Une machine virtuelle pour tester des protocoles de compilation/recompilation en héritage multiple

Julien Pagès

LIRMM, MaREL

Séminaire, LIRMM, 2 février 2016

Le projet

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

Deux principaux objectifs

- Développer une **machine virtuelle** pour un **langage à objets** en **héritage multiple**
- Étudier les **protocoles de compilation/recompilation** dans ce système
  - Les optimisations sont la clé de la performance des machines virtuelles
  - Mais souvent très peu décrites dans la littérature scientifique

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

Les machines virtuelles

Trois grandes familles de systèmes d'exécution : interpréteur, compilateur, machine virtuelle

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

Caractéristiques usuelles d'une machine virtuelle

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

Outils choisis

Contexte et outils

- Le langage Nit [3]
- Le hachage parfait [2] pour l'héritage multiple
- Développée à partir de l'interpréteur Nit
- Simulation de compilation à la volée du code

Le langage Nit

Historique

- 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)

Caractéristiques du langage

- Complètement orienté objet
- Héritage multiple (de classes)
- Typage statique
- Modules, raffinements de classes
- Généricité...

Une machine virtuelle à partir d'un interpréteur

L'interpréteur Nit existant

- Fonctionne en **monde fermé**
- Pas optimisé du tout, son code est simple et réutilisable
- Interprète les programmes à partir de l'AST (représentation arborescente du programme) décoré par le méta-modèle

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

La machine virtuelle pour Nit

Caractéristiques

- Fonctionne en chargement dynamique
- Simulation de compilation
  1. Numérotation des variables locales
  2. Calcul de l'algorithme SSA *Single-Static Assignment*
  3. Choix de la compilation des portions de code avant l'exécution

On a maintenant un système suffisamment réaliste pour effectuer des expérimentations sur les protocoles de compilation.

Implémentation des mécanismes objets

Trois mécanismes objets :

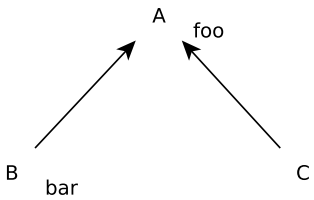
- Appel de méthode
- Accès aux attributs
- Test de sous-typage

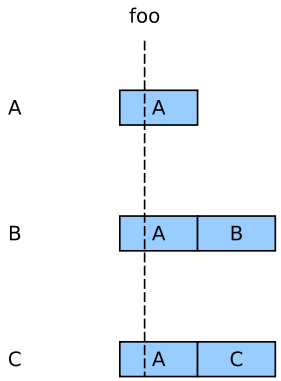
Implémentations

Du moins efficace au plus efficace :

- *Hachage parfait* (héritage multiple)
- SST (héritage simple)
- Statique
- *Inlining*

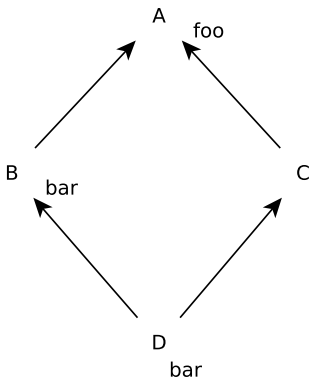
Héritage multiple et chargement dynamique

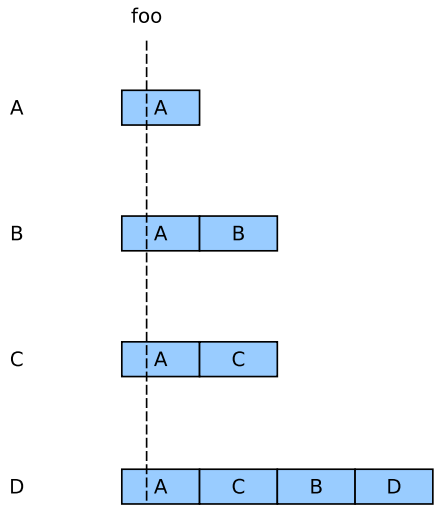




Situation d'héritage simple

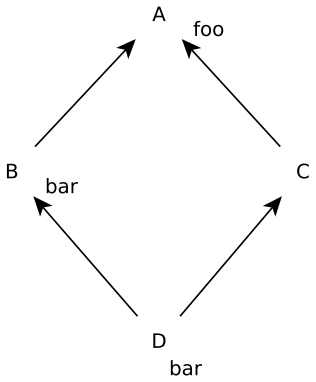
Héritage multiple et chargement dynamique

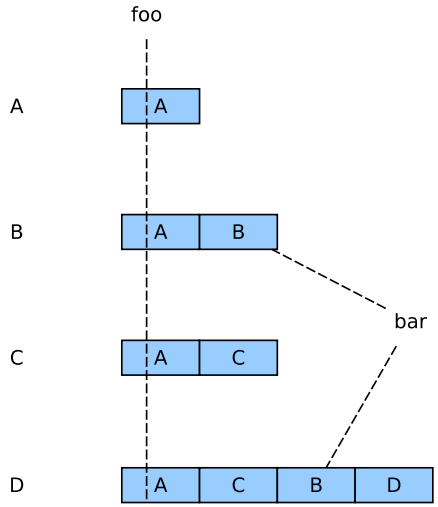




Situation d'héritage multiple

Héritage multiple et chargement dynamique





Problèmes :

