

Techniques pour la génération dynamique de code

Philippe Wang

philippe.wang@etu.upmc.fr

Université Pierre et Marie Curie, Paris, France

Master de Recherche en Informatique,

Spécialité Science et Technologie du Logiciel - Algorithmique et Programmation

Résumé Cet article présente une étude générale de différentes techniques de génération dynamique de code, et met en évidence leurs coûts ainsi que leurs apports. Parmi les familles de techniques de génération dynamique de code, il y a principalement la compilation juste-à-temps - qui compose la majeure partie de l'article -, les modifications « en place » d'un programme - il s'agit souvent de spécialiser du code lors de l'exécution. D'autres applications plus ou moins exotiques existent. Par exemple, des études sont en cours pour implanter les fermetures à l'aide de génération dynamique de code, afin d'éviter l'approche classique qui consiste à construire des structures de données. Les techniques de génération dynamique de code sont indispensables aux besoins actuels en programmation, et ont un avenir certain puisque d'une part, certaines tâches (comme la réification ou la réflexion par exemple) ne peuvent être effectuées que dynamiquement, et d'autre part la plupart de ces techniques sont basées sur des heuristiques et des compromis, donc toujours perfectibles.

1 Introduction

Les paradigmes des langages de programmation sont nombreux. Citons-en quelques uns : d'assemblage, à objets, impératifs, fonctionnels, déclaratifs, concurrents, par contrats, synchrones, et par aspects. Tous les langages de programmation, quelques soient leurs paradigmes, ont un point commun fondamental qui est celui de décrire des calculs destinés à être exécutés par un (ou plusieurs) microprocesseur(s). Ces différents paradigmes de programmation permettent de choisir un langage de programmation en fonction de la nature du travail à réaliser. Actuellement, par exemple, les couches bas niveau des systèmes d'exploitation sont programmées principalement avec le langage C - qui est souvent vu comme un « assembleur portable » -, les interfaces graphiques suivent pour la plupart le modèle par objets, et les gestionnaires de contenus de sites internet sont également programmés par objets pour la plupart.

D'autre part, les fonctionnalités introduites ont toutes un coût en termes de performances, que ce soit à la compilation ou à l'exécution. Dans le but de réduire les coûts engendrés, l'on utilise différentes techniques. Les compilateurs (statiques) effectuent des optimisations basées sur des techniques d'analyses statiques, puisqu'il est possible de déterminer à l'avance (i.e. avant son exécution) le comportement de certaines parties de certains programmes. Par

exemple, le typage statique fort permet de s'affranchir des coûts liés aux vérifications de types à l'exécution. Quand aux techniques de génération dynamique de code, elles permettent notamment la création dynamique de morceaux de programmes.

Cette article présente une étude générale de différentes techniques de génération dynamique de code, et montre leurs nécessités pour les modèles de programmes réflexifs, puis les applications et les performances des techniques présentées.

Nous verrons dans un premier temps l'importance de l'existence de différents paradigmes de langages de programmation ainsi que leurs leurs répercussions sur les coûts. Dans un second temps, nous verrons la génération de code dans le cadre des chargements dynamiques de classes et modules, puis nous aborderons brièvement la sérialisation. Ensuite nous présenterons la compilation juste-à-temps, suivie de la compilation basée sur des modèles pour la spécialisation dynamique. Enfin, nous verrons une technique en cours d'étude : l'implantation des fermetures basée sur la génération dynamique de code.

2 Les langages de programmation et les conséquences de leurs paradigmes

Les différents paradigmes des langages de programmation sont apparus pour différentes raisons. Pour faire la promotion d'un langage de programmation, on parle de ses « performances »¹, de son expressivité, de sa facilité de maintenance, de sa portabilité, etc.

Aucun langage n'est arrivé à satisfaire tous les critères d'exigences imaginés et imaginables, ce qui fait qu'il n'existe pas de « meilleur langage de programmation » universellement parlant.

PHP² se veut facile d'apprentissage et rapide pour le développement de sites web. Java³ se veut portable. C et C++ se veulent performants. Objective Caml⁴ se veut performant et très expressif. Haskell⁵ est un langage purement fonctionnel. Perl⁶ se veut rapide à écrire. (Nous ne parlerons pas de tous les paradigmes des langages)

Chacun en fait à sa tête et assume plus ou moins bien les coûts engendrés par leurs fonctionnalités ou « arguments de vente ». PHP est un langage interprété et relativement « lent ». Java a acquis une réputation de lenteur à cause de son interprète de code-octet et tente de gagner en performances en expérimentant différentes techniques de compilations juste-à-temps. Son concurrent direct, Microsoft .Net, est un adversaire à sa mesure. C et C++ restent les langages qui sont en tête dans les performances. Objective Caml souffre d'un manque de

¹ en fait, des performances des compilateurs et/ou machines virtuelles associés

² <http://www.php.net/>

³ <http://java.sun.com/>

⁴ <http://www.ocaml.org/>

⁵ <http://www.haskell.org/>

⁶ <http://www.perl.org/>

présence dans l'industrie et de promotion, et reste un langage de recherche, même s'il gagne en parts de marché. Haskell est peu performant, son mode d'évaluation retardée n'est gagnant que dans de rares cas.

