Une machine virtuelle pour un langage à objets en héritage multiple

Julien Pagès

MaREL, LIRMM, CNRS & Université de Montpellier

Semindoc, LIRMM, 28 Janvier 2015

Le projet

Thèse commencée en 2013 (avec Roland Ducournau) : Étude d'une machine virtuelle en héritage multiple basée sur le hachage parfait

Objectifs principaux

- Preuve de faisabilité d'une telle machine virtuelle
- Expérimenter des techniques d'implémentation dans les machines virtuelles
- Étudier les protocoles d'optimisation dynamiques
- Avoir une machine virtuelle (relativement) efficace

On cherche à avoir un modèle d'exécution similaire à celui de Java.

Les langages à objets

Les objets : environ 50 ans aujourd'hui.

Éléments de base

- Classe
- Objets
- Encapsulation
- Envoi de message
- Polymorphisme

→ Un paradigme adapté à la modélisation du monde réel.

La compilation

Définitions

1. Pratique : Transformation d'un langage source en un langage cible
2. Scientifique : Étude des langages de programmation et de leur systèmes d'exécution

Trois grandes familles de systèmes d'exécution :

1. Interpréteur
2. Compilateur
3. Machine virtuelle

Les machines virtuelles

Caractéristiques usuelles d'une machine virtuelle

- Chargement dynamique (monde ouvert)
- Compilation à la volée
- Gestion automatique de la mémoire

Un peu d'histoire

- Les bases de la compilation à la volée sont posées depuis LISP : 1966
- Smalltalk et Self ont introduits les machines virtuelles : 1980
- Java puis C♯ ont popularisés ces systèmes

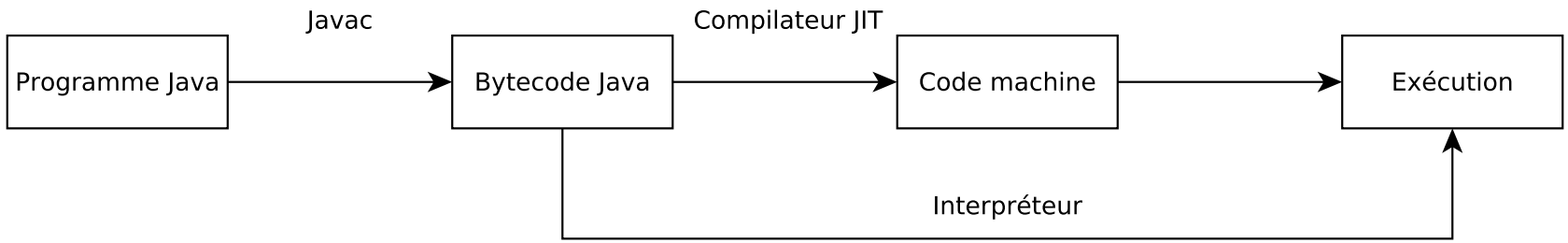
Java et sa machine virtuelle

Java a été créé en 1995 par James Gosling. La machine virtuelle Java (JVM) est à l'origine une spécification.

Caractéristiques

- Langages à objets en typage statique
- Langage à héritage simple mais avec des interfaces (déclarant uniquement des signatures de méthodes) en sous-typage multiple
- Un programme Java fonctionne sur la machine virtuelle Java (JVM)

Modèle d'exécution de Java :



Ce que l'on veut faire différemment de Java

Dans le langage

- Langage à héritage multiple
- Une implémentation de la généricité non-effacée
- ...

Dans la machine virtuelle

- Meilleures implémentations des mécanismes objets
- Protocoles d'optimisations simple pour des performances correctes

Ce que l'on veut faire différemment de Java

Dans le langage

- Langage à héritage multiple
- Une implémentation de la généricité non-effacée
- ...

Dans la machine virtuelle

- Meilleures implémentations des mécanismes objets
- Protocoles d'optimisations simple pour des performances correctes

Cette combinaison de caractéristiques, en chargement dynamique n'a jamais été implémenté

Étude des techniques utilisées

Les performances des langages sont liées à l'implémentation efficace de trois mécanismes :

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

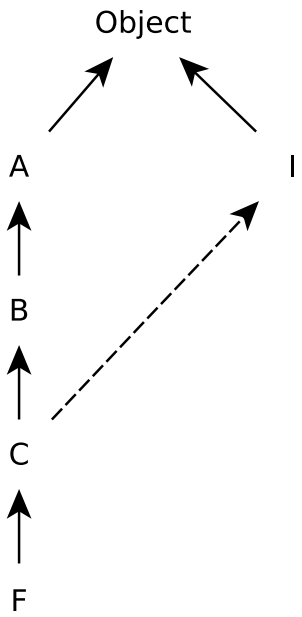
Dans le cas de Java, seules les interfaces sont en sous-typage multiple