- Une méthode (ou un attribut) a plusieurs positions dans les tables de méthodes
- On ne peut pas prévoir le chargement de `D` au début du programme

Définition

Un protocole est une boîte à outils (intelligente) composée d'éléments de trois catégories :

1. Collecte d'informations : analyses statiques, profilage de l'exécution
2. Optimisations : dévirtualisation, inlining
3. Réparations

Problème

Des chargements futurs peuvent invalider des optimisations précédentes

But

Trouver un protocole efficace, et pas trop coûteux en terme de réparations du code

Techniques de réparation du code

Techniques usuelles

- *Gardes* : une technique qui évite d'avoir à réparer le code, en gardant une version optimisée du code, et la version originelle.
- *On-stack replacement* : patch de l'adresse de retour de la méthode pendant son exécution.
- *Code patching* : patch local du code compilé.

Ces trois techniques fonctionnent mais il est préférable de les éviter (coût, complexité)

Préexistence

Définition

- Propriété introduite dans [1], est une propriété d'un receveur
- Un receveur préexistant assure qu'il n'y aura pas de recompilations du code pendant l'appel
- Évite d'utiliser les trois techniques précédentes

Exemple (avec une syntaxe à la Nit) :

```
fun foo(x: A)
do
    x.bar() // x est préexistant
end
```

Règles de préexistence

Préexistence originale

- Le receveur vient d'un paramètre (règle de préexistence originelle)
- Le receveur vient d'un attribut privé immutable cf `final`, `const`, `val` (s'applique mal à Nit)

La préexistence est uniquement de *valeur*.

Préexistence étendue

Définition

Un receveur est maintenant préexistant si son type est chargé

Exemple :

```
fun bar()
do
    return new A
end

fun foo(x: A)
do
    var z = bar
    z.bar() // z est préexistant
end
```

Préexistence étendue

Nouvelles règles de préexistence

Le receveur provient de :

- Retour d'un appel de méthode préexistant
- D'une lecture d'un attribut privé avec des types concrets (affecté uniquement par des *new*)
- D'un cast limité (sous ensemble des types de la cible)

Préexistence étendue

Optimisations des sites monomorphes

- Le type statique du receveur est final (et chargé)
- Le receveur provient d'un *new* d'une classe instanciée

Nouvelles règles de préexistence

Le receveur provient de :

- Retour d'un appel de méthode préexistant
- Lecture d'un attribut privé avec des types concrets (affecté uniquement par des *new*)

Préexistence étendue

Optimisations des sites monomorphes

- Le type statique du receveur est final (et chargé)
- Le receveur provient d'un *new* d'une classe instanciée

Nouvelles règles de préexistence

Le receveur provient de :

- Retour d'un appel de méthode préexistant
- Lecture d'un attribut privé avec des types concrets (affecté uniquement par des *new*)

La préexistence étendue est mutable.

Expérimentations

Benchmark utilisé : la machine virtuelle sur laquelle est exécuté l'interpréteur (avec un petit programme en entrée)

Statistiques

- Le nombre de sites objets de chaque catégorie (par implémentation)
- Les sites que l'on peut inliner, sans prendre en compte le critère de la taille du code
- Les sites préexistants

Deux méthodes :

- En cours d'exécution
- En fin d'exécution (tout le programme est chargé)

Résultats sur la préexistence

On cherche à *inliner* le plus possible de mécanismes objets (ou à défaut de faire des appels statiques)

Version originelle

	méthodes	attributs	casts	total
monomorph	4130	2267	157	6554
preexisting	3984	2040	275	6299
non preexisting	3163	1043	761	4967
total polymorph	7147	3083	1036	11266
preexistence rate	55%	66%	26%	55%

35 % de tous les sites sont préexistants avec les règles originelles.

Résultats sur la préexistence

On cherche à *inliner* le plus possible de mécanismes objets (ou à défaut de faire des appels statiques)

Version étendue

	méthodes	attributs	casts	total
monomorph	4130	2267	157	6554
preexisting	4455	2191	294	6940
non preexisting	2692	892	742	4326
total polymorph	7147	3083	1036	11266
preexistence rate	62%	71%	28%	61%

Amélioration

Le taux de préexistence global passe de 55% à 61%.
76% des sites optimisables (monomorphes + préexistants) en étendue.

Résultats préexistence étendue

Impact des différentes règles

	méthodes	attributs	casts	total	
Read	109	60	13	182	8%
New	21	0	0	21	70%
Call	364	91	7	462	17%
Subtype	2	2	0	4	5%
other	18	10	0	28	19%
total improved	514	163	20	697	14%

Conclusion

Résultats

- La machine virtuelle Nit est suffisamment développée pour étudier les protocoles
- Un modèle des protocoles a été implémenté
- Apport intéressant de la préexistence étendue

Dans le futur

- Mesurer les recompilations (coût du protocole) plus finement
- Étude des effets de l'*inlining* avec la préexistence

Références

Detlefs, D., and Agesen, O.
Inlining of virtual methods.
In *ECOOP'99*. Springer, 1999, pp. 258–277.

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/.

Merci de votre attention