

# Le Hachage Parfait Fait-il un Parfait Test de Sous-typage ?

Roland Ducournau

LIRMM  
CNRS – Université Montpellier II

Langages et Modèles à Objets  
Nîmes 2006



# Plan

- 1 Contexte et Motivation
- 2 Du Hachage à la Perfection
  - Hachage Parfait
  - Application à JAVA
- 3 Expérimentations et évaluation
- 4 Comparaison, Conclusion et Perspective

# Plan

- 1 **Contexte et Motivation**
- 2 Du Hachage à la Perfection
  - Hachage Parfait
  - Application à JAVA
- 3 Expérimentations et évaluation
- 4 Comparaison, Conclusion et Perspective

# Implémentation des Objets

## Mécanismes primitifs

- Appel de méthodes
- Accès aux attributs
- Test de sous-typage

## Exigences d'efficacité et de fonctionnalité

- Temps constant
- Espace linéaire
- *Inlining*
- Héritage multiple
- Compilation modulaire et chargement dynamique

# Héritage simple, typage statique

A

A

# Héritage simple, typage statique

A



B

A

A

B

## Problème : héritage multiple impossible

- Invariants de référence et de position
- Espace linéaire dans la taille de la relation de spécialisation
- Implémentation idéale

# Héritage simple, typage statique

A



B



C

A

A

B

A

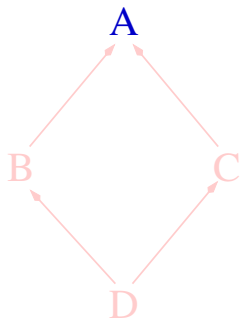
B

C

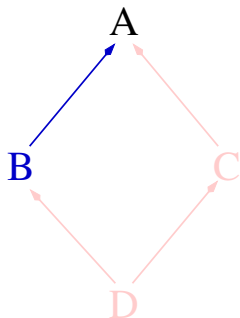
## Problème : héritage multiple impossible

- Invariants de référence et de position
- Espace linéaire dans la taille de la relation de spécialisation
- Implémentation idéale

# Coloration



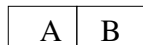
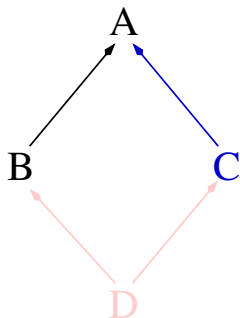
# Coloration



## Problème : chargement dynamique impossible

- La coloration est une optimisation globale
- Invariants maintenus au prix de quelques trous

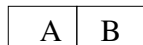
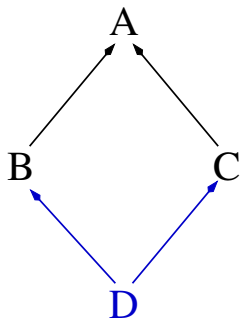
# Coloration



## Problème : chargement dynamique impossible

- La coloration est une optimisation globale
- Invariants maintenus au prix de quelques trous

# Coloration



## Problème : chargement dynamique impossible

- La coloration est une optimisation globale
- Invariants maintenus au prix de quelques trous

# Accès direct dans une grande table

← propriétés / superclasses →

	a	b	c	d	e	f	g	h	i	j	k	l	...
A		x		x	x								
B		x	x				x	x					
C				x									
D							x				x		
E			x					x					
F				x									
...													

↑ classes ↓

# Accès direct dans une grande table

← propriétés / superclasses →

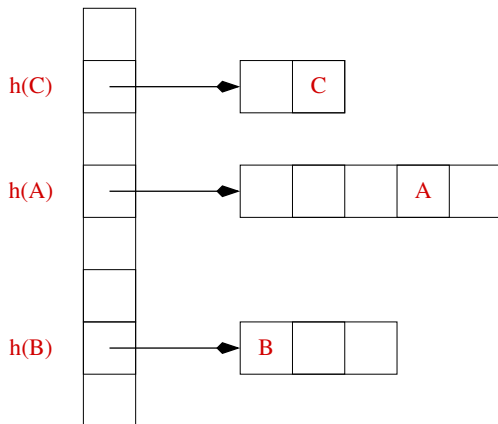
	a	b	c	d	e	f	g	h	i	j	k	l	...
A		x		x	x								
B		x	x				x	x					
C				x									
D							x				x		
E			x					x					
F				x									
...													

