# Les talons d'Achille de la programmation par objets

Roland Ducournau

LIRMM – Université de Montpellier & CNRS

avec J. Privat (UQAM), F. Morandat (LaBRI),
O. Sallenave (IBM TJ Watson), G. Dupéron, J. Pagès, ...

GDR GPL, Paris – juin 2014

LIRMM

# Motivation : la bonne nouvelle

## Les objets

Devenus universels pour la programmation, la modélisation, etc.

## Théorie et technologie matures

- + 2000 ans après Aristote,
- ≈ un demi-siècle depuis Simula (1967),
- ≈ 30 ans pour les premiers langages *mainstream* (Eiffel, C++)
- ≈ 20 ans pour Java, 10-15 ans pour C#, Java 1.5 et Scala

☛ La plus grande réussite du dernier millénaire !

LIRMM

# Motivation : la bonne nouvelle

## Les objets

Devenus universels pour la programmation, la modélisation, etc.

## Théorie et technologie matures

- $+$ 2000 ans après Aristote,
- $\approx$ un demi-siècle depuis Simula (1967),
- $\approx$ 30 ans pour les premiers langages *mainstream* (Eiffel, C++)
- $\approx$ 20 ans pour Java, 10-15 ans pour C#, Java 1.5 et Scala

☛ La plus grande réussite du dernier millénaire !

LIRMM

# Motivation : la bonne nouvelle

## Les objets

Devenus universels pour la programmation, la modélisation, etc.

## Théorie et technologie matures

- $+$ 2000 ans après Aristote,
- $\approx$ un demi-siècle depuis Simula (1967),
- $\approx$ 30 ans pour les premiers langages *mainstream* (Eiffel, C++)
- $\approx$ 20 ans pour Java, 10-15 ans pour C#, Java 1.5 et Scala

☛ La plus grande réussite du dernier millénaire !

LIRMM

# Motivation : la mauvaise nouvelle

**Le plus grand échec du dernier millénaire ?**

Les langages de programmation par objets !

- Chacun séparément !
- Tous ensemble !

**mais il y a un espoir de synthèse ...**

- une pépite dans chacun !

LIRMM

# Motivation : la mauvaise nouvelle

**Le plus grand échec du dernier millénaire ?**

Les langages de programmation par objets !

- Chacun séparément !
- Tous ensemble !

**mais il y a un espoir de synthèse ...**

- une pépite dans chacun !

LIRMM

# Motivation : la mauvaise nouvelle

**Le plus grand échec du dernier millénaire ?**

Les langages de programmation par objets !

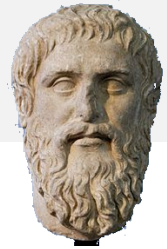
- Chacun séparément !
- Tous ensemble !

**mais il y a un espoir de synthèse ...**

- une pépite dans chacun !

LIRMM

# Motivation Platonicienne



☛ Le langage objet idéal existe !

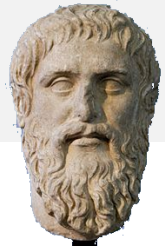
## Des arguments

- ontologiques (méta-modèle objet)
- de nécessité (rasoir d'Occam)
- formels (théorie des types, des ensembles, logique)
- empiriques
- de bon sens
- d'usage

... et un peu de mauvaise foi s'il le faut

LIRMM

# Motivation Platonicienne



☛ Le langage objet idéal existe !

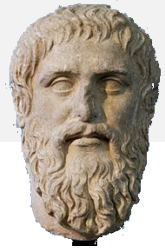
## Des arguments

- ontologiques (méta-modèle objet)
- de nécessité (rasoir d'Occam)
- formels (théorie des types, des ensembles, logique)
- empiriques
- de bon sens
- d'usage

... et un peu de mauvaise foi s'il le faut

LIRMM

# Motivation Platonicienne



☛ Le langage objet idéal existe !

## Des arguments

- ontologiques (méta-modèle objet)
- de nécessité (rasoir d'Occam)
- formels (théorie des types, des ensembles, logique)
- empiriques
- de bon sens
- d'usage

... et un peu de mauvaise foi s'il le faut

LIRMM

# Plan

1. Les spécifications des langages
2. L'implémentation des objets
3. Le cas Scala
4. Conclusions et perspectives

LIRMM

# Plan



LIRMM

# La surcharge statique (1/3)

*Mal nommer les choses,*
*c'est ajouter au malheur du monde*
(Albert Camus)

## Principe

- nom dénotant des entités différentes dans un contexte commun
- désambiguïsation par les types statiques du contexte
- origines : langages pré-objet comme PL/1 et C

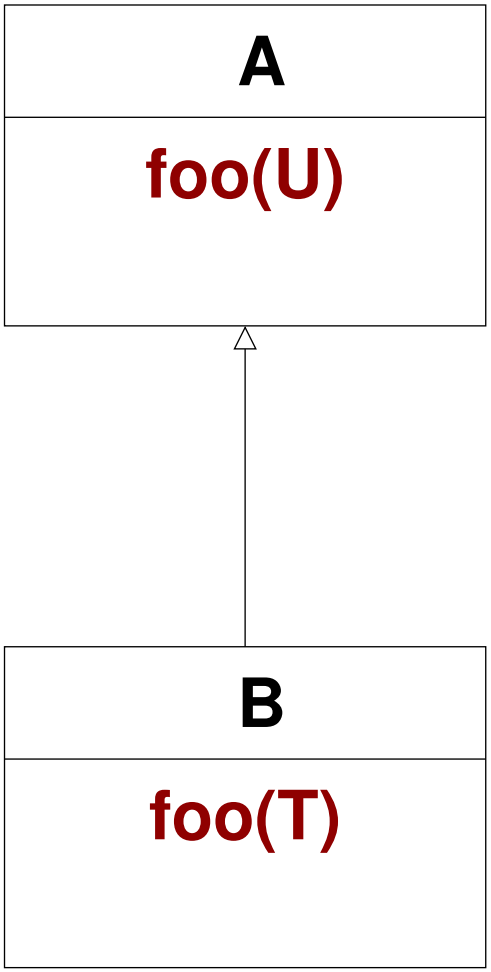
# La surcharge statique (1/3)

*Mal schtroumpfer les schtroumpfs,*
*c'est ajouter au malheur du monde*
(Albert Camus)

## Principe

- nom dénotant des entités différentes dans un contexte commun
- désambiguïsation par les types statiques du contexte
- origines : langages pré-objet comme PL/1 et C

# La surcharge statique (2/3)

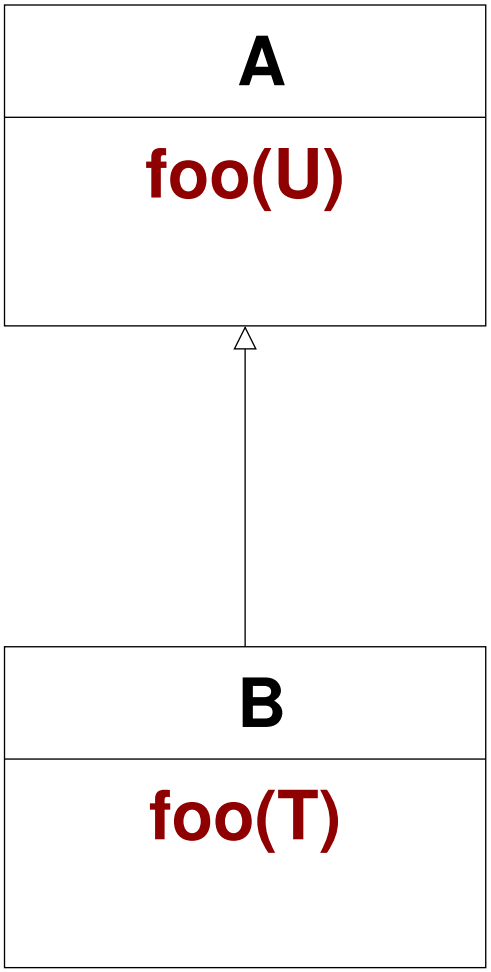



$U <: T$

x : B
y : U

| | x.foo(y) | |
|---|---|---|
| C++ | | |
| Java 1.4 | | |
| Java 1.5 | | |
| Scala | | |
| C# | | |

# La surcharge statique (2/3)

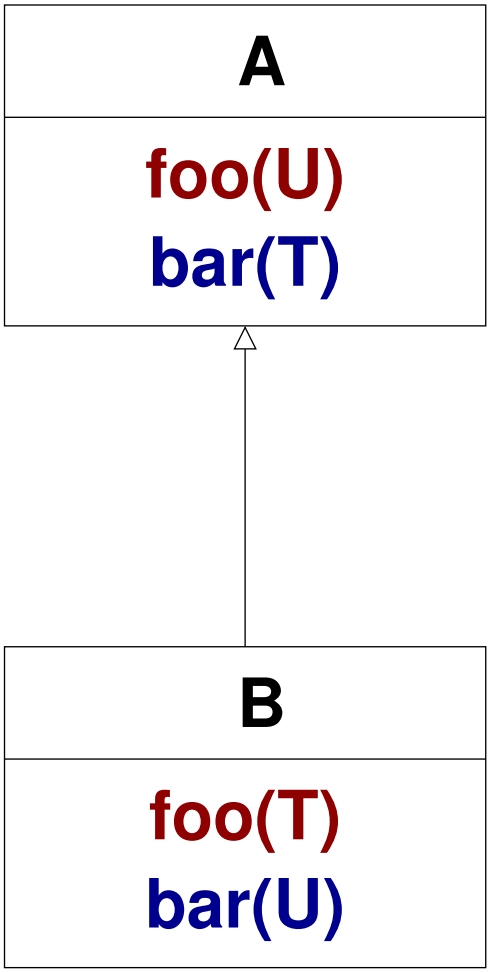



**U** $<:$ **T**

**x : B**
**y : U**

| | **x.foo(y)** |
|---|---|
| C++ | foo(**T**) |
| Java 1.4 | erreur |
| Java 1.5 | foo(**U**) |
| Scala | erreur |
| C# | foo(**T**) |

# La surcharge statique (2/3)

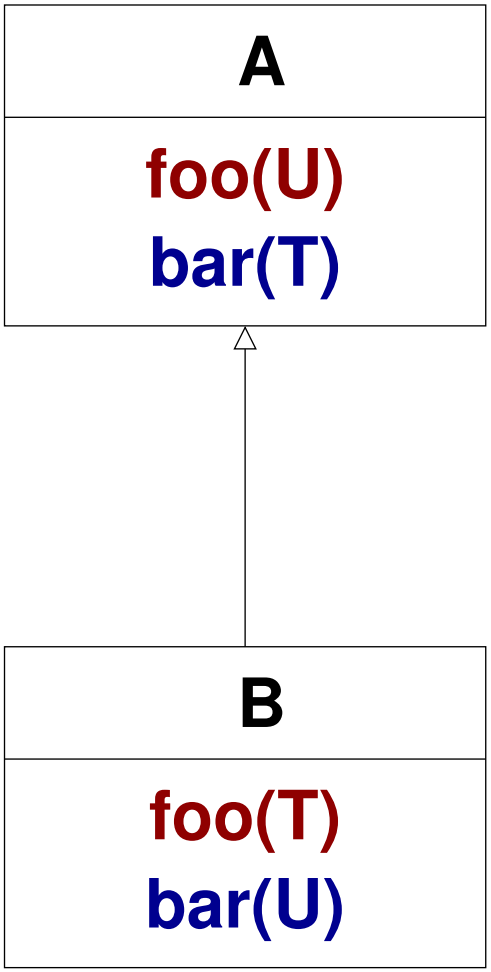



$U <: T$

x : B
y : U
z : T = new U

| | x.foo(y) | x.bar(z) |
|---|---|---|
| C++ | foo(T) | erreur |
| Java 1.4 | erreur | bar(T) |
| Java 1.5 | foo(U) | bar(T) |
| Scala | erreur | bar(T) |
| C# | foo(T) | bar(T) |

# La surcharge statique (2/3)




$U <: T$

```
x : B
y : U
z : T = new U
```

Confusion avec la covariance
ou la sélection multiple...

| | x.foo(y) | x.bar(z) |
|---|---|---|
| Eiffel | erreur | erreur |
| *intuition* | — | bar(**U**) |

# La surcharge statique (3/3)

## L'idéal

- l'exclure des spécifications (Eiffel, Nit),
- à défaut, prendre les spécifications de Java 1.5,
- surtout si l'on ne s'en sert pas !

☛ la surcharge statique équivaut à un renommage
donc renommez ! ! !

## Alternative : la sélection multiple (G. Castagna)

mais problème de modularité :

- spécifications difficiles en monde ouvert

LIRMM

# La surcharge statique (3/3)

## L'idéal

- l'exclure des spécifications (Eiffel, Nit),
- à défaut, prendre les spécifications de Java 1.5,
- surtout si l'on ne s'en sert pas !

☛ la surcharge statique équivaut à un renommage
donc renommez ! ! !

## Alternative : la sélection multiple (G. Castagna)

mais problème de modularité :

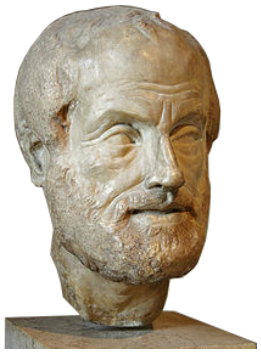
- spécifications difficiles en monde ouvert

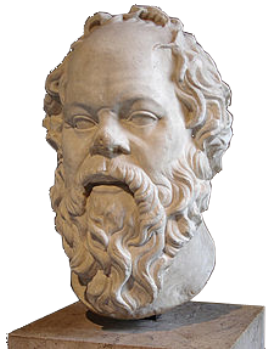
LIRMM

# L'héritage

*Socrate est un homme,*
*les hommes sont mortels,*
*donc Socrate est mortel.*
(d'après la tradition Aristotélicienne)

☛ l'idéal Platonicien de l'héritage serait Aristotélicien !



