller jusqu'à 31 stages, sur plusieures instructions en parallèle : une bulle dans le *pipeline* coûte très cher.

C'est d'ailleurs encore pire avec des processeurs à mots d'instructions très larges (*VLIW*)

Antémémoires

En réalité, de nombreux appels vont toujours à la même classe.

On peut améliorer les performances du *dispatching* grâce aux antémémoires (*inline caching*) :

- Pour chaque instruction d'appel se rappeler le code qui a été utilisé à la dernière exécution de cette instruction.
- Sauter immédiatement vers ce code sans utiliser le *dispatching* dynamique.
- Au début du code cible, tester si l'on est dans la bonne classe.
- Si ce n'est pas le cas, retourner au schéma de *dispatching* dynamique classique, plus lent.

Ce schéma permet un gain important si les appels vont toujours à la même classe . . . sinon c'est une grosse perte !

Le *inline caching* est utilisé pour implanter les appels aux méthodes d'interface dans HotSpot :

- L'instruction `invoke_interface` a un champ qui contient la position relative de l'entrée de la méthode qui a été invoquée durant la dernière exécution de cette instruction (par rapport au début de la VMT).
- Quand `invoke_interface` est exécutée, il est d'abord vérifié qu'une entrée pour la méthode appelée se trouve bien à la position donnée.
- Sinon on recherche linéairement parmi toutes les méthodes de l'objet donné une méthode qui corresponde au nom et au type de la méthode appelée.

Antémémoires polymorphes

Le *inline caching* est une optimisation «tout ou rien» :

- c'est soit très rapide soit inutile (voir même néfaste).

Un compromis est de garder un tableau des n dernières cibles.

- Si la cible courante est dans le tableau, sauter directement, sinon continuer avec le *dispatching dynamique* et ajouter la nouvelle cible dans le tableau.
- Si le tableau devient grand : retour au dispatching dynamique.
- Ce schéma est décrit dans la thèse de Urs Hölzle.
- Il est utilisé dans les implantations de Self et HotSpot.

Avantage : on évite les bulles dans le pipeline : potentiellement de très bonnes performances, même meilleures que le dispatching simple avec VMT pour l'héritage simple.

Désavantage : imprévisible : peut être (légèrement) pire que le dispatching avec VMT dans les mauvais cas.

Compilateurs JIT

- L'interprétation du *bytecode* Java réduit les performances.
- La distribution des classes Java sous forme de code natif améliorerait les performances, mais au prix de la portabilité et de la sécurité.

Les compilateurs JIT (*just-in-time*) offrent une solution.

- Un compilateur JIT compile le *bytecode* en code natif, soit au chargement, soit après quelques exécutions du code.

En principe le code compilé JIT peut être plus rapide que du code natif compilé statiquement vu qu'il y a plus d'informations disponibles à l'exécution qu'à la compilation :

- quelles méthodes sont appelées le plus souvent ?
- combien de méthodes différentes cet appel invoque-t-il ?

En pratique le code compilé JIT est généralement plus lent que du code natif car :

- le surcoût de la compilation ralentit l'exécution,
- les optimisations des compilateurs JIT doivent aller vite et sont donc moins agressives que les optimisations des compilateurs de code natif.

Il faut trouver un compromis sur :

- compilateur JIT lent et code généré rapide ou compilateur JIT rapide et code généré lent ?
- quand invoquer le compilateur JIT ?
  Symantec le fait à la première exécution, Inprise à la seconde, HotSpot à la 1'000e et HotSpot (serveur) à la 10'000e.