↑ classes ↓

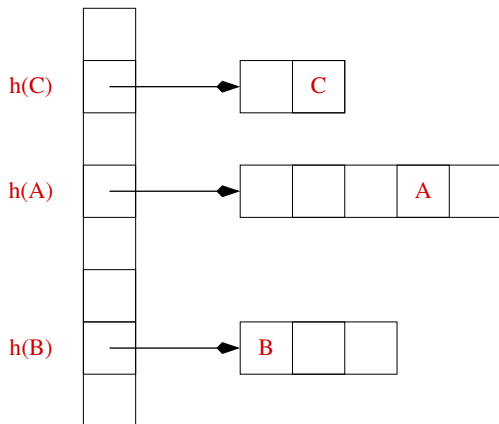
**Problème : espace non linéaire**

Table gigantesque, presque vide

# Tables de hachage



# Tables de hachage



## Problème : temps non constant

- recherche séquentielle à partir de la valeur de hachage
- efficacité spatiale médiocre (au moins  $\times 2$ )

## Question

Mais comment diable font-ils donc avec des tables de hachage ?

# Plan

- 1 Contexte et Motivation
- 2 Du Hachage à la Perfection**
  - Hachage Parfait
  - Application à JAVA
- 3 Expérimentations et évaluation
- 4 Comparaison, Conclusion et Perspective

# Test de sous-typage par table de hachage

## Principe

Soit 2 classes  $C$  et  $D$ , ou une classe  $C$  et un objet  $o$  :

- $D \prec C \iff id_C$  est trouvé dans la table de  $D$
- $o \in C \iff id_C$  est trouvé dans la table de  $o$



# Test de sous-typage par table de hachage

## Principe

Soit 2 classes  $C$  et  $D$ , ou une classe  $C$  et un objet  $o$  :

- $D \prec C \iff id_C$  est trouvé dans la table de  $D$
- $o \in C \iff id_C$  est trouvé dans la table de  $o$

## Particularité

Au chargement de la classe  $D$ , toutes ses super-classes sont connues.

- plus besoin d'ajout, ni de retrait
- $\Rightarrow$  La table de  $D$  peut être optimisée statiquement

# Test de sous-typage par table de hachage

## Principe

Soit 2 classes  $C$  et  $D$ , ou une classe  $C$  et un objet  $o$  :

- $D \prec C \iff id_C$  est trouvé dans la table de  $D$
- $o \in C \iff id_C$  est trouvé dans la table de  $o$

## Particularité

Au chargement de la classe  $D$ , toutes ses super-classes sont connues.

- plus besoin d'ajout, ni de retrait
- $\Rightarrow$  La table de  $D$  peut être optimisée statiquement

## Solution : le “hachage parfait” [Sprugnoli 77]

- hachage sans collision
- temps constant, espace à déterminer

# Hachage parfait

## Principe

- un univers  $E$ ,
- un ensemble  $I \subset E$  à hacher,
- une fonction de hachage  $h : E \rightarrow [0..H - 1]$ ,
- une table  $T$  de taille  $H$ .

$h$  est injective sur  $I \quad \Rightarrow \quad x \in I \text{ ssi } T[h(x)] = x$

## Problème

Etant donné  $I$ , calculer  $h$  et  $H$

# Application au test de sous-typage

## Pour chaque classe $C$

- $E = [0..N - 1]$  ensemble des identifiants de classes
- $I_C = \{id_D \mid C \preceq D\}$
- fonction  $h_C : [0..N - 1] \rightarrow [0..H_C - 1]$  injective sur  $I_C$  avec  $h_C(x) = hash(x, H_C)$

# Application au test de sous-typage

## Pour chaque classe $C$

- $E = [0..N - 1]$  ensemble des identifiants de classes
- $I_C = \{id_D \mid C \preceq D\}$
- fonction  $h_C : [0..N - 1] \rightarrow [0..H_C - 1]$  injective sur  $I_C$  avec  $h_C(x) = hash(x, H_C)$

## Quelle fonction pour *hash* ?

- Exigences : **mono-instruction**, si possible **mono-cycle**
- Deux candidates naturelles :  
reste de division entière (**mod**) et conjonction booléenne (**and**)

# Application au test de sous-typage

## Pour chaque classe $C$

- $E = [0..N - 1]$  ensemble des identifiants de classes
- $I_C = \{id_D \mid C \preceq D\}$
- fonction  $h_C : [0..N - 1] \rightarrow [0..H_C - 1]$  injective sur  $I_C$  avec  $h_C(x) = hash(x, H_C)$

## Quelle fonction pour *hash* ?

- Exigences : **mono-instruction**, si possible **mono-cycle**
- Deux candidates naturelles :  
reste de division entière (**mod**) et conjonction booléenne (**and**)

## Problème : calculer $H_C$ **minimal** tq $h_C$ soit injective ?

- algorithme polynomial facile

# Applications

## Test de sous-typage en C++

- ... ou dans tout langage en héritage multiple “pur”
- et avec chargement dynamique