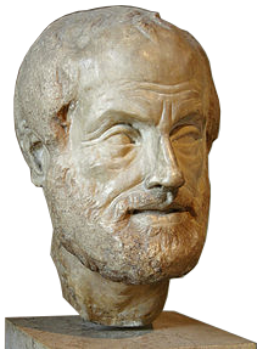


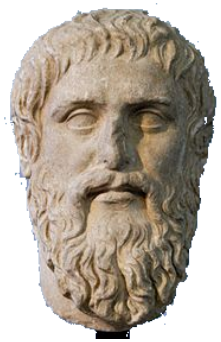
LIRMM

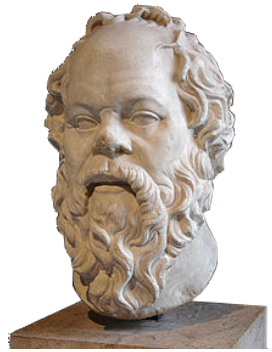
# L'héritage

*Socrate est un homme,*
*les hommes sont mortels,*
*donc Socrate est mortel.*
(d'après la tradition Aristotélicienne)

☛ l'idéal Platonicien de l'héritage serait Aristotélicien !







LIRMM

# L'héritage multiple (1/4)

## Omniprésent en typage statique

- au moins sous la forme du sous-typage multiple (Java, C#, Ada 2005),
- asymétrique, à base de mixins ou traits (Scala)
  ☛ classes de première et deuxième "classe"
- symétrique, sans restriction (C++, Eiffel, Nit),

## Pourquoi pas en typage dynamique ?

☛ parce que Java = Smalltalk + typage statique

# L'héritage multiple (1/4)

## Omniprésent en typage statique

- au moins sous la forme du sous-typage multiple (Java, C#, Ada 2005),
- asymétrique, à base de mixins ou traits (Scala)
  ☛ classes de première et deuxième "classe"
- symétrique, sans restriction (C++, Eiffel, Nit),

## Pourquoi pas en typage dynamique ?

☛ parce que Java = Smalltalk + typage statique

LIRMM

# L'héritage multiple (2/4)

## Conflits de nom mal gérés

Distinguer 2 méthodes introduites par 2 interfaces/classes différentes

- impossible en Java, Scala ou C++
- en C++ : incohérence attributs/méthodes
  ☛ 1 seul accesseur pour 2 attributs homonymes !

## Solution

- en C#, par qualification à la définition et *cast* à l'appel
- en Nit, par qualification
- en Eiffel, avec rename, undefine, redefine, compliqué et usage "incorrect" possible

LIRMM

# L'héritage multiple (3/4)

## Combinaison des méthodes

- double évaluation (C++, Eiffel)

☛ solution basée sur des linéarisations (Scala, CLOS, Python)

## Le cas de C++

- compliqué (avec ou sans virtual),
- schizophrène (sans virtual)
  - duplication d'attributs

  ☛ nombre exponentiel d'interprétations
  taille des objets exponentielle dans la taille du source

# L'héritage multiple (4/4)

## L'idéal

- héritage multiple symétrique
  - ☛ pas de distinction classes / traits
    (on peut les appeler classes ou traits, c'est égal...)
- désambiguisation par noms qualifiés
  - ☛ comme en C# (mais à l'utilisation aussi)
- combinaison des méthodes par linéarisation
  - ☛ comme en Scala (super) ou en CLOS (call-next-method)
- linéarisation monotone C3
  - ☛ comme en Python (pour une fois, j'en dis du bien ! !)

☛ [*Sci. Comp. Prog.* 2011]

LIRMM

# La généricité


## La généricité est devenue universelle

En typage statique,

- les langages à objets sont devenus génériques (tous)
- les langages génériques sont devenus à objets (Ada)

## Généricité + sous-typage = difficultés

- de spécifications
- de compréhension et d'utilisation
- d'implémentation

LIRMM

# Spécifications par l'implémentation


**Au moins 3 versions**

**hétérogène** pure substitution textuelle (C++)

- ☛ pas de types récursifs + explosion du code

**homogène** effacement de type et partage de code (Java 1.5, Scala)

- ☛ expressivité limitée, *casts* non sûrs, *boxing*

**mixte** partage ou spécialisation de code sans effacement de type (C#)

- ☛ meilleur compromis expressivité-efficacité-sûreté

LIRMM

# Généricité contrainte


## Au moins 3 spécifications des contraintes

- aucune (C++)
  ☛ pas de vérification/compilation avant l'instanciation
- type formel borné par sous-typage
  ☛ compréhension et utilisation simples
- borne récursive (*F-bounded*) (Java, C#, Scala, Eiffel, ...)
  ☛ puissant mais rigide et difficile à comprendre...
  pour cloner des structures strictement isomorphes

LIRMM

# Généricité et (co)variance


## Relations difficiles


- covariance généralisée non sûre (Eiffel)
  - ☛ type formel non sûr en paramètre de méthode
- covariance non sûre des tableaux (Java, C#)
  - ☛ écriture non sûre
- effacement de type (Java, Scala)
  - ☛ type formel non sûr en type de retour
- annotations de variance sûres
  - à la définition (Scala, C# uniquement sur les interfaces)
  - à l'utilisation (*wildcards* Java, Scala)

LIRMM

# Covariance des tableaux


## Un chat n'est pas un chien

- x : Cat[]
- ...
- y : Animal[]
- y = x // *dangereux mais compilé*
- y[i] = new Dog // *compilé mais exception*

# Effacement de type


## Un chat n'est pas un chien (re)

- x : Stack<Cat>
- ...
- y : Stack<Dog>
- y = (Stack<Dog>)x  // *stupide mais compilé*
- y.push(new Dog)  // *le type est effacé ! Alzheimer*
- ...
- z : Cat = x.pop()  // *exception à retardement*

☛ traçabilité désastreuse des erreurs

# Effacement de type


## Un chat n'est pas un chien (re)

- x : Stack<Cat>
- ...
- y : Stack<Dog>
- y = (Stack<Dog>)x  // *stupide mais compilé*
- y.push(new Dog)  // *le type est effacé ! Alzheimer*
- ...
- z : Cat = x.pop()  // *exception à retardement*

☛ traçabilité désastreuse des erreurs

# Complications


- interface Comparable<T extends Comparable<T>>
- class OrderedSet<T extends Comparable<T>>
- class A implements Comparable<A>
- OrderedSet<A>

- class B extends A
- OrderedSet<B>

**Annotation de contravariance**

- Java : class OrderedSet<T extends Comparable< **? super** T>>
- Scala : interface Comparable<−T extends Comparable<T>>

# Complications


- interface Comparable<T extends Comparable<T>>
- class OrderedSet<T extends Comparable<T>>
- class A implements Comparable<A>
- OrderedSet<A>

- class B extends A
- OrderedSet<B>

## Annotation de contravariance

- Java : class OrderedSet<T extends Comparable< ? super T>>
- Scala : interface Comparable<−T extends Comparable<T>>

# Complications


- interface Comparable<T extends Comparable<T>>
- class OrderedSet<T extends Comparable<T>>
- class A implements Comparable<A>
- OrderedSet<A>

- class B extends A implements Comparable<B>
- OrderedSet<B>

## Annotation de contravariance

- Java : class OrderedSet<T extends Comparable< ? super T>>
- Scala : interface Comparable<−T extends Comparable<T>>

# Complications

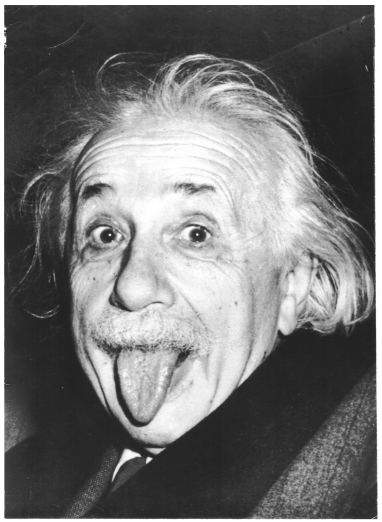

- interface Comparable<T extends Comparable<T>>
- class OrderedSet<T extends Comparable<T>>
- class A implements Comparable<A>
- OrderedSet<A>

- class B extends A
- OrderedSet<B>

## Annotation de contravariance

- Java : class OrderedSet<T extends Comparable< **? super** T>>
- Scala : interface Comparable<−T extends Comparable<T>>

# La hiérarchie des programmeurs

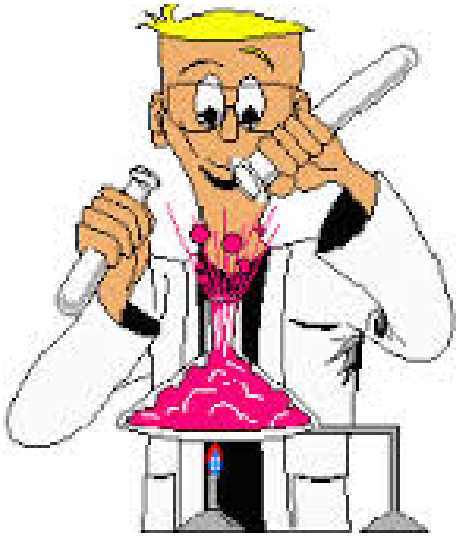



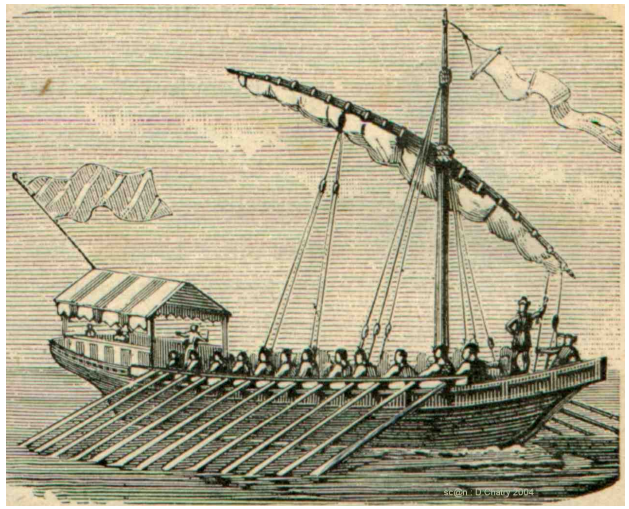
concepteur de langage

pure lumière de la programmation

implémenteur de langage ou de bibliothèque

ingénieux ingénieur



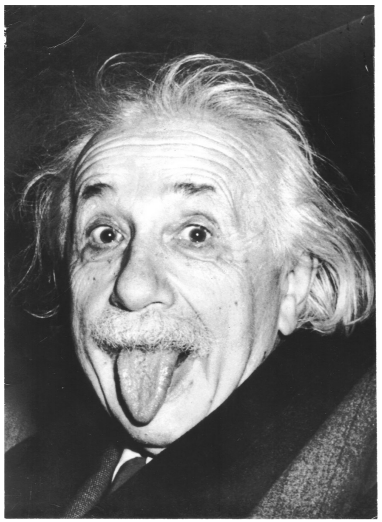


programmeur de base

obscur soutier

☛ C'est déloyal de faire supporter au soutier ce qu'un ingénieux ingénieur ou une pure lumière aurait pu faire...