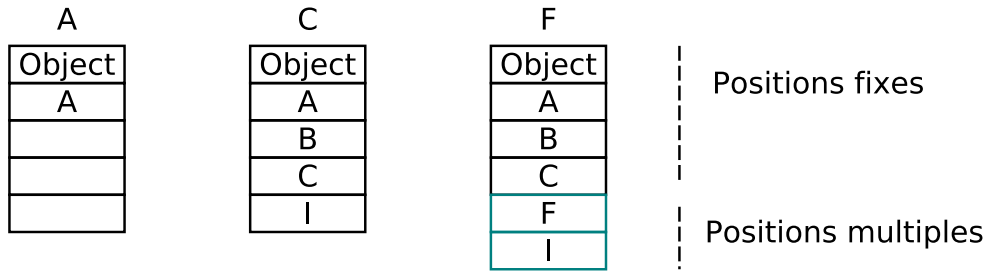
Les deux mécanismes complexes à implémenter :

- Appel de méthode d'une interface (opération *invokeinterface*)
- Test de sous-typage par rapport à une interface

Test de sous-typage : difficultés

Le sous-typage multiple fait perdre la position invariante dans les tables de types





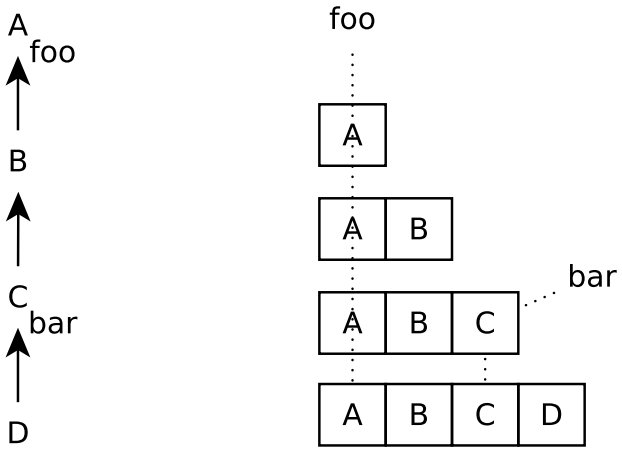
Appel de méthode : difficultés

Appels de méthodes dans la JVM : 4 opérations bytecode

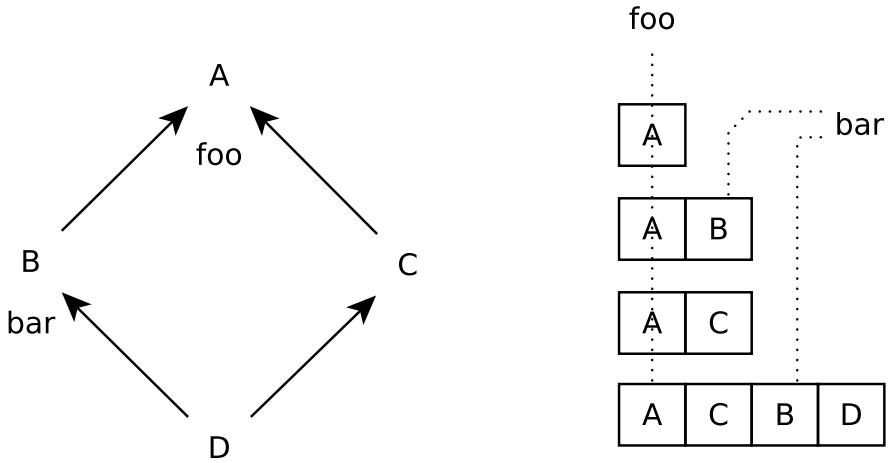
- `invokestatic`, `invokespecial`, `invokevirtual` : héritage simple
- `invokeinterface` : héritage multiple

Appel de méthode pour Java :

Sous-typage simple :



Sous-typage multiple :



Techniques utilisées dans les JVM

Bilan

- Beaucoup de techniques différentes utilisées
- Peu de techniques connues d'un point de vue algorithmique
- L'implémentation efficace est difficile

Aucune ne remplit les conditions suivantes:

- Temps constant
- Espace linéaire dans la taille de la relation de spécialisation
- Compatible avec l'héritage multiple
- Compatible avec le chargement dynamique
- Séquence de code courte