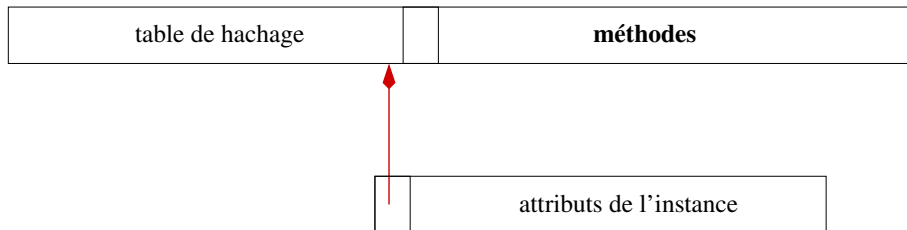
## Langage à la JAVA

... ou tout autre langage en **sous-typage multiple**

2 applications :

- test de sous-typage quand la cible est une **interface**
- appel de méthode quand le receveur est typé par une **interface**

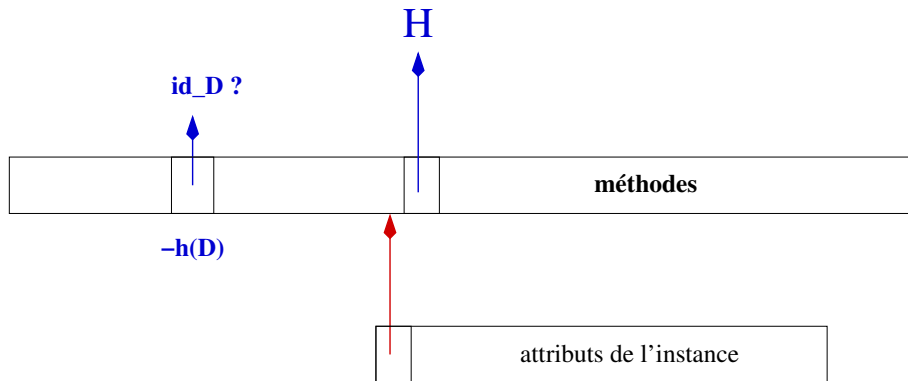
# Implémentation pour JAVA



## Table bidirectionnelle

- Partie positive pour les classes, comme en héritage simple
- Partie négative pour les interfaces, avec hachage parfait

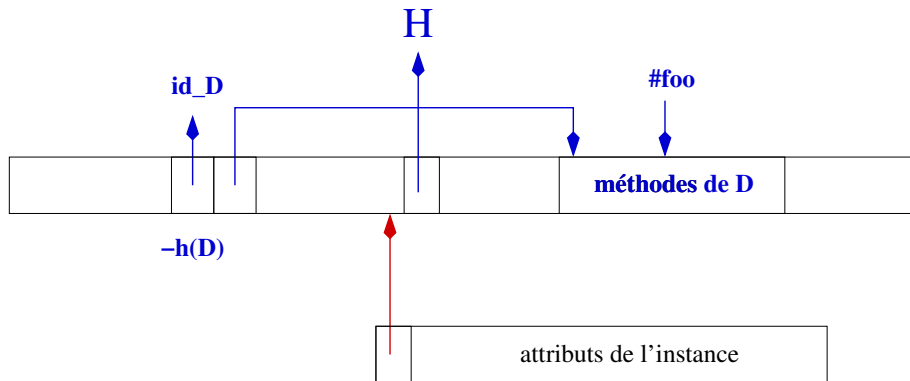
# Implémentation pour JAVA



## Table bidirectionnelle

- Test de sous-typage pour interfaces
- $o \in D?$

# Implémentation pour JAVA



## Table bidirectionnelle

- Appel de méthode pour interfaces (`invokeinterface`)
- `D x ; x.foo(arg)`

# Motivation pour JAVA

## JAVA normal

- utilisation fréquente des *downcasts*  
manque de généricité en 1.4
- et de *invokeinterface*  
suivant le style de programmation / conception

## Langage SCALA

- JAVA + mixins
- compile → JVM ou DOTNET CLR
- les attributs sont médiatisés par des accesseurs
- tous les types sont des interfaces
- ⇒ *invokeinterface* est omniprésent

# Plan

- 1 Contexte et Motivation
- 2 Du Hachage à la Perfection
  - Hachage Parfait
  - Application à JAVA
- 3 Expérimentations et évaluation**
- 4 Comparaison, Conclusion et Perspective

# Description de l'expérimentation

## Benchmarks

- Famille de benchmarks classiques,
- utilisés depuis plus de 10 ans par l'ensemble de la communauté d'implémentation des objets
- 17 hiérarchies, plus de 30000 classes

## Simulation

- Plate forme de simulation en LISP,
- utilisée depuis 2000 pour évaluer toutes les implémentations (C++, coloration, etc.)