# La hiérarchie des programmeurs

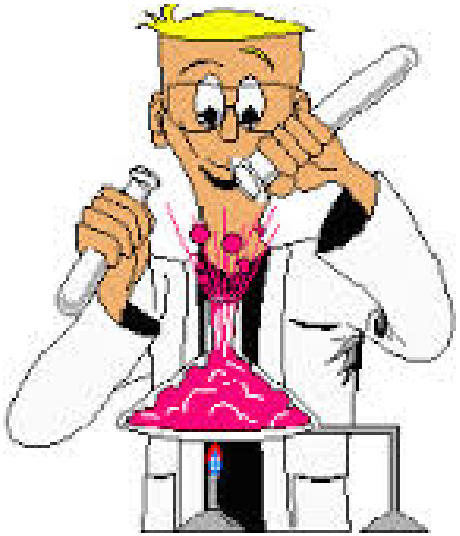



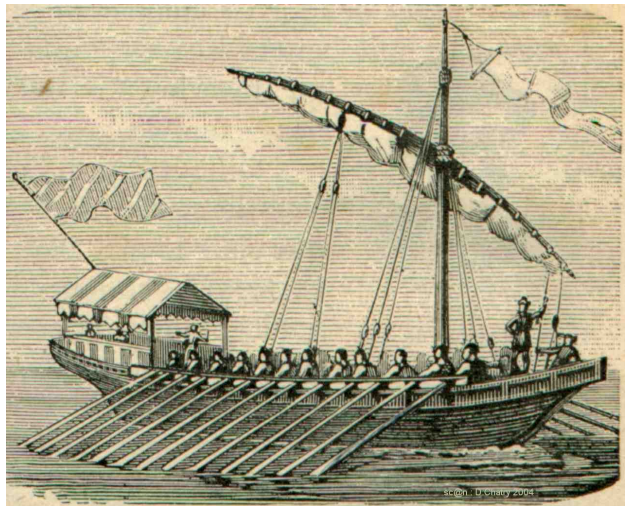
concepteur de langage

pure lumière de la programmation

implémenteur de langage ou de bibliothèque

ingénieux ingénieur



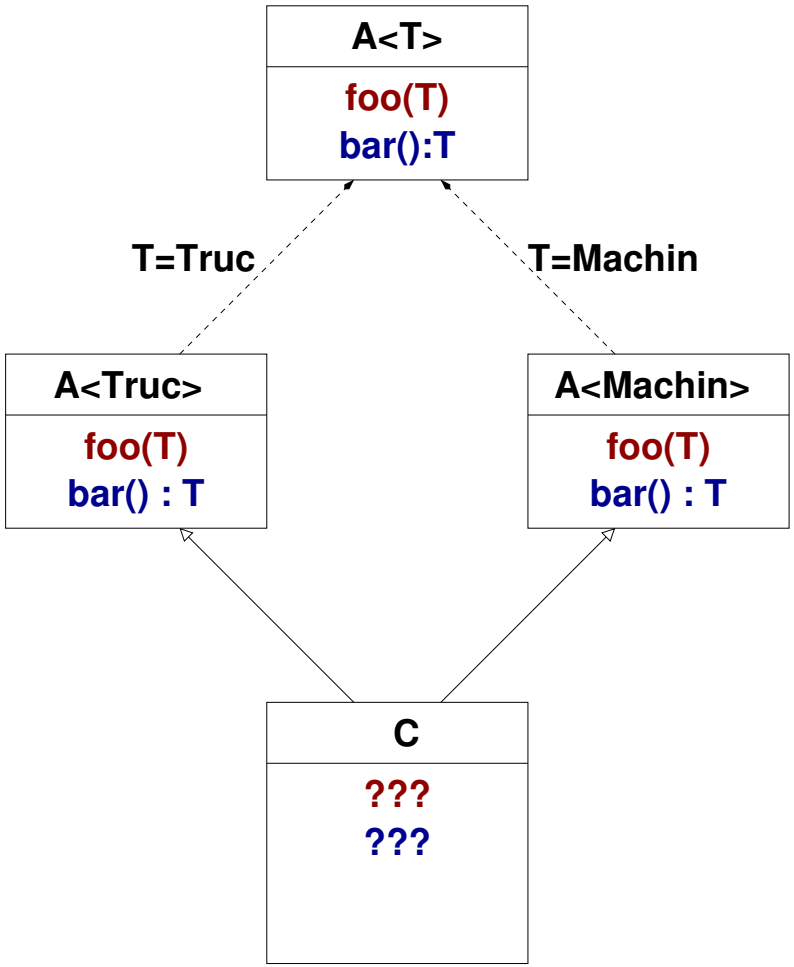


programmeur de base

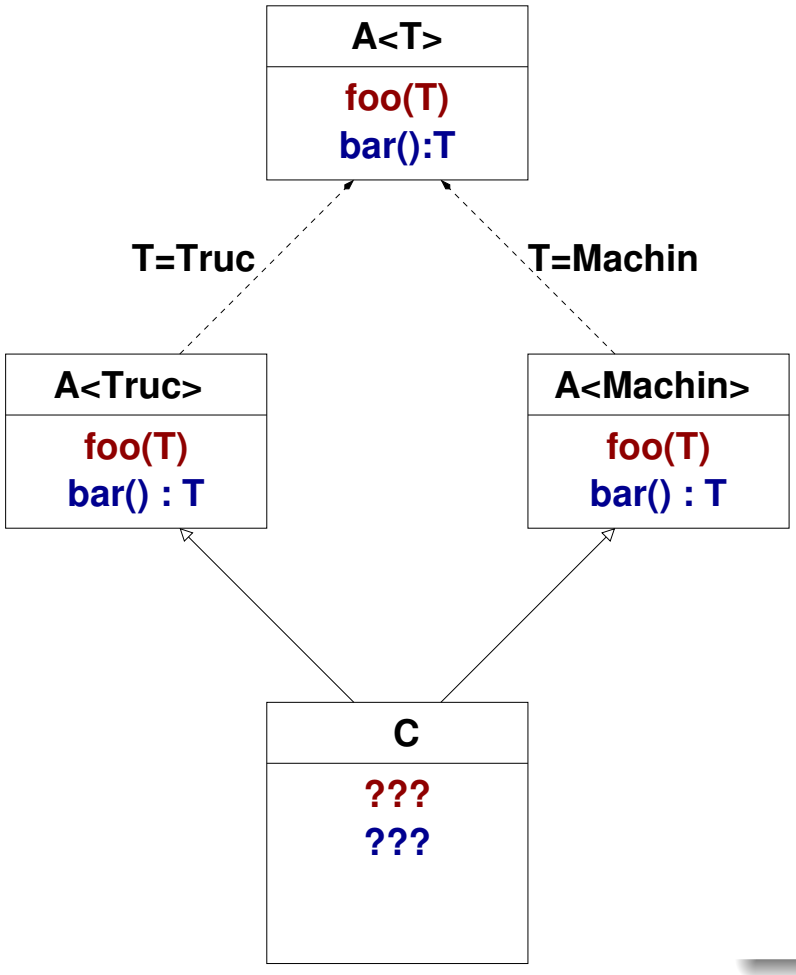
obscur soutier

☛ C'est déloyal de faire supporter au soutier ce qu'un ingénieux ingénieur ou une pure lumière aurait pu faire...

# Surcharge + héritage multiple +

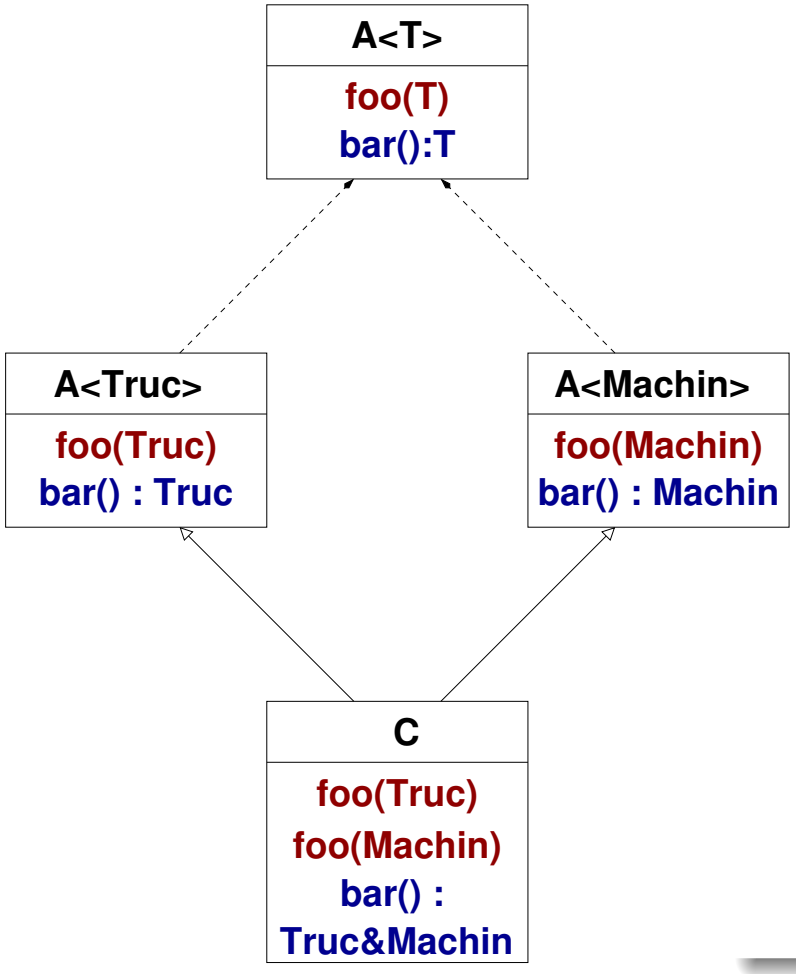



# Surcharge + héritage multiple +

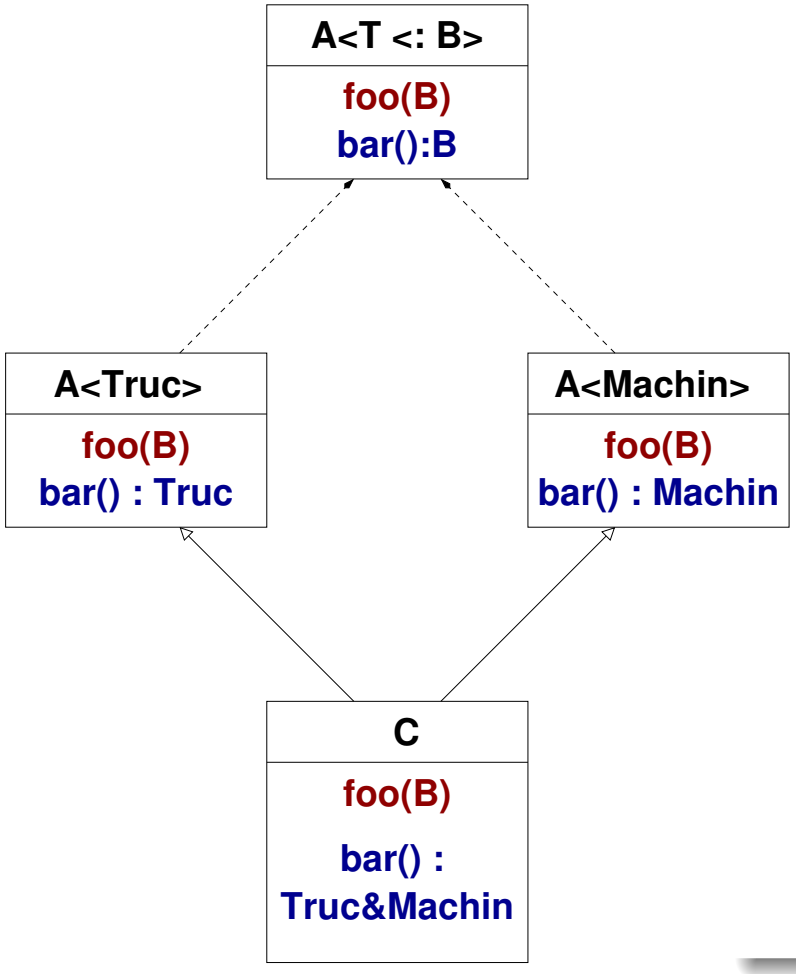

- arguments mathématico-ontologiques et bon-sens
  ☛ impossible

# Surcharge + héritage multiple +

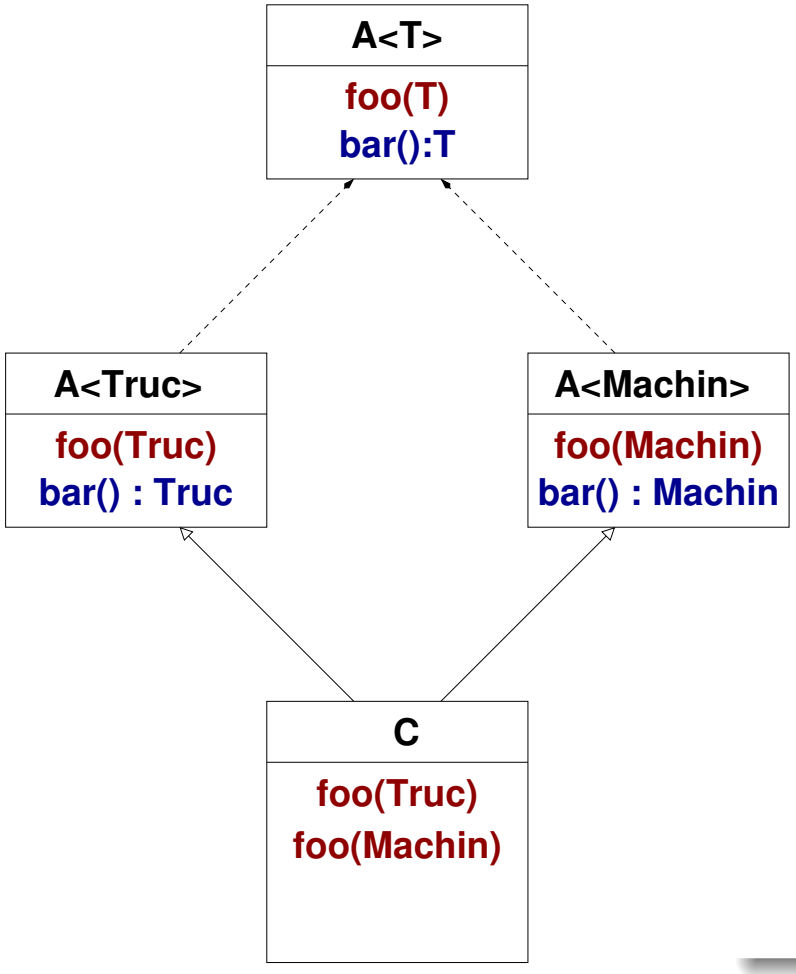

- en C++
  (substitution textuelle)
  ☛ aucun problème ! !

# Surcharge + héritage multiple +


- pour la JVM (effacement de type)
  ☛ aucun problème
  mais interdit par Java !