Techniques utilisées dans les JVM

Bilan

- Beaucoup de techniques différentes utilisées
- Peu de techniques connues d'un point de vue algorithmique
- L'implémentation efficace est difficile

Aucune ne remplit les conditions suivantes:

- Temps constant
- Espace linéaire dans la taille de la relation de spécialisation
- Compatible avec l'héritage multiple
- Compatible avec le chargement dynamique
- Séquence de code courte

$\rightarrow$ sauf le hachage parfait[1]

Les protocoles d'optimisations dans les JVM existantes

Constat : systèmes parfois très efficaces mais peu décrits et très complexes

Les protocoles d'optimisations dans les JVM existantes

Constat : systèmes parfois très efficaces mais peu décrits et très complexes

Des systèmes faisant du profilage

- Compteurs : permet de compter le nombre d'appels à une méthode directement dans son code
- Échantillonnage : on stoppe l'exécution pour inspecter la pile

Une méthode souvent appelée est recompilée plus efficacement

Les protocoles d'optimisations dans les JVM existantes

Constat : systèmes parfois très efficaces mais peu décrits et très complexes

Des systèmes faisant du profilage

- Compteurs : permet de compter le nombre d'appels à une méthode directement dans son code
- Échantillonnage : on stoppe l'exécution pour inspecter la pile

Une méthode souvent appelée est recompilée plus efficacement

Optimisations principales

1. Inlining, expansion en ligne, le code de l'appelé dans l'appelant
2. Court-circuit de l'appel de méthode

Optimisations gardées : garantir la correction tout en optimisant

Outils

Développer une telle machine virtuelle est un projet très important

Outils

Développer une telle machine virtuelle est un projet très important

La réutilisation d'outils est obligatoire !

Outils

1. Un langage : Nit[2]
2. Le hachage parfait
3. Une aide au développement : machine virtuelle développée à partir de l'interpréteur Nit

Outils

Développer une telle machine virtuelle est un projet très important

La réutilisation d'outils est obligatoire !

Outils

1. Un langage : Nit[2]
2. Le hachage parfait
3. Une aide au développement : machine virtuelle développée à partir de l'interpréteur Nit

nitvm : une machine virtuelle pour Nit, en Nit

Le langage Nit

Un peu d'histoire

- Développé à partir de 2004 au LIRMM sous le nom de PRM par Jean Privat.
- Plusieurs thèses autour de ce langage : Jean Privat, Floréal Morandat
- Devenu Nit en 2008, il est maintenant développé à l'UQAM (Canada)

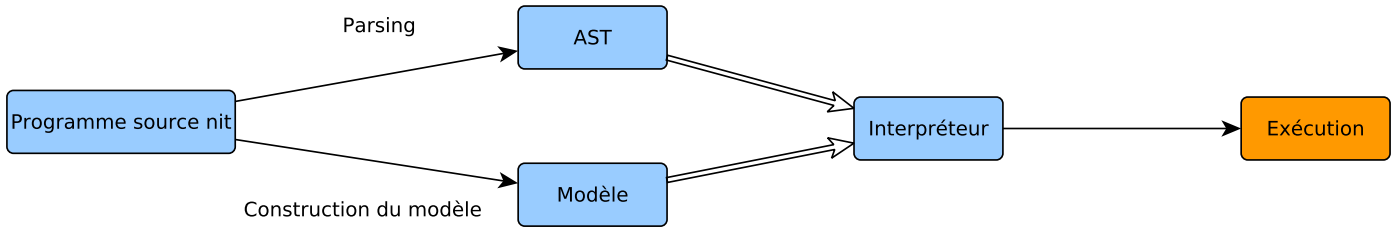
Le langage

- Un langage à objets
- Héritage multiple
- Typage statique (n'est pas toujours lié au paradigme objets)
- Modules, raffinements de classes
- Généricité non-effacé, types virtuels...