# Résultats pour JAVA

## Espace et temps

technique	appel de méthode			test sous-typage			nég pos
	cycles		code	cycles		code	
COL <sub>2</sub>	$2L+B$	16	3	$2L+2$	8	4	€
DA	$3L+B$	19	4	$2L+3$	9	7	4.2

## Latences

- de chargement :  $L = 3$
- de saut indirect :  $B = 10$
- de division entière :  $D = 6$

# Résultats pour JAVA

## Espace et temps

technique	appel de méthode			test sous-typage			nég pos
	cycles	code	code	cycles	code	code	
COL <sub>2</sub>	$2L+B$	16	3	$2L+2$	8	4	€
DA	$3L+B$	19	4	$2L+3$	9	7	4.2
PH and	$4L+B+2$	24	7	$3L+4$	13	7	1.2
PH mod	$4L+B+D+1$	29	7	$3L+D+3$	18	7	0.4

## Latences

- de chargement :  $L = 3$
- de saut indirect :  $B = 10$
- de division entière :  $D = 6$

# Résultats pour JAVA

## Espace et temps

technique	appel de méthode			test sous-typage			nég pos
	cycles		code	cycles		code	
COL <sub>2</sub>	$2L+B$	16	3	$2L+2$	8	4	€
DA	$3L+B+1$	20	5	$2L+3$	9	7	2.1
PH and	$4L+B+3$	25	8	$3L+4$	13	7	0.6
PH mod	$4L+B+D+1$	30	8	$3L+D+3$	18	7	0.2

## Presqu'assez rapide, presqu'assez compact !

- adresses → offsets :  
⇒ 1 cycle et 1 instruction en plus, espace divisé par 2
- $3 \leq D \leq 25$  cycles, suivant processeurs et données

# Résultats pour JAVA

## Espace et temps

technique	appel de méthode			test sous-typage			nég pos
	cycles	code	code	cycles	code	code	
COL <sub>2</sub>	$2L+B$	16	3	$2L+2$	8	4	€
DA	$3L+B+1$	20	5	$2L+3$	9	7	2.1
PH and	$4L+B+3$	25	8	$3L+4$	13	7	0.6
PH mod	$4L+B+D+1$	49	8	$3L+D+3$	37	7	0.2

## Presqu'assez rapide, presqu'assez compact !

- adresses → offsets :  
⇒ 1 cycle et 1 instruction en plus, espace divisé par 2
- $3 \leq D \leq 25$  cycles, suivant processeurs et données

# Plan

- 1 Contexte et Motivation
- 2 Du Hachage à la Perfection
  - Hachage Parfait
  - Application à JAVA
- 3 Expérimentations et évaluation
- 4 Comparaison, Conclusion et Perspective**



# Autres travaux

## Test de sous-typage

- codage d'ordres (Ait-Kaci, Caseau, Habib, Vitek, Zibin, ..)
- efficace mais incompatible avec le chargement dynamique

## En JAVA

- techniques *ad hoc* : hachage + cache
- taille constante  $\Rightarrow$  espace mal optimisé et tests supplémentaires
- techniques dépareillées
- pas d'*inlining*
- mais efficacité statistique grâce au cache

## Le hachage parfait se compare honorablement

- compatible avec chargement dynamique
- efficacité constante et plutôt bonne

# Conclusion

## Un test de sous-typage presque parfait

- temps constant assez petit
- espace pas trop grand
- séquence assez courte
- compatible héritage multiple et chargement dynamique

## Une “invocation d’interface” presque parfaite

- pour les langages en sous-typage multiple
- avec une technique qui sert aux deux

## Expérimentation réelle

... recherche machine virtuelle JAVA.

# Perspective

## Mais un test de sous-typage pas parfait

- PH `and` prend trop de place : dégradation marquée avec la taille
- la division entière n'est pas efficace en temps
- le temps de `and` pour l'espace de `mod` serait parfait !

## Améliorations

- hachage *quasi-parfait* : décevant !
- PH + cache : 4 instructions + 2 cycles en plus ou  $2L + 2$  cycles au lieu de  $3L + D + 3$
- fonction *hash* à deux opérations mono-cycles
  - ▶ addition, décalage et opérations bit-à-bit
  - ▶ 1 cycle et 1 ou 2 instructions supplémentaires
  - ▶ pour un espace réduit de plus de moitié

## Un dernier doute

Depuis 30 ans,  
personne n'aurait songé à se servir du hachage parfait  
pour implémenter les objets ???