# Surcharge + héritage multiple +

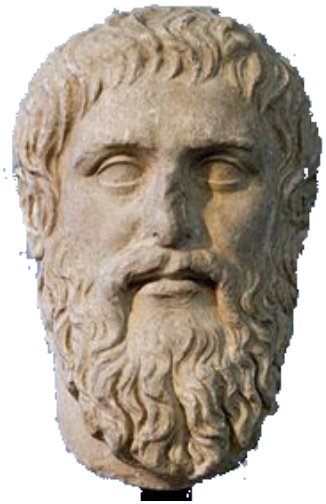




- en C# (pas de covariance)
  - ☛ impossible pour **bar**
  - ☛ aucun problème pour **foo**

# La généricité (fin)


## L'idéal

- sans effacement de type
- implémentation mixte à la C#
- variance à la définition et à l'utilisation (Scala)
- invariance des tableaux
- borne récursive (*F-bounded*)
- pas d'héritage d'instanciations multiples
- pas de surcharge statique sur les types formels
- meilleur support de l'IDE



LIRMM

# Plan



LIRMM

# Implémentation des objets (1/3)

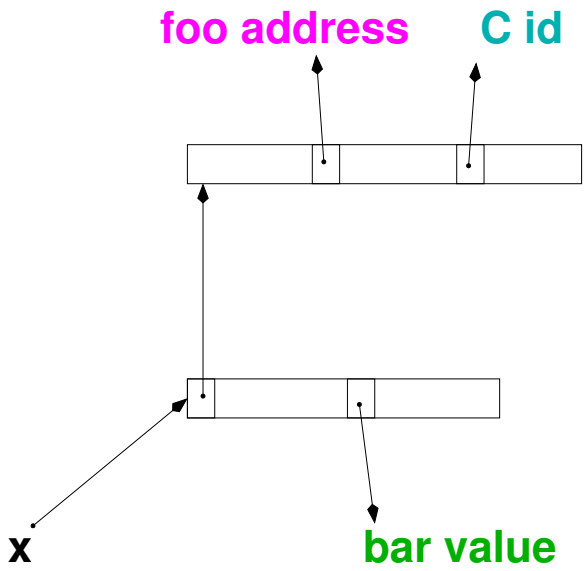
```
x.foo(arg)
x.bar
x instanceOf C
```

**La programmation objet implique**

- **invocation de méthode** (aka envoi de message, liaison tardive)
- **accès aux attributs**
- **test de sous-typage** (aka *casts*)

# Implémentation des objets (1/3)

x.foo(arg)
x.bar
x instanceOf C

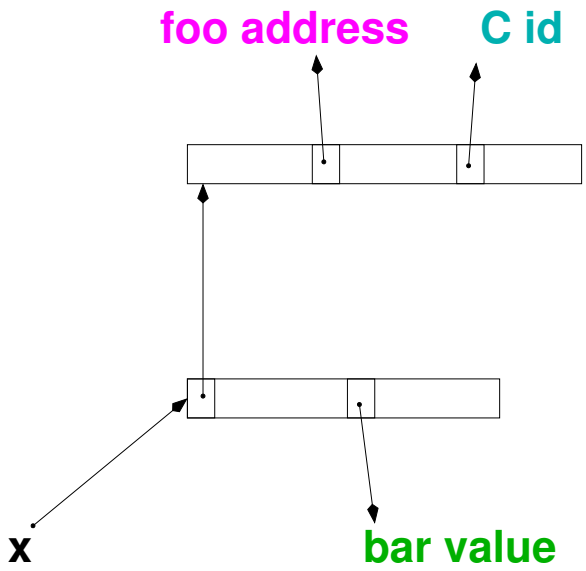



**La programmation objet implique**

- **invocation de méthode** (aka envoi de message, liaison tardive)
- **accès aux attributs**
- **test de sous-typage** (aka *casts*)

# Implémentation des objets (1/3)

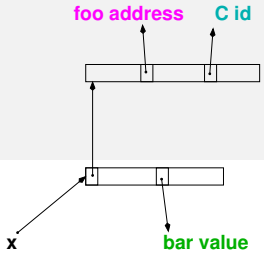
x.foo(arg)
x.bar
x instanceOf C




## Le problème

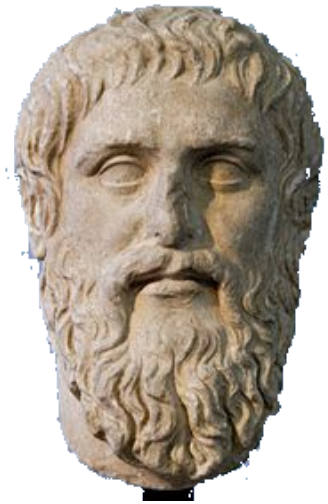
- Quelle structure de données ?
- Comment déterminer les positions de **foo**, **bar** et **C** dans **x** ?
- Quand ? à la compilation, au chargement ou à l'exécution ?

# Implémentation des objets (2/3)

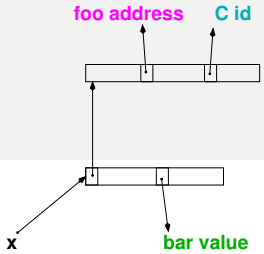



## Idéal de performance

- temps constant
- espace *scalable*
- code compact (*inlinable*)



# Implémentation des objets (3/3)

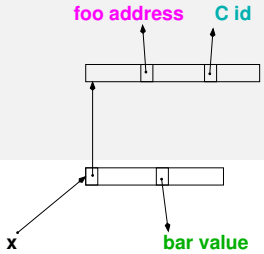



## Solutions complètes

Dans deux cas seulement

- en sous-typage simple
  - ☛ héritage simple, avec types = classes
- sous l'hypothèse du monde fermé
  - ☛ compilation ou édition de liens globales

# En sous-typage simple (1/2)

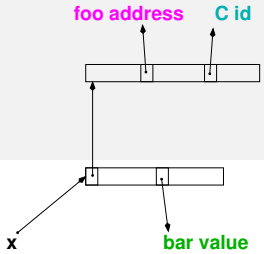


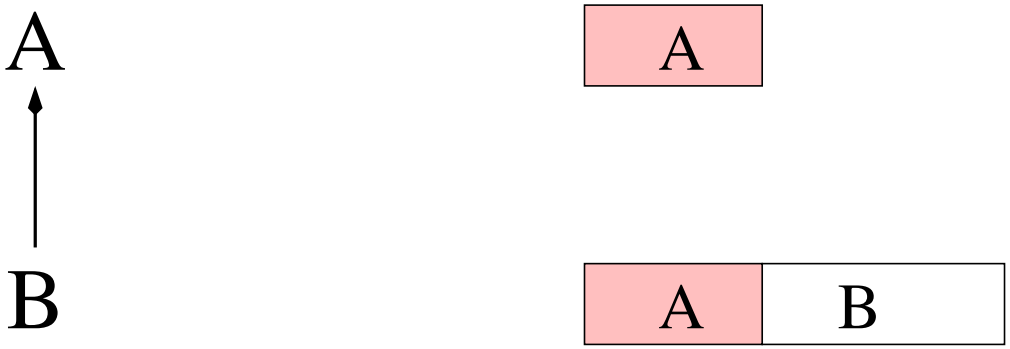

A

A

L'implémentation d'une classe étend celle de son unique super-classe

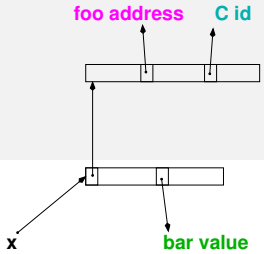
# En sous-typage simple (1/2)

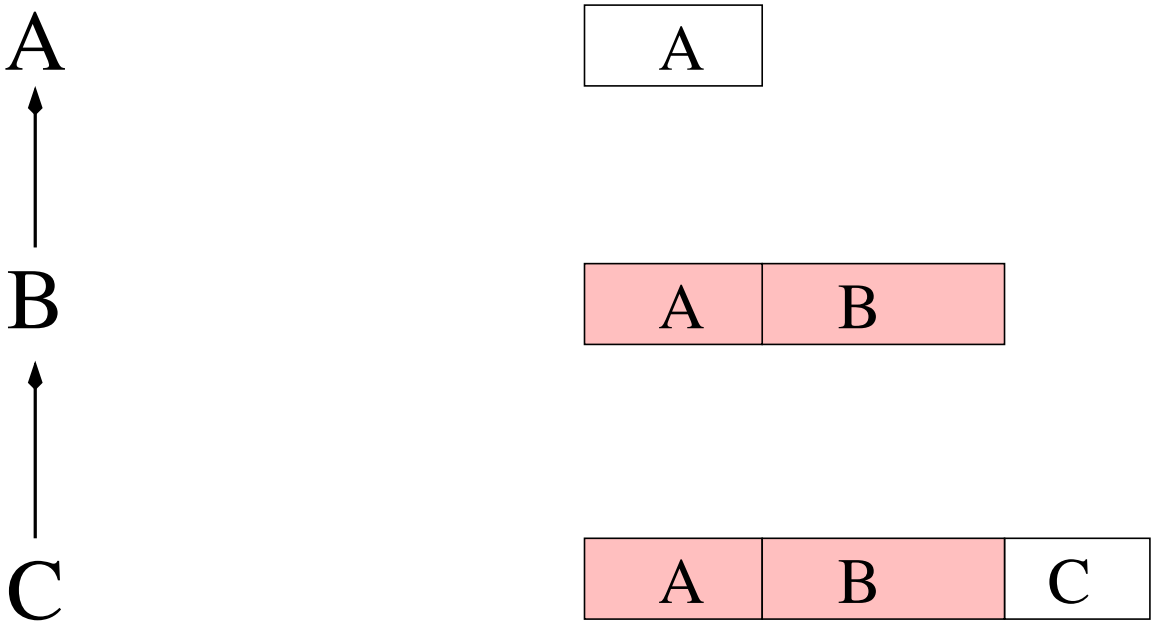



L'implémentation d'une classe étend celle de son unique super-classe

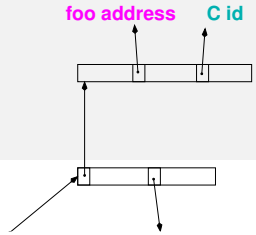
# En sous-typage simple (1/2)




L'implémentation d'une classe étend celle de son unique super-classe

# En sous-typage simple (2/3)




## Bonnes propriétés

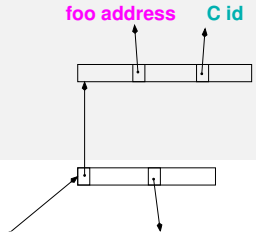
- même principe pour méthodes et attributs
- les méthodes/attributs introduits par une classe sont groupés,
- avec des positions invariantes,
- les super-classes sont totalement ordonnées,
- l'ordre de la super-classe est un préfixe de celui de la sous-classe

## Idéal de performance

- temps constant
- espace linéaire dans la taille de la relation d'héritage
  i.e. quadratique dans le nombre de super-classes (pire des cas)
- compatible avec monde ouvert et chargement dynamique

LIRMM

# En sous-typage simple (2/3)




## Bonnes propriétés

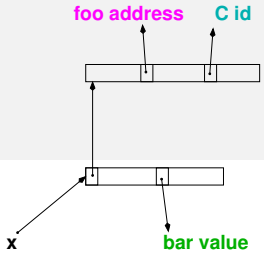
- même principe pour méthodes et attributs
- les méthodes/attributs introduits par une classe sont groupés,
- avec des positions invariantes,
- les super-classes sont totalement ordonnées,
- l'ordre de la super-classe est un préfixe de celui de la sous-classe

## Idéal de performance

- temps constant
- espace linéaire dans la taille de la relation d'héritage
  i.e. quadratique dans le nombre de super-classes (pire des cas)
- compatible avec monde ouvert et chargement dynamique

LIRMM

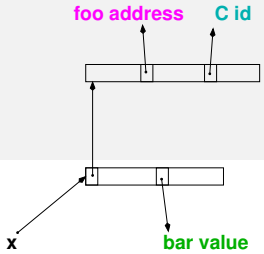
# En sous-typage simple (3/3)




**La mauvaise nouvelle**

Aucun langage en sous-typage simple !

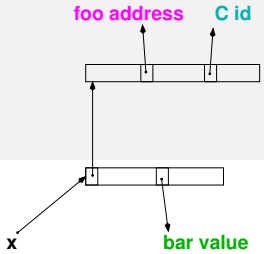
# En sous-typage simple (3/3)

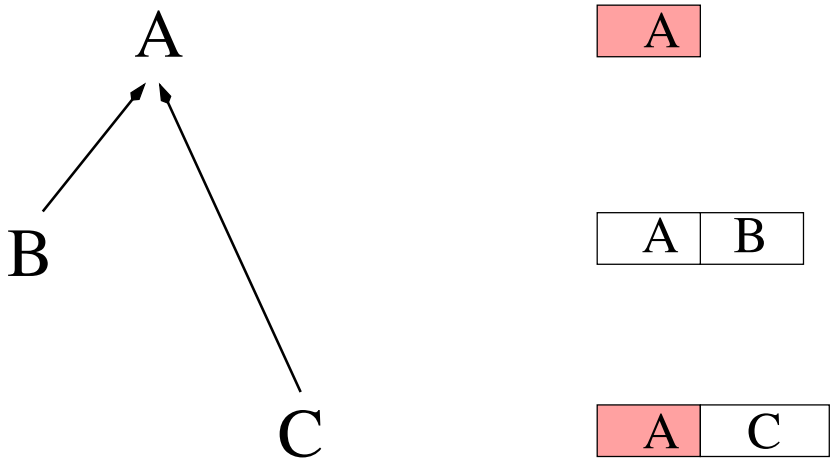



**La mauvaise nouvelle**

Aucun langage en sous-typage simple !