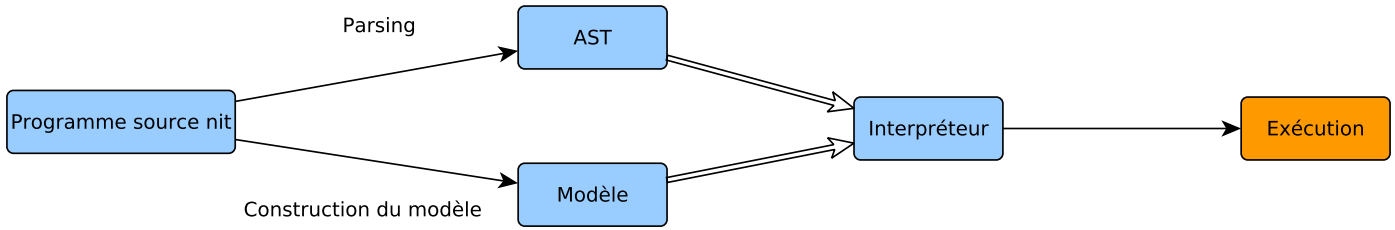
Le langage Nit : exemple

```
module fibonacci

    redef class Int
    fun fibonacci: Int
    do
        if self < 2 then
            return 1
        else
            return (self-2).fibonacci + (self-1).fibonacci
        end
    end
end

print "Fibo(10) = " + 10.fibonacci
```

Fonctionnement de l'interpréteur Nit



Fonctionnement de l'interpréteur Nit



Fonctionnement

1. Construction globale du modèle et de l'AST
2. Parcours de l'AST en appelant le service approprié selon le nœud
3. L'exécution est directement produite (pas de compilation du code)

Interpréteur vs machine virtuelle

Caractéristiques de l'interpréteur

- Pas de compilation
- Pré-traitement global sur le programme en entrée : AST, modèle
- Gestion automatique de la mémoire
- Très simple mais peu efficace

Interpréteur vs machine virtuelle

Caractéristiques de l'interpréteur

- Pas de compilation
- Pré-traitement global sur le programme en entrée : AST, modèle
- Gestion automatique de la mémoire
- Très simple mais peu efficace

Éléments manquants par rapport à une machine virtuelle

- Chargement dynamique
- Protocoles d'optimisations
- Compilation à la volée du code
- Implémentation efficace des mécanismes objets

Comment développer une telle machine virtuelle

Théorème : *ce qui fonctionne avec l'interpréteur fonctionne avec la nitvm*

Méthodologie

- Développement par héritage/raffinement de l'interpréteur
- Ce qui n'est pas remplacé fonctionne encore
- Utilisation du C pour les structures de bas-niveau depuis Nit

Se rapprocher du modèle d'une machine virtuelle

1. Chargement dynamique : chargement de classe à la première instanciation
2. Compilation à la volée : construction paresseuse des structures

Comment avoir une machine virtuelle efficace ?

Implémentation efficace des mécanismes objet

Plusieurs implémentations différentes :

1. Héritage simple → rapide : implémentation SST
2. Héritage multiple → lent : hachage parfait

Comment avoir une machine virtuelle efficace ?

Implémentation efficace des mécanismes objet

Plusieurs implémentations différentes :

1. Héritage simple → rapide : implémentation SST
2. Héritage multiple → lent : hachage parfait

Mais aussi

- Un modèle paresseux ne compile que le nécessaire : gain "gratuit"

Comment avoir une machine virtuelle efficace ?

Implémentation efficace des mécanismes objet

Plusieurs implémentations différentes :

1. Héritage simple → rapide : implémentation SST
2. Héritage multiple → lent : hachage parfait

Mais aussi

- Un modèle paresseux ne compile que le nécessaire : gain "gratuit"
- Des optimisations pendant l'exécution

Le hachage parfait comme technique d'implémentation de l'héritage multiple

Hachage parfait

- Le hachage parfait est un hachage sans collision d'un ensemble d'entiers
- Le masque de hachage est calculé pour former une fonction injective entre l'ensemble d'identifiants et la fonction binaire *AND*

Le hachage parfait comme technique d'implémentation de l'héritage multiple

Hachage parfait

- Le hachage parfait est un hachage sans collision d'un ensemble d'entiers
- Le masque de hachage est calculé pour former une fonction injective entre l'ensemble d'identifiants et la fonction binaire *AND*

Numérotation parfaite

- Optimisation de l'espace utilisé pour les tables de hachage
- Lors de l'attribution d'un nouvel identifiant : on choisit celui qui convient le mieux aux identifiants déjà présents dans les super-classes

Le hachage parfait comme technique d'implémentation de l'héritage multiple

Hachage parfait

- Le hachage parfait est un hachage sans collision d'un ensemble d'entiers
- Le masque de hachage est calculé pour former une fonction injective entre l'ensemble d'identifiants et la fonction binaire *AND*

Numérotation parfaite

- Optimisation de l'espace utilisé pour les tables de hachage
- Lors de l'attribution d'un nouvel identifiant : on choisit celui qui convient le mieux aux identifiants déjà présents dans les super-classes

Le hachage parfait remplit les cinq conditions (temps constant, espace linéaire...)