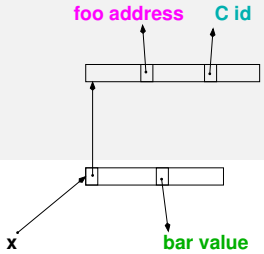
# Héritage multiple (1/4)

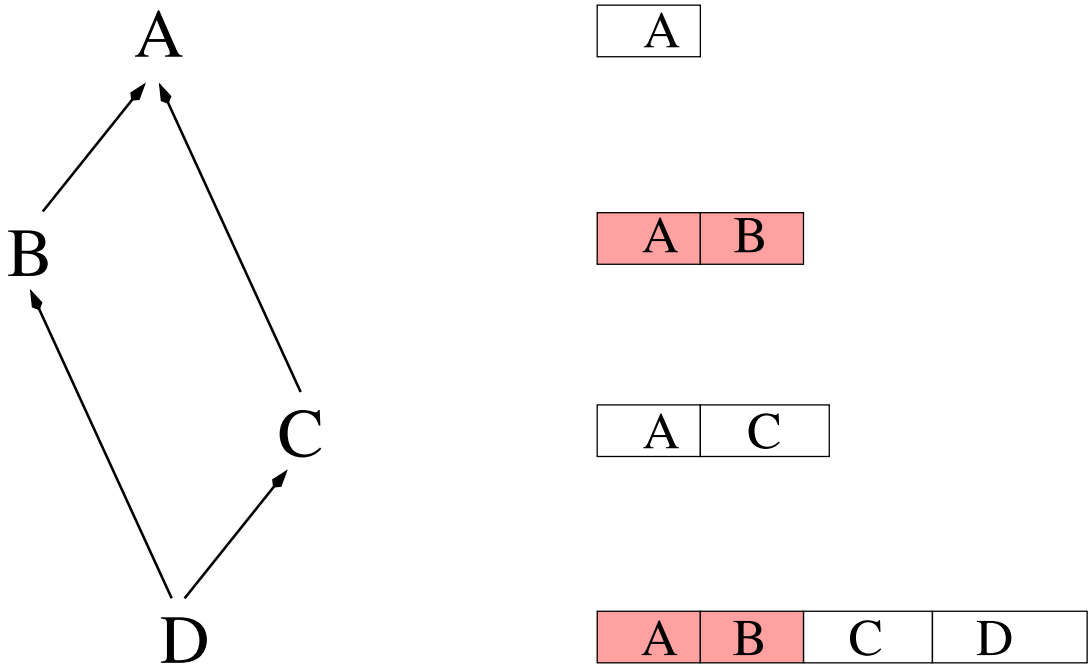



- $D$ vérifie la condition du préfixe vis-à-vis de $B$, pas de $C$,
- la position de $C$ n'est pas invariante

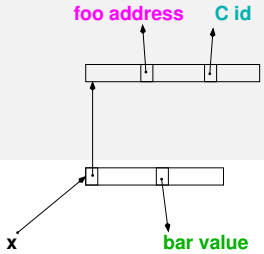
# Héritage multiple (1/4)

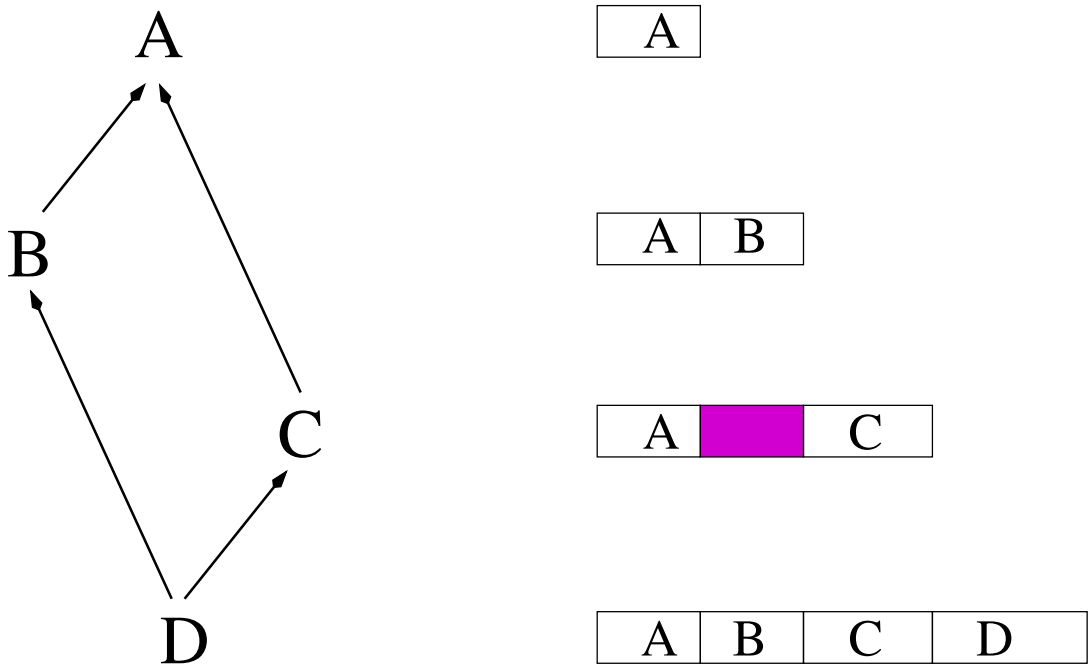



- *D* vérifie la condition du préfixe vis-à-vis de *B*, pas de *C*,
- la position de *C* n'est pas invariante

# Héritage multiple (2/4)

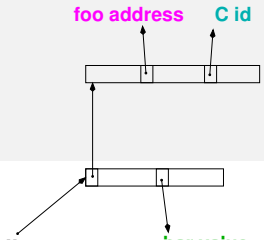



## Solution complète en monde fermé

- optimisations globales de type coloration

# Héritage multiple (3/4)




## Solutions partielles en monde ouvert

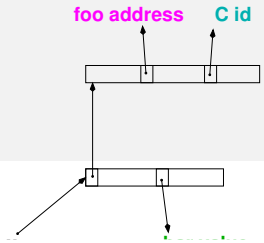
- en sous-typage multiple (Java, C#)
  - ☛ pour classes seulement
- en typage dynamique et héritage simple (Smalltalk)
  - ☛ pour accès aux attributs et tests de sous-typage

## Solution partielle et compliquée

Implémentation par sous-objets (virtual inheritance de C++)

- pour invocation de méthodes et accès aux attributs,
- temps constant, mais constante élevée,
- espace cubique dans le nombre de super-classes
  exponentiel sans virtual ! !

# Héritage multiple (3/4)




## Solutions partielles en monde ouvert

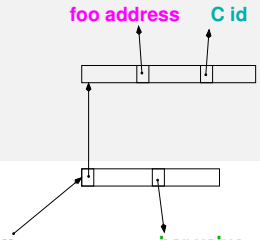
- en sous-typage multiple (Java, C#)
  - ☛ pour classes seulement
- en typage dynamique et héritage simple (Smalltalk)
  - ☛ pour accès aux attributs et tests de sous-typage

## Solution partielle et compliquée

Implémentation par sous-objets (virtual inheritance de C++)

- pour invocation de méthodes et accès aux attributs,
- temps constant, mais constante élevée,
- espace cubique dans le nombre de super-classes
  exponentiel sans virtual ! !

# Héritage multiple (3/4)




## Solutions partielles en monde ouvert

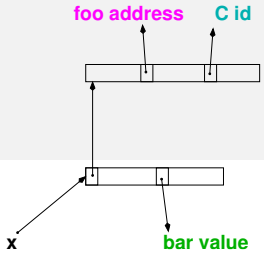
- en sous-typage multiple (Java, C#)
  - ☛ pour classes seulement
- en typage dynamique et héritage simple (Smalltalk)
  - ☛ pour accès aux attributs et tests de sous-typage

## Solution partielle et compliquée

Implémentation par sous-objets (virtual inheritance de C++)

- pour invocation de méthodes et accès aux attributs,
- temps constant, mais constante élevée,
- espace cubique dans le nombre de super-classes exponentiel sans virtual ! !

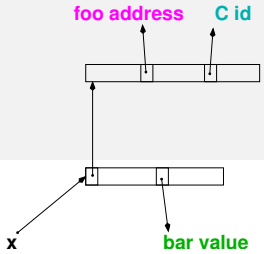
# Héritage multiple (4/4)




**Pas de solution complete, i.e. scalable, en monde ouvert**

- en sous-typage multiple pour les interfaces (Java, C#)
- en typage dynamique pour l'appel de méthodes (Smalltalk)
- en héritage multiple pour les tests de sous-typage (C++)

# Héritage multiple (4/4)

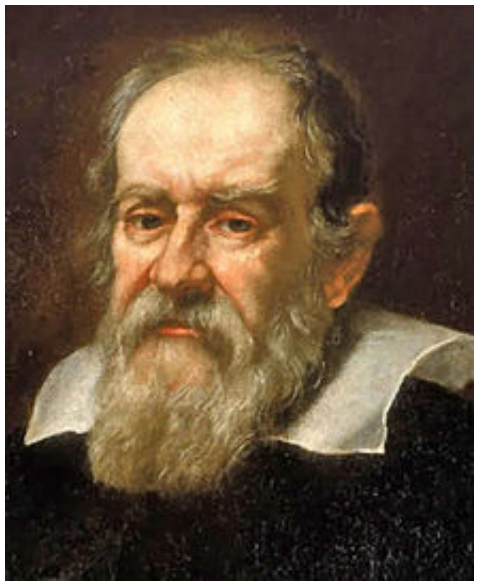



**Pas de solution complete, i.e. scalable, en monde ouvert**

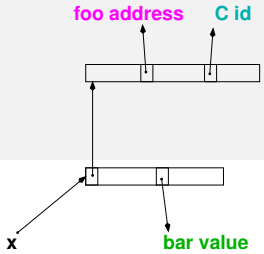
- en sous-typage multiple pour les interfaces (Java, C#)
- en typage dynamique pour l'appel de méthodes (Smalltalk)
- en héritage multiple pour les tests de sous-typage (C++)

Eppur si muove !

Galileo Galilei



# Héritage multiple (4/4)

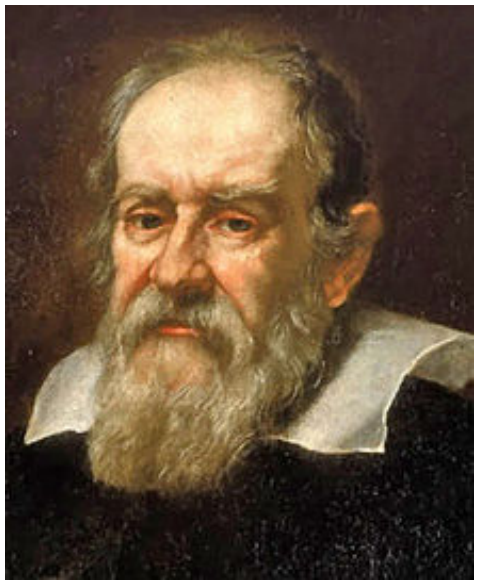



## Etant donnés

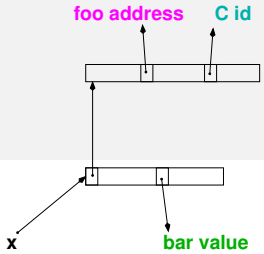
- une implémentation,
- un compilateur JIT optimisant,
- ☛ écrire un programme qui fait échouer les optimisations.

Eppur si muove !

Galileo Galilei



# Conclusion

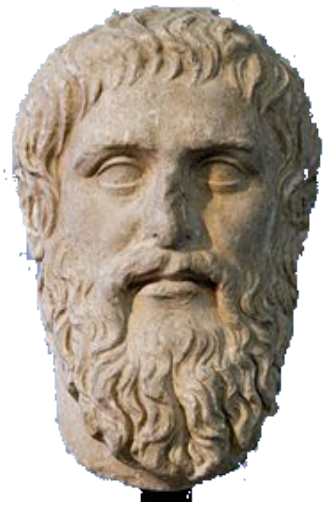



## Recherche (désespérément) implémentation

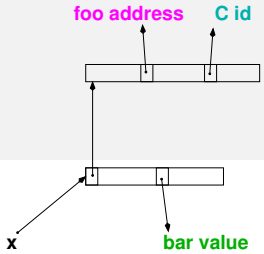
- respectant l'idéal de performance espace-temps
- compatible avec le chargement dynamique
- et l'héritage multiple

☛ pour les interfaces Java et .Net

☛ pour les langages en héritage multiple comme Eiffel, Nit ou Scala



# Ranger et retrouver




## Hachage

- le problème de l'implémentation
  = instance du problème général ranger et retrouver,
- le hachage est une solution générique

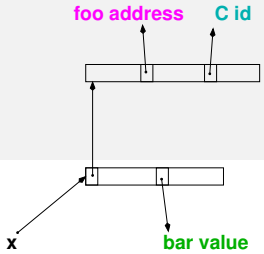
☛ le hachage pourrait être une solution du problème spécifique

## Hachage pour l'implémentation des objets

- hacher les identifiants de méthodes, attributs et classes
  ☛ petits entiers
- temps constant, mais en moyenne seulement

☛ dans le pire des cas, aussi mauvais qu'une recherche séquentielle

# Ranger et retrouver




## Hachage

- le problème de l'implémentation
  = instance du problème général ranger et retrouver,
- le hachage est une solution générique

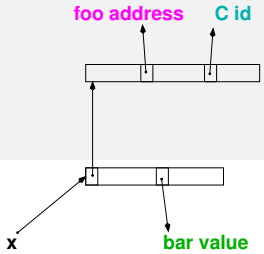
☛ le hachage pourrait être une solution du problème spécifique

## Hachage pour l'implémentation des objets

- hacher les identifiants de méthodes, attributs et classes
  ☛ petits entiers
- temps constant, mais en moyenne seulement

☛ dans le pire des cas, aussi mauvais qu'une recherche séquentielle

# Hachage parfait (1/5)



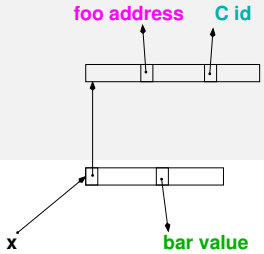

Délicieux oxymore !