Structures utilisées pour l'implémentation

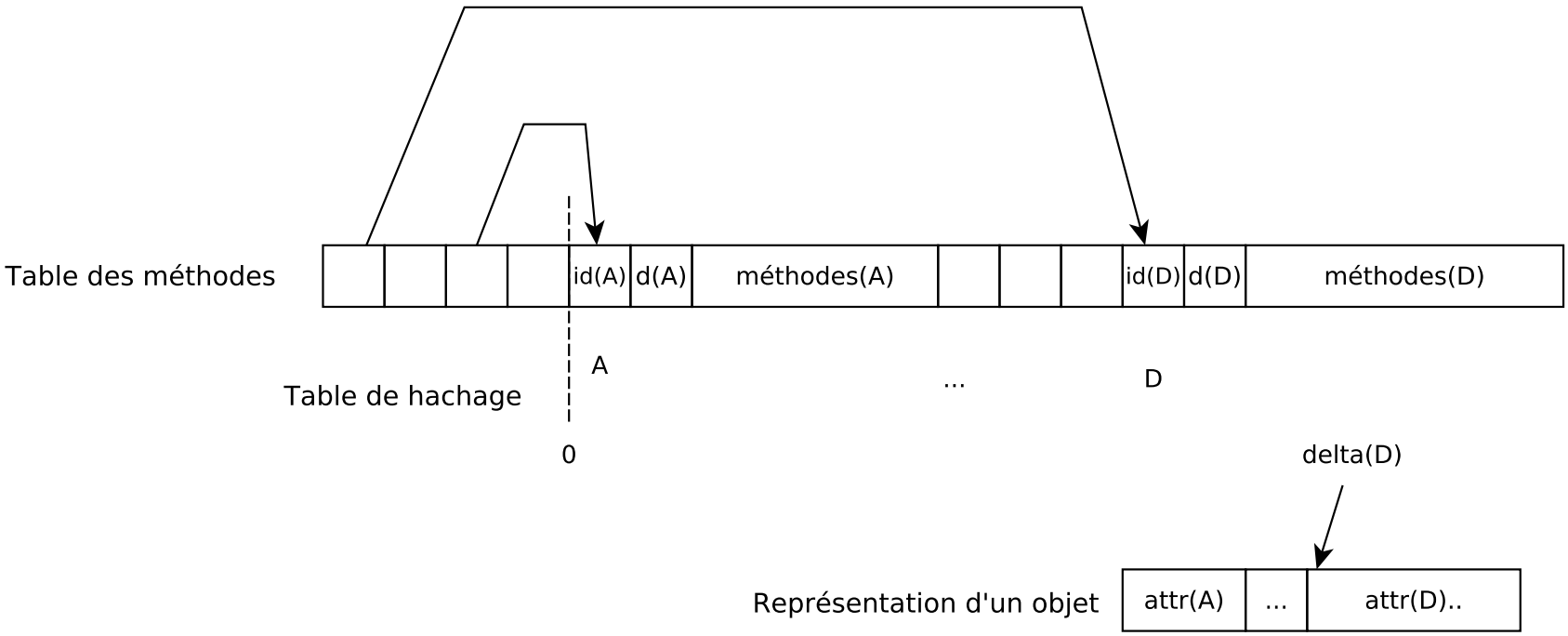
Prérequis :

- Groupement des méthodes/attributs introduits par une même classe
- Ordonnancement des superclasses
- Allocation contiguë des structures d'une classe/objet

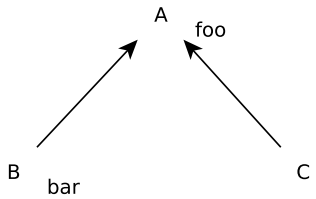
Structures utilisées pour l'implémentation

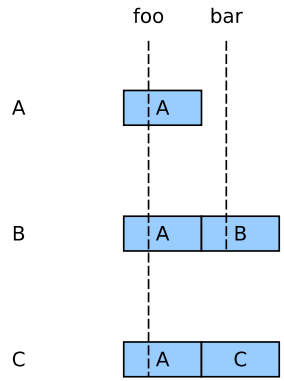
Prérequis :

- Groupement des méthodes/attributs introduits par une même classe
- Ordonnancement des superclasses
- Allocation contiguë des structures d'une classe/objet



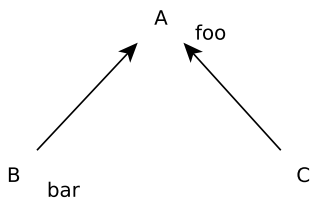
Implémentation des méthodes en héritage simple

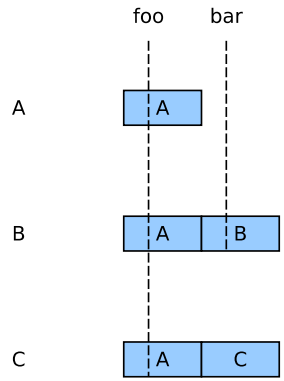





```
A x = new A()
x.foo
```

Implémentation des méthodes en héritage simple

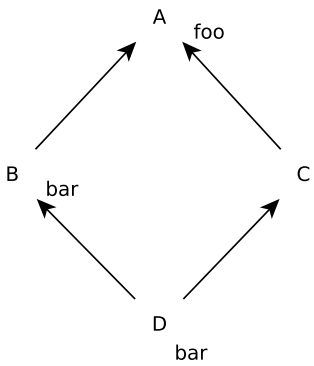


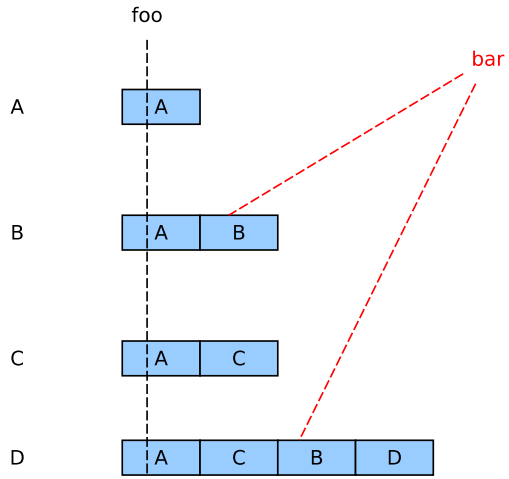



```
A x = new A()
x.foo
```

```
load [x + #tableOffset], table
load [table + #fooOffset], methAddr
call #methAddr
```

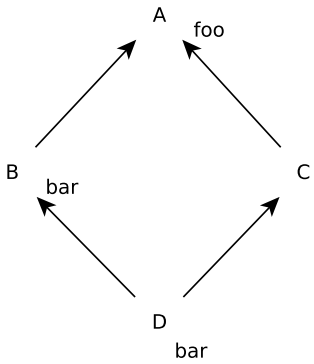
Implémentation des méthodes en héritage multiple