## Principe

- hachage sans collisions, en temps vraiment constant
- pour hacher des ensembles immutables

☛ c'est notre cas
pas de modification incrémentale des classes

# Hachage parfait (2/5)

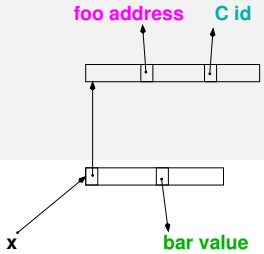



## Approche naïve

Pour chaque classe nouvellement chargée

1. calculer l'ensemble des éléments dont l'ID est à hacher,
   - classes pour le test de sous-typage,
   - méthodes
   - attributs
2. affecter des ID, pour
   - classes nouvellement chargées
   - méthodes et attributs nouvellement introduits
3. calculer le paramètre de hachage optimal pour l'ensemble des ID

# Hachage parfait (3/5)

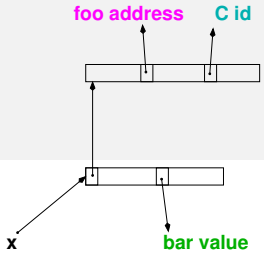



## Approche optimisée (aka numérotation parfaite)

### Problème inverse

1. calculer l'ensemble des éléments dont l'ID est à hacher,
2. partitionner en deux sous-ensembles
   - éléments hérités, avec ID
   - éléments nouvellement introduits, sans ID
3. calculer le paramètre de hachage optimal, à partir de
   - des ID du premier ensemble,
   - du cardinal du deuxième ensemble
4. allouer des ID libres hachés sur des places libres
5. les affecter aux éléments du deuxième ensemble

# Hachage parfait (4/5)

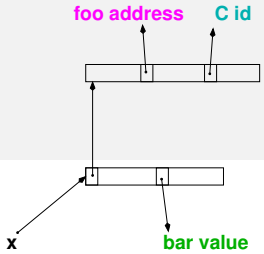



## Fonctions de hachage

- deux candidates naturelles avec une instruction unique
  - et binaire = 1 cycle
  - modulo ≈ 10-50 cycles !
- modulo produit des tables plus compactes

☛ et binaire est le meilleur compromis

# Hachage parfait (4/5)

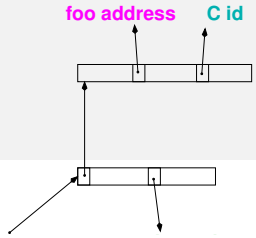



## Fonctions de hachage

- deux candidates naturelles avec une instruction unique
  - et binaire = 1 cycle
  - modulo ≈ 10-50 cycles !
- modulo produit des tables plus compactes

☛ et binaire est le meilleur compromis

# Hachage parfait (5/5)




## Définition mathématique

**soit** $\mathbf{C}$ la classe considérée
$\mathbf{I_C}$ l'ensemble des ID à hacher

**alors** $\mathbf{m_C}$ est le plus petit entier tel que
la fonction $\mathbf{x} \mapsto \mathbf{et}(\mathbf{x}, \mathbf{m_C})$ soit injective sur $\mathbf{I_C}$

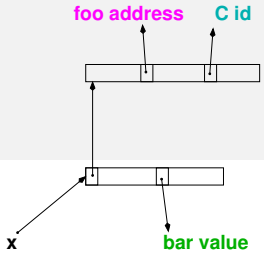
☛ la taille de la table est $\mathbf{H_C} = \mathbf{m_C} + \mathbf{1}$.

## Algorithmes

- simples mais non triviaux
- faible complexité
- utilisables à l'exécution sans problème

☛ [*TOPLAS* 2008, *Soft. Pract. & Exp.* 2011]

# Evaluation (1/3)




## Occupation mémoire

- évaluations théoriques et pratiques
- simulations sur de grands benchmarks

☛
  - et binaire est erratique
  - simulations randomisées nécessaires

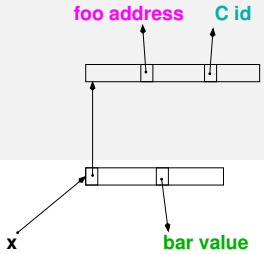
- l'approche naïve n'est pas terrible ☹
- l'approche inverse *scalable* quand seules les classes sont hachées

☛ ratio d'occupation pas nettement supérieur à 2 ☺

## Conclusion 1

☛ PH devrait être restreint à de petits ensembles

# Evaluation (2/3)




## Performances temporelles

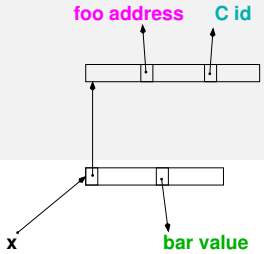
- par dénombrement de cycles
- dans le compilateur PRM (F. Morandat) [*Oopsla 2009*]

☛
- surcoût marqué par rapport au sous-typage simple
- pas pire (et beaucoup plus simple) que les sous-objets de C++

## Conclusion 2

☛ PH devrait être restreint à un sous-ensemble des sites d'appel

# Evaluation (3/3)

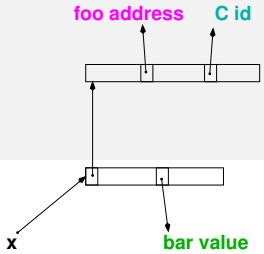


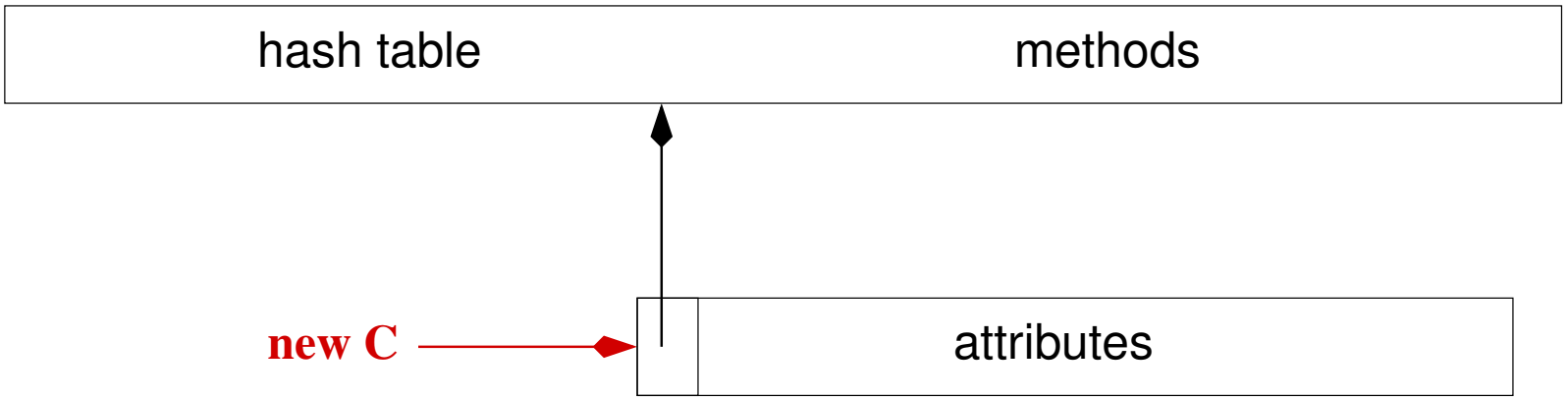

## Pistes pour utiliser PH

- en sous-typage multiple
  - pour les invocations d'interface
  - hacher les interfaces
- en héritage multiple, typage statique [*J. Obj. Tech.* 2012]
  - pour les invocations de classes à positions multiples
  - hacher les classes

☛ toujours avec compilation dynamique

# Application aux interfaces Java

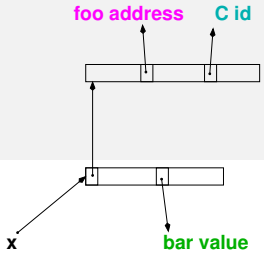


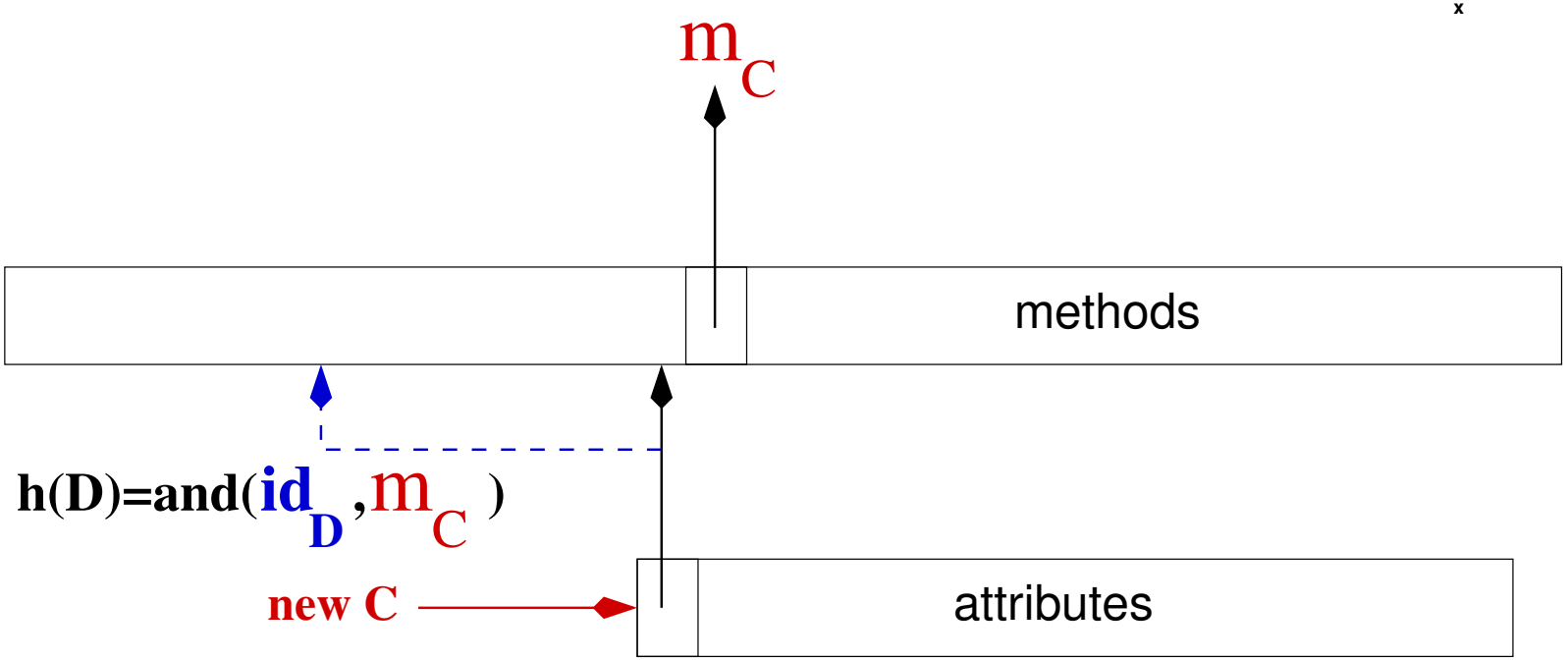




## Tables de méthodes bidirectionnelles

- partie positive pour les classes, comme en sous-typage simple
- partie négative pour la table de hachage

# Application aux interfaces Java

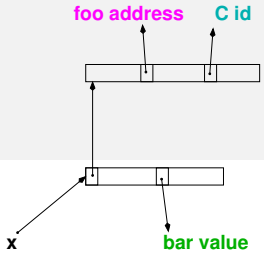


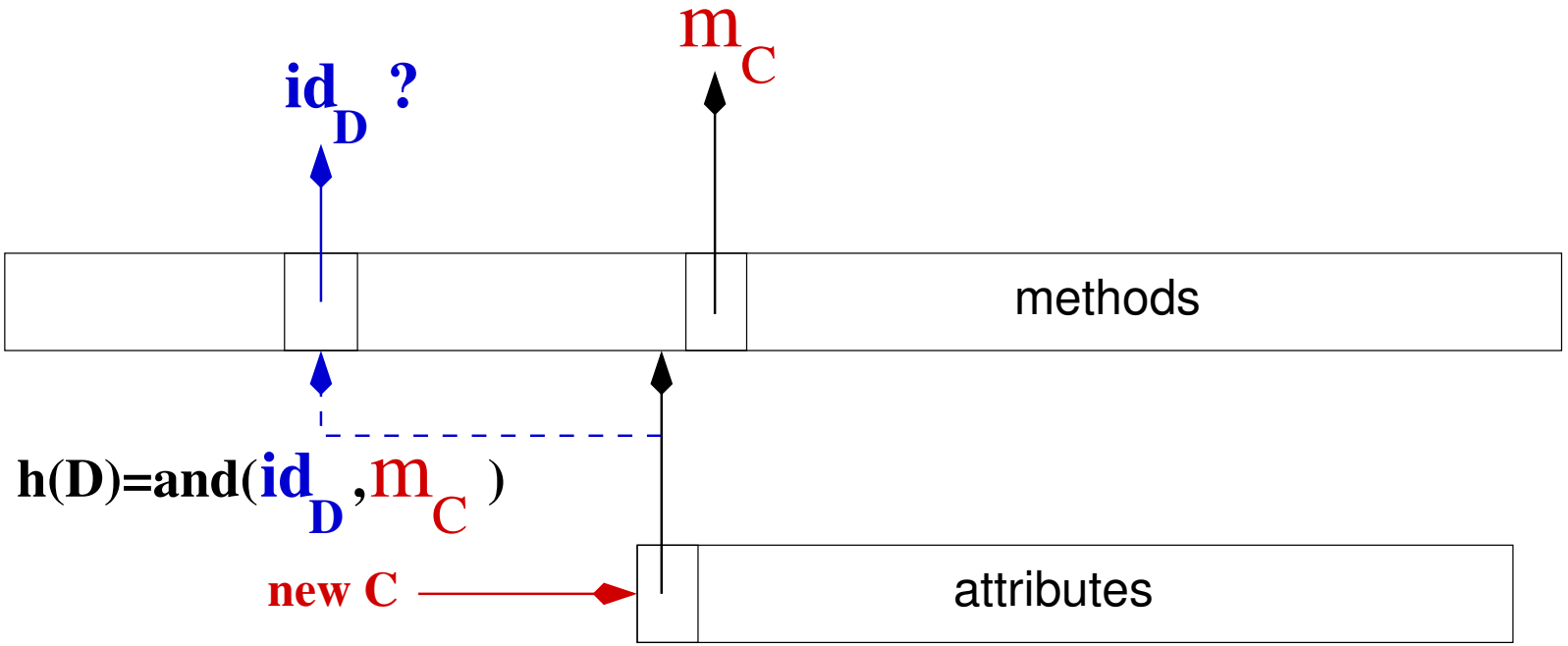




## Test de sous-typage

- o = instance d'une classe C inconnue,
- D est une interface possiblement implémentée par C,
- ☛ **o instanceOf D ?**