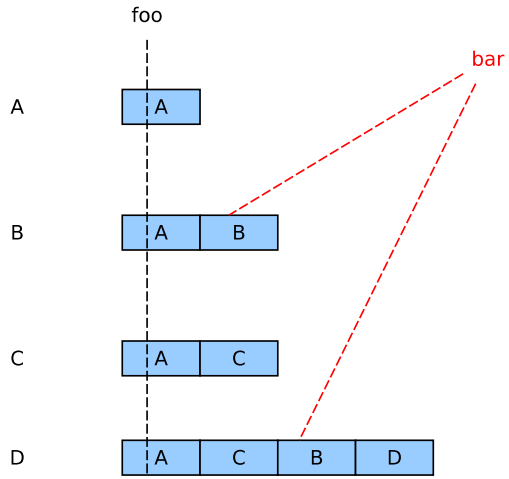





```
B x = new D()
x.bar
```

Implémentation des méthodes en héritage multiple

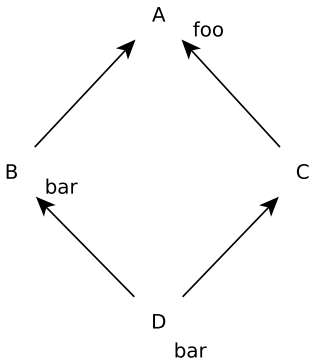


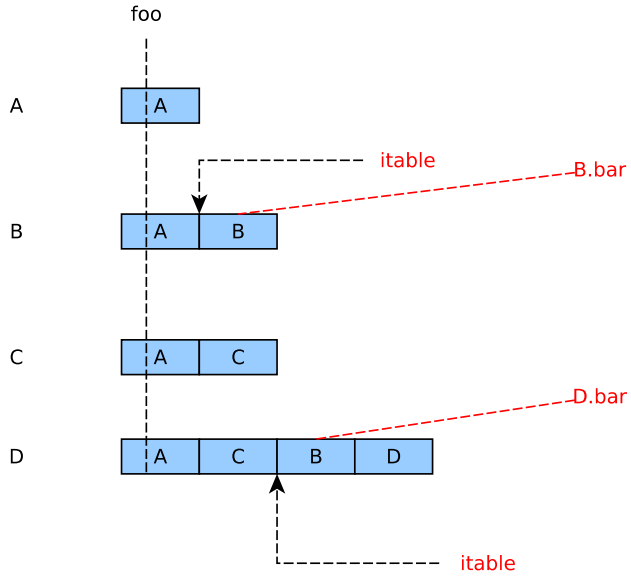



```
B x = new D()
x.bar
```

```
load [x + #tableOffset], table
load [table + #barOffset], methAddr
call #methAddr
```

Implémentation des méthodes en héritage multiple



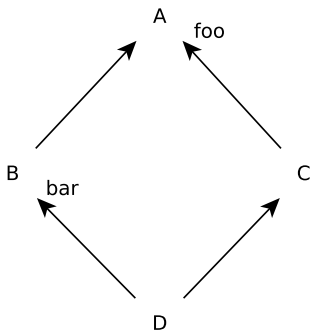


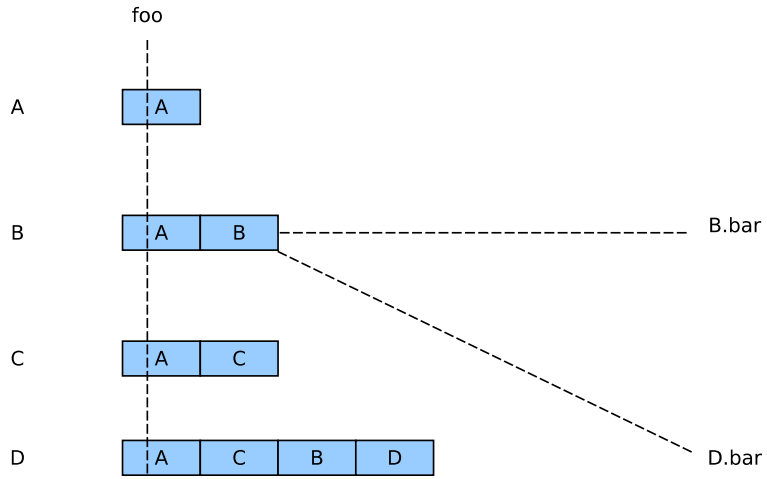

```
B x = new D()
x.bar
```

```
# séquence de hachage parfait
itable = offset du bloc concerné

load [x + #tableOffset], itable
load [itable + #barOffset], methAddr
call #methAddr
```

Les optimisations avec le chargement dynamique

Problème : On ne peut pas prédire l'avenir



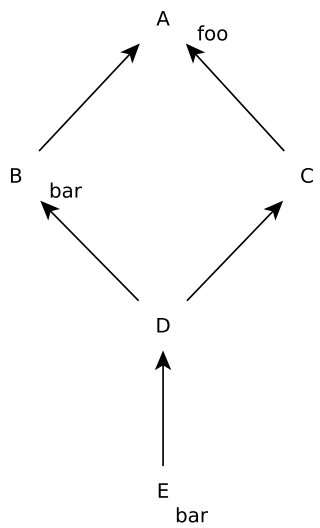


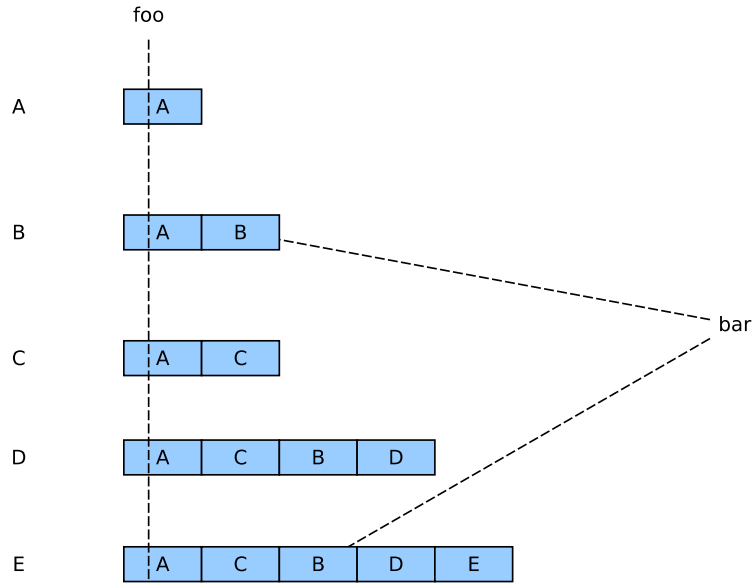
Si la méthode `bar` de `B` *n'est jamais redéfinie* :