# Application aux interfaces Java

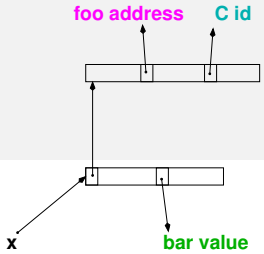


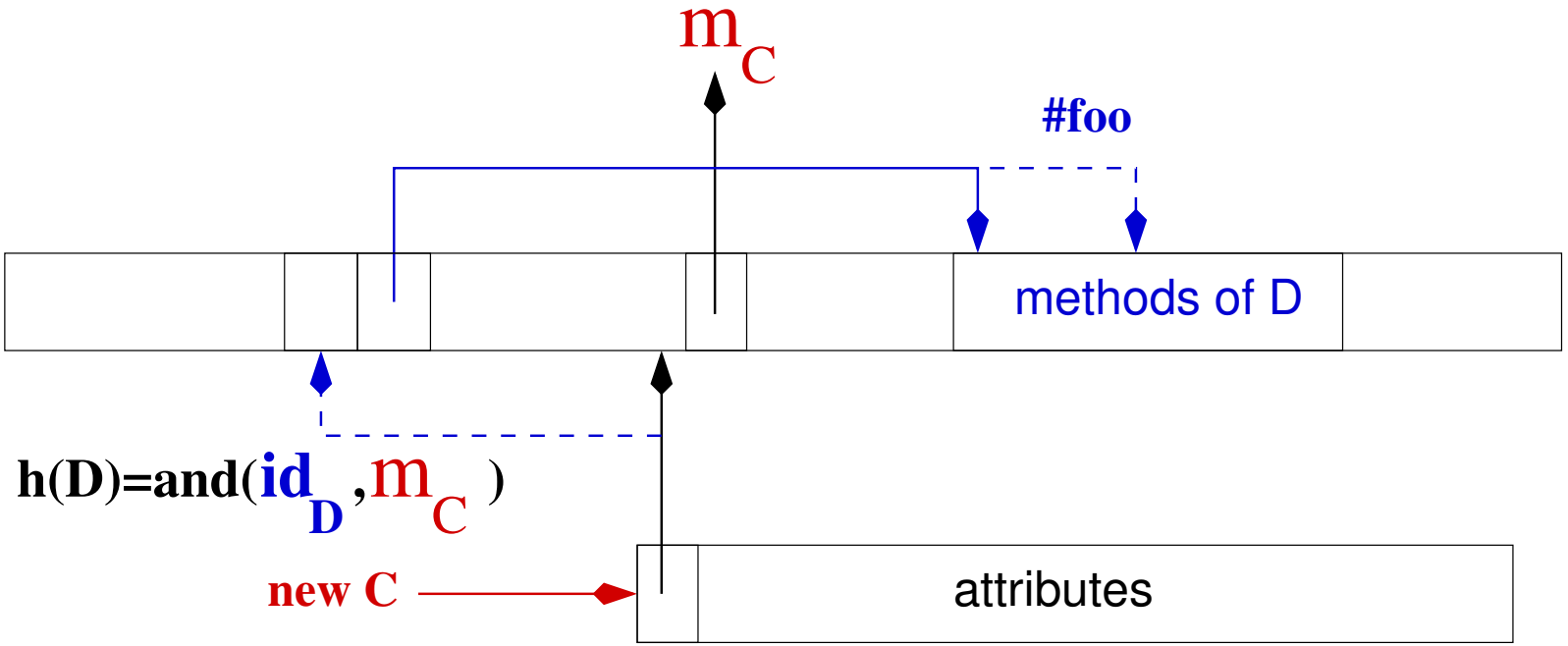




## Test de sous-typage

- o = instance d'une classe C inconnue,
- D est une interface possiblement implémentée par C,
- ☛ **o instanceOf D ?**

# Application aux interfaces Java

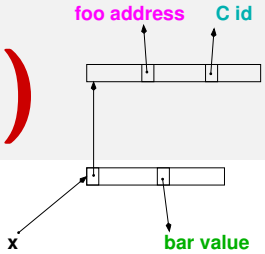


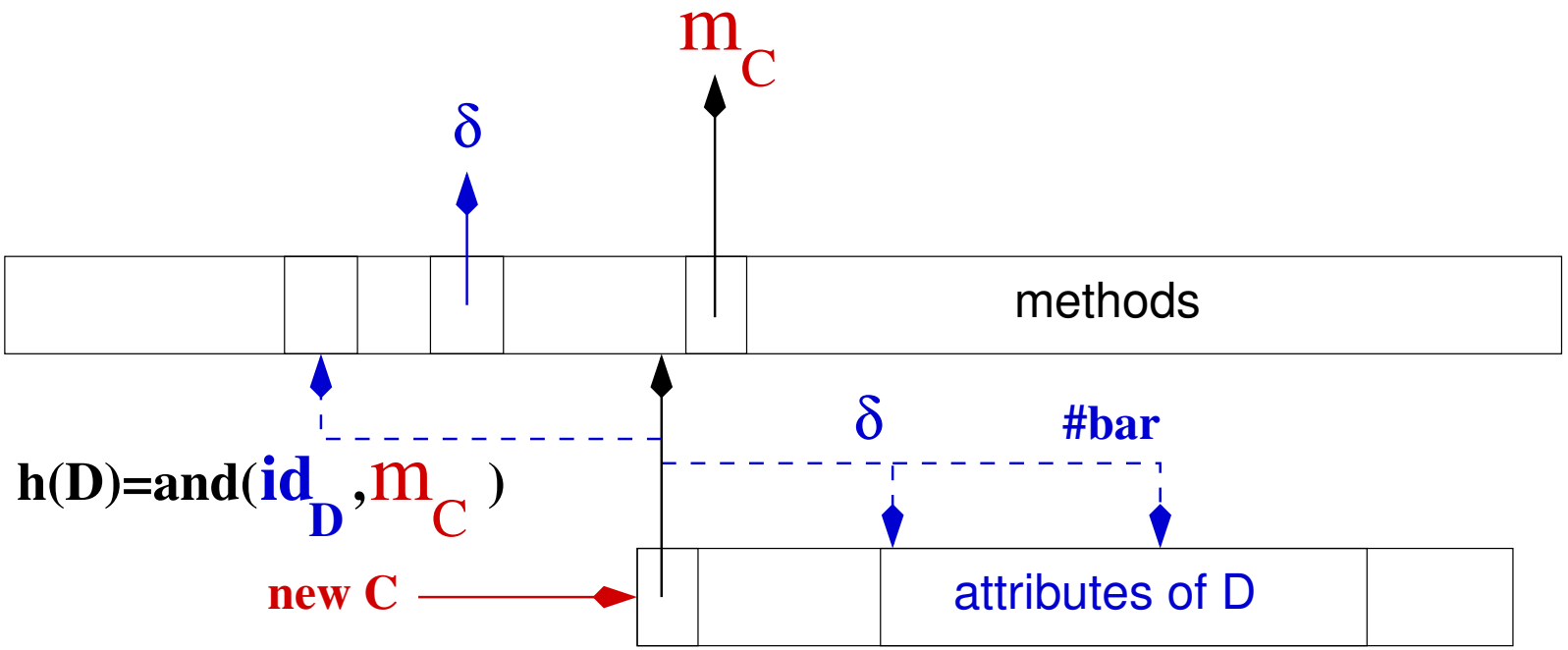




## Invokeinterface

- D o ; o.foo(arg)
- C <: D et D interface

# Application à l'héritage multiple (1/2)

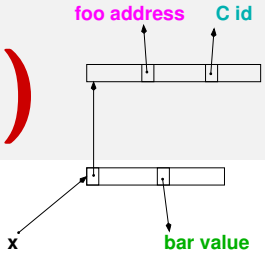



## Principe

- même implémentation qu'avec le sous-typage multiple
- + accès aux attributs par simulation d'accesseur

# Application à l'héritage multiple (2/2)




## Usage

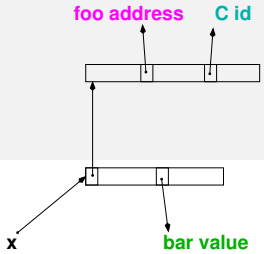
- surcoût important

☛ seulement si la position de la classe n'est pas invariante

## Compilation dynamique nécessaire

- avoir une position invariante est une propriété provisoire
- un site d'invocation peut muter
  du sous-typage simple au hachage parfait
- recompilations nécessaires, quand le chargement d'une classe "déplace" une de ses super-classes

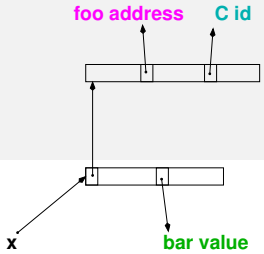
# Implémentation des objets (1/3)




## Héritage multiple et chargement dynamique

- les méthodes/attributs introduits par une classe sont groupés,
- les super-classes sont ordonnées,
- la condition du préfixe tient pour une super-classe directe
- position invariante pour la super-classe du préfixe,
- ☛ l'ordre des super-classes dépend de l'ordre de chargement.

# Implémentation des objets (2/3)



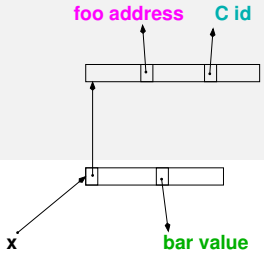

← ordre de chargement —

A

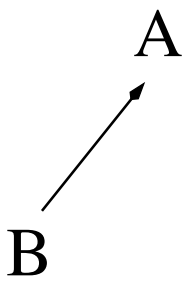
A

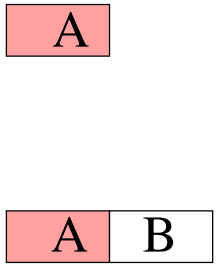
- $D$ vérifie la condition du préfixe vis-à-vis de $B$, pas de $C$,
- $C$ a plusieurs positions,
- les invocations de $C$ doivent être recompilées en chargeant $D$

# Implémentation des objets (2/3)



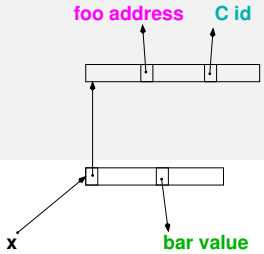

← ordre de chargement —



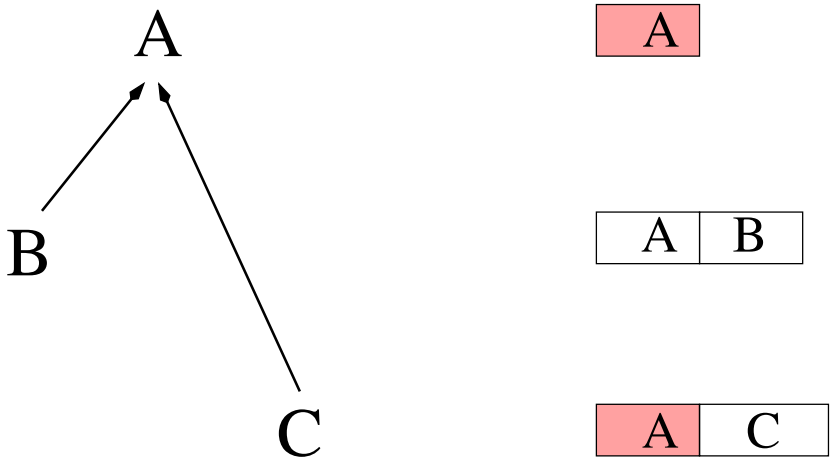




- $D$ vérifie la condition du préfixe vis-à-vis de $B$, pas de $C$,
- $C$ a plusieurs positions,
- les invocations de $C$ doivent être recompilées en chargeant $D$

# Implémentation des objets (2/3)

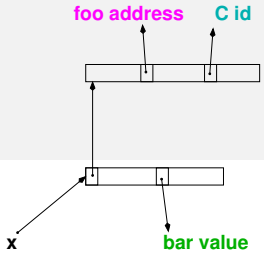


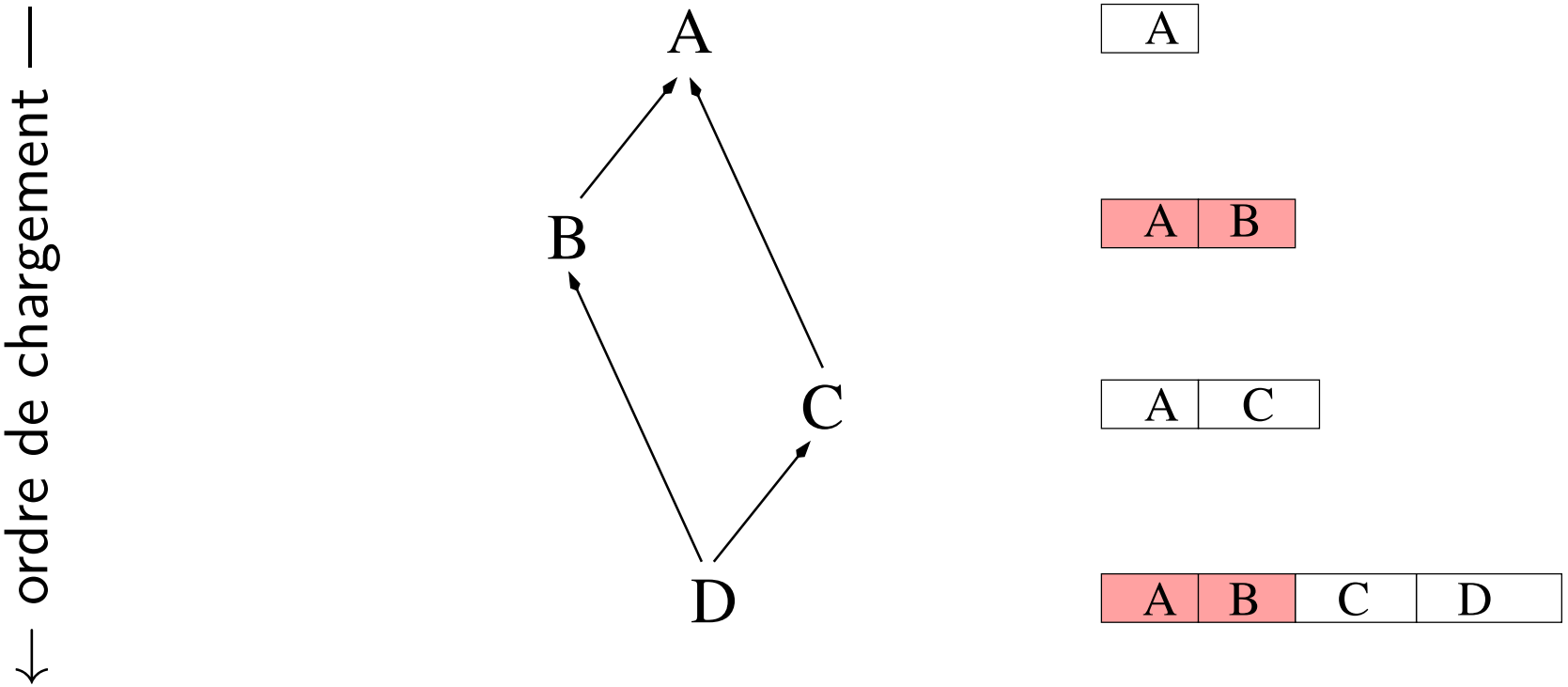

← ordre de chargement —




- $D$ vérifie la condition du préfixe vis-à-vis de $B$, pas de $C$,
- $C$ a plusieurs positions,
- les invocations de $C$ doivent être recompilées en chargeant $D$

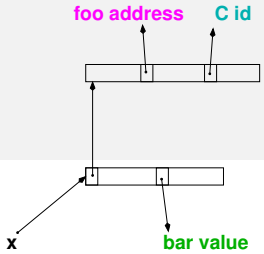
# Implémentation des objets (2/3)




- $D$ vérifie la condition du préfixe vis-à-vis de $B$, pas de $C$,
- $C$ a plusieurs positions,
- les invocations de $C$ doivent être recompilées en chargeant $D$ ☹

# Implémentation des objets (3/3)

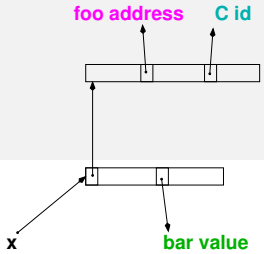



## Ressemble à l'approche trait/mixin

- avec super-classes primaires et secondaires,
  - primaire = classe = dans le préfixe ☛ position unique possible
  - secondaire = trait = hors du préfixe ☛ positions multiples
- cependant, ici,
  - la distinction n'est pas au niveau language,
  - mais au niveau implémentation,
  - et déterminée dynamiquement par l'exécution

☛ Implémentation asymétrique de l'héritage multiple symétrique

# Implémentation des objets (3/3)

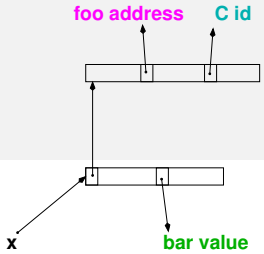



## Ressemble à l'approche trait/mixin

- avec super-classes primaires et secondaires,
  - primaire = classe = dans le préfixe ☛ position unique possible
  - secondaire = trait = hors du préfixe ☛ positions multiples
- cependant, ici,
  - la distinction n'est pas au niveau language,
  - mais au niveau implémentation,
  - et déterminée dynamiquement par l'exécution

☛ Implémentation asymétrique de l'héritage multiple symétrique

# Compilation dynamique

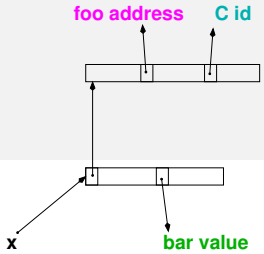



## Paresse et gloutonnerie

- compilation paresseuse des méthodes
  - le plus tard possible
- compilation gloutonne des sites
  - le plus efficacement possible (dans le contexte courant)
  - appel statique, sous-typage simple ou hachage parfait,
- recompilations possibles, paresseuses et gloutonnes
  - d'appels statiques au sous-typage simple ou hachage parfait,
  - du sous-typage simple au hachage parfait

LIRMM

# Evaluation

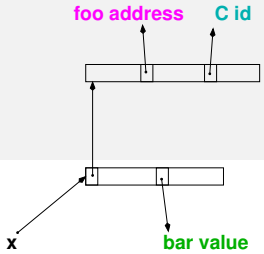



## Simulation par interprétation abstraite aléatoire

- variante aléatoire de Rapid Type Analysis
- statistiques sur le nombre de
  - classes, méthodes et attributs avec positions (in)variables,
  - sites d'invocation optimisés (ou pas),
  - (re)compilations de méthodes,
  - *thunks*.
- résultats plutôt satisfaisants

LIRMM

# Conclusion sur PH




## Avec PH

- l'héritage multiple ne coûte que si l'on s'en sert

  l'héritage multiple symétrique au prix du sous-typage multiple

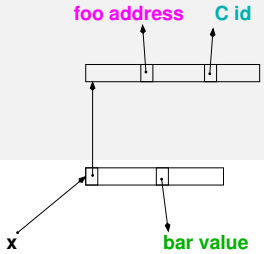
  devrait être testé dans une vraie machine virtuelle

## La preuve que c'est bien

- breveté par le leader mondial des VM

  googler "perfect hashing method dispatch"

# Conclusion sur PH




## Avec PH

- l'héritage multiple ne coûte que si l'on s'en sert

  l'héritage multiple symétrique au prix du sous-typage multiple

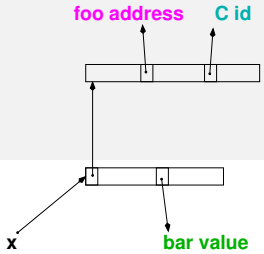
  devrait être testé dans une vraie machine virtuelle

## La preuve que c'est bien

- breveté par le leader mondial des VM

  googler "perfect hashing method dispatch"

# Conclusion sur PH




## Avec PH

- l'héritage multiple ne coûte que si l'on s'en sert

  l'héritage multiple symétrique au prix du sous-typage multiple

  devrait être testé dans une vraie machine virtuelle

## La preuve que c'est bien

- breveté par le leader mondial des VM

  googler "perfect hashing method dispatch"

LIRMM

# Plan



LIRMM

# Jvm, Java, premiers barreaux de Scala

L'un des langages récents les plus intéressants

## Compile vers la Jvm

- hérite de nombreuses spécifications de Java
- bénéficie de la machine d'exécution et de la bibliothèque Java

☛ réduction considérable de l'effort de R&D

## Nombreuses extensions

- héritage multiple (traits)
- variance à la définition, types virtuels
- fermetures, *pattern matching*, ...

LIRMM

# Scala traîne le boulet de la Jvm (1/2)

## Héritage des défauts de la machine d'exécution

### Effacement de type

- échec du typage sûr
- nombreuses inefficacités (*boxing* + *casts*)

### Héritage multiple asymétrique

Complication conceptuelle,

- pour les programmeurs de base,
- pour les formalisations, la compilation,
- pour l'efficacité.

# Scala traîne le boulet de la Jvm (2/2)

- une partie de l'effort de R&D consacrée à ces défauts
- Scala n'est pas le langage idéal

# Scala traîne le boulet de la Jvm (2/2)

- une partie de l'effort de R&D consacrée à ces défauts
- Scala n'est pas le langage idéal

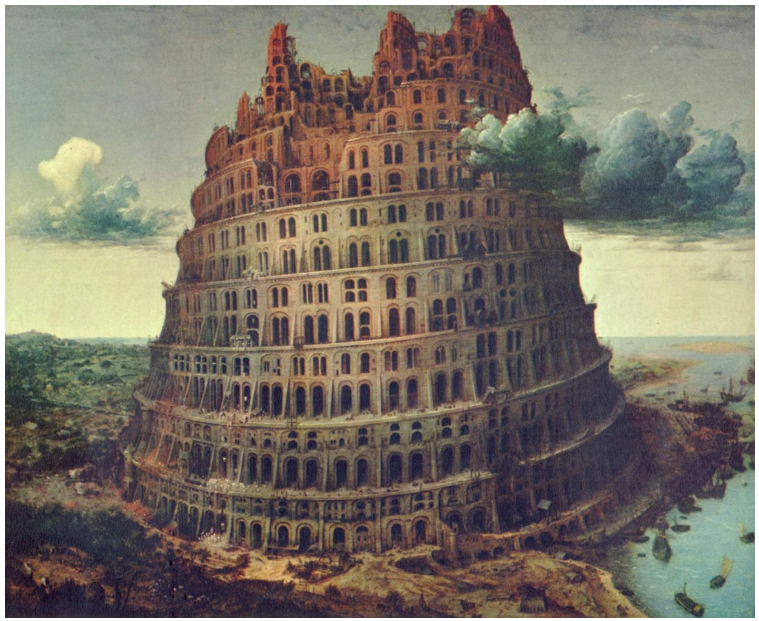
# Scala : bilan a contrario

## Le langage idéal

- n'est pas simple (hélas)
- est idéalement cohérent
- est idéalement documenté
- est idéalement appareillé (les orthèses du programmeur)

# Plan





LIRMM

# Les causes ?

## Problème général avec le temps

- fuite en avant
- faible capitalisation
- succès trop rapides
- contraintes de rétro-compatibilité
- mauvaise anticipation des conséquences des choix

## Approches peu scientifiques

- solutions ad hoc
- sémantique par l'implémentation

☞ le monde académique a désinvesti la conception de langages

LIRMM

# Les causes ?

## Problème général avec le temps

- fuite en avant
- faible capitalisation
- succès trop rapides
- contraintes de rétro-compatibilité
- mauvaise anticipation des conséquences des choix

## Approches peu scientifiques

- solutions ad hoc
- sémantique par l'implémentation

☛ le monde académique a désinvesti la conception de langages

# Les causes ?

☛ absence d'idéal Platonicien ...

# Quels espoirs d'évolution ?

## Malheureusement faibles

- évolutions mineures sans impact sur la rétro-compatibilité
- par exemple, en Java :
  - variance à la définition, à la Scala
  - gestion des conflits de nom par qualification, à la C#

  mais toujours pas en Java 8 ! !
- ☛ corriger l'effacement de type nécessite une évolution majeure
  - des spécifications et implémentation de la Jvm,
  - des programmes Java/Scala existants

☛ La solution est dans de nouveaux langages !

LIRMM

# Quels espoirs d'évolution ?

## Malheureusement faibles

- évolutions mineures sans impact sur la rétro-compatibilité
- par exemple, en Java :
  - variance à la définition, à la Scala
  - gestion des conflits de nom par qualification, à la C#

  mais toujours pas en Java 8 ! !

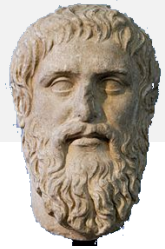
☛ corriger l'effacement de type nécessite une évolution majeure

- des spécifications et implémentation de la Jvm,
- des programmes Java/Scala existants

☛ La solution est dans de nouveaux langages !

LIRMM

# Les besoins



## Spécifications “canoniques” des langages à objets

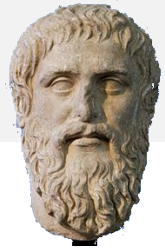
- pour la généricité,
- pour l'héritage multiple,
- ☛ liste à compléter ...

## Systèmes d'exécution

- similaires à la JVM (ou à CIL/CLR),
- sans effacement de type,
- pour l'héritage multiple,
- basés sur le hachage parfait,
  ou tout autre technique complètement *scalable*.

LIRMM

# Perspectives personnelles



## Le langage NIT

- développé à l'UQAM autour de Jean Privat,
- à la suite de ses travaux au LIRMM (langage PRM)
- avec généricité non-effacée, héritage multiple, types virtuels, ...
- un interprète et plusieurs compilateurs statiques,
- *work in progress*

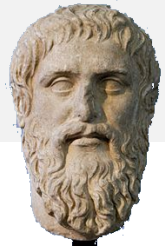
☛ Pas l'idéal Platonicien, juste un moyen d'y parvenir

## La machine virtuelle NIT

- le projet démarre ...

LIRMM

# Perspectives personnelles



## Le langage NIT

- développé à l'UQAM autour de Jean Privat,
- à la suite de ses travaux au LIRMM (langage PRM)
- avec généricité non-effacée, héritage multiple, types virtuels, ...
- un interprète et plusieurs compilateurs statiques,
- *work in progress*

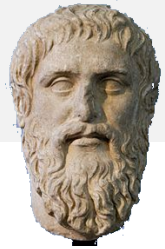
☛ Pas l'idéal Platonicien, juste un moyen d'y parvenir

## La machine virtuelle NIT

- le projet démarre ...

LIRMM

# Perspectives personnelles



## Le langage NIT

- développé à l'UQAM autour de Jean Privat,
- à la suite de ses travaux au LIRMM (langage PRM)
- avec généricité non-effacée, héritage multiple, types virtuels, ...
- un interprète et plusieurs compilateurs statiques,
- *work in progress*

☛ Pas l'idéal Platonicien, juste un moyen d'y parvenir

## La machine virtuelle NIT

- le projet démarre ...

LIRMM

avec la participation de :

Achille (et ses talons), Aristote,
Babel, Bruegel (et leur tour),
Camus (Albert), Einstein (Albert),
Galilei (Galileo), Greg, Homère,
Jacob (et son échelle), Kush (Vladimir),
Patrocle, Peyo, Platon, Socrate,
Schtroumpf (à lunettes),
Talon (Achille), ...

`http://nitlanguage.org`

(à suivre)