- On compile tous les appels à `bar` par un appel direct à `bar` de `B`

Les optimisations avec le chargement dynamique

Problème : On ne peut pas prédire l'avenir





Si la méthode `bar` de `B` *est redéfinie* dans `E` :

- On recompile certains appels à `bar` par un appel en héritage multiple

Pré-optimisations

L'ordre des classes dans les tables des méthodes est important

- On préfère changer de place une classe qui a peu de méthodes
- Une classe ayant plusieurs superclasses directes en choisit une qui sera son préfixe

La règle du préfixe : position invariante pour la classe du préfixe

Pré-optimisations

L'ordre des classes dans les tables des méthodes est important

- On préfère changer de place une classe qui a peu de méthodes
- Une classe ayant plusieurs superclasses directes en choisit une qui sera son préfixe

La règle du préfixe : position invariante pour la classe du préfixe

Exemple : on calcule l'ordre des classes de `D`

- Si `B` a plus de méthodes que `C` :

```
A B C D
```

- Sinon :

```
A C B D
```

Protocole de compilation/recompilation : objectifs

Objectifs concrets

- Maximiser les appels directs et ceux en implémentation SST...
- Quitte à devoir recompiler certains sites de manière moins efficace au chargement d'une classe

Protocole de compilation/recompilation : objectifs

Objectifs concrets

- Maximiser les appels directs et ceux en implémentation SST...
- Quitte à devoir recompiler certains sites de manière moins efficace au chargement d'une classe

Aspects recherches

- Faire des protocoles de compilation/recompilation un objet d'étude
- Avoir des protocoles simples pour une efficacité correcte
- Trouver le protocole avec le meilleur compromis coût/efficacité

Expérimentations sur les protocoles

Expérimentations prévues

- Test de différentes techniques d'optimisations et de recompilation
- Recompilation en modifiant les noeuds d'AST, donc les implémentations
- Statistiques par comptage des nœuds de l'arbre syntaxique
- Comparaison des temps d'exécution avec différents protocoles (temps relatif)

Pistes de travail

Analyses statiques si possible plutôt que profilage

Quelques optimisations :

- Appel d'une méthode introduite dans `Object` : appel SST
- Site d'appel monomorphe : appel direct
- Site d'appel avec position invariante : appel SST
- Les autres sites d'appels : hachage parfait

Plus d'appels directs et SST que d'appels avec le hachage parfait.

Transformation dans la structure

Pistes de travail

- Structures d'exécution plus efficaces
- Encore plus de paresse dans le chargement
- Dérécursiver l'exécution : ajout de piles

Il est possible de travailler sur les entrées de la machine virtuelle et les aspects compilations

Une représentation intermédiaire pour Nit

Objectifs

- Préparer statiquement l'exécution : première phase de compilation
- Utilisable avant l'exécution et/ou pendant
- Chargement dynamique réellement paresseux en chargeant les fichiers au besoin
- Modèle plus réaliste et gain de performance

Une machine virtuelle idéale pour Nit

Caractéristiques

- Chargement le plus paresseux possible
- Compilation à la volée en code de bas niveau
- Implémentation efficace des mécanismes objets
- Protocoles d'optimisations peu complexes et efficaces
- Le code serait maintenable et permettrait les expérimentations

Bilan

État actuel du développement

- Simulation du chargement dynamique
- Structures efficaces (objets + exécution)
- Ordonnancement des superclasses
- Plusieurs implémentations des mécanismes objets : SST et PH

À venir dans les prochains mois

- Travail sur les protocoles de compilation/recompilation
- Langage ou représentation intermédiaire pour Nit

La machine virtuelle Nit permet d'exécuter des programmes Nit

Références

Ducournau, R., and Morandat, F.
Perfect class hashing and numbering for object-oriented implementation.
Software: Practice and Experience 41, 6 (2011), 661–694.

Privat, J.
Nit language.
http://nitlanguage.org/.

Pour aller plus loin

`http://nitlanguage.org/`

`https://github.com/privat/nit`
`https://github.com/jpages/nit`

Pour aller plus loin

`http://nitlanguage.org/`

`https://github.com/privat/nit`
`https://github.com/jpages/nit`

Merci de votre attention