Chaque trait de « haut niveau » qu'on donne à un langage de programmation a son coût qu'on essaie parfois de rattraper par des techniques de compilation plus complexes. Par exemple, dans les langages à objets, les appels de méthodes (ou envois de messages) sont en général très coûteux à cause de la nécessité de déterminer à quelle classe appartient une méthode appelée. Les langages fonctionnels savent très bien optimiser les appels récursifs terminaux, mais la création des fermetures lors des applications partielles reste chère.

3 Génération et chargement dynamiques

3.1 Motivations

Dans le cadre d'une application ayant besoin de modifier son comportement dynamiquement afin d'introduire de nouveaux types sans être arrêtée (propriété de réflexion), l'usage de la génération dynamique de code est importante. En Java, cette caractéristique est essentielle dans le cadre des moteurs de pages JSP [24]. Ainsi, lorsque le développeur modifie le contenu d'une page JSP, le serveur génère l'implantation d'une *servlet* liée à la page en question. Cette *servlet* est ensuite chargée par le conteneur et devient aussitôt disponible pour l'application. Il est ainsi possible, à tout moment, de créer et compiler de nouvelles *servlets* sans avoir à arrêter le serveur Web. [13]

3.2 Fonctionnement

En C# .Net, la classe `Assembly` représente les classes chargées dynamiquement.

La méthode `Assembly.LoadFrom` 1 permet de charger dynamiquement un fichier.

Name	Description
<code>Assembly.LoadFrom (String)</code>	<i>Loads an assembly given its file name or path. Supported by the .NET Compact Framework.</i>
<code>Assembly.LoadFrom (String, Evidence)</code>	<i>Loads an assembly given its file name or path and supplying security evidence.</i>
<code>Assembly.LoadFrom (String, Evidence, Byte[], AssemblyHashAlgorithm)</code>	<i>Loads an assembly given its file name or path, security evidence hash value, and hash algorithm.</i>

FIG. 1. La méthode `Assembly.LoadFrom` en C# .Net

En Java, la classe `ClassLoader` 2 permet de charger dynamiquement un fichier de code-octet.

```
Class<?> loadClass(String name)  
Loads the class with the specified binary name.
```

FIG. 2. La méthode loadClass de la classe ClassLoader en Java

Le premier problème posé par le chargement dynamique de code est la cohérence du programme. Java et .Net utilisent un vérificateur afin de garantir que le code chargé ne brise pas les règles élémentaires (typage, gestion mémoire, ...). Ensuite, se pose le problème des performances en temps de calculs. Il faut bien sûr garder à l'esprit que toutes ces opérations dynamiques coûtent plus chères que les opérations statiques. En effet, la première étape prenant du temps est la génération même du code à charger : si on produit directement du code-octet, c'est plus difficile mais potentiellement plus rapide que la production de code de langage de haut niveau puisque ce dernier serait à passer au compilateur pour le traduire en code octet. L'intérêt alors est de récupérer, rentabiliser, d'une façon ou d'une autre le temps passé à la génération du code.

3.3 Schémas d'utilisation

Un schéma d'utilisation simple (fig. 3) de génération dynamique de code est l'architecture client-serveur où le serveur se charge de générer les programmes pour le client. Cela permet au serveur de rendre le client totalement dépendant du serveur, le client ne pouvant pas exécuter le programme sans le serveur.

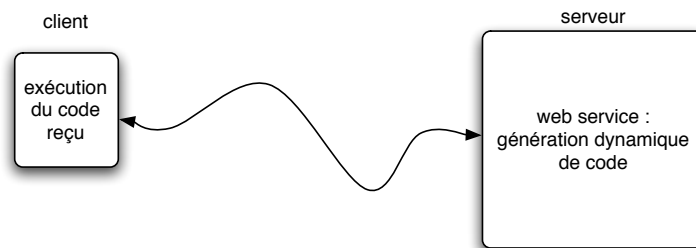


FIG. 3. Schéma de dépendance du client au serveur

Un autre schéma d'utilisation est le calcul distribué. Un serveur se charge de générer les programmes correspondant à des morceaux de programmes, et une batterie de clients se chargent d'effectuer les calculs et d'envoyer les résultats.

4 La sérialisation

Il est également possible d'utiliser les fonctions de sérialisation pour faire des applications distribuées.

Un ensemble d'informations peut être un programme. Un programme est un ensemble d'informations. Sortons un instant du cadre théorique de la sémantique des langages de programmation. Parlons des jeux-vidéos modernes qui utilisent le réseau internet[15]. Qu'ils aient une architecture client-serveur ou pair-à-pair, différents programmes échangent des informations. Ces informations sont générées dynamiquement par les programmes et transitent sous forme « sérialisée » d'un programme à l'autre. Revenons dans un cadre plus général et frôlons un moment la technologie Java RMI⁷[14]. Cette technologie permet de faire des appels distants de méthodes à travers le réseau et ainsi partager des calculs sur plusieurs machines.

La sérialisation est une forme de génération dynamique de code et pose un certain nombre de problèmes techniques, et c'est pour cela que nous en parlons. En effet, la sérialisation permet d'écrire des données sur disque, pour pouvoir les recharger ultérieurement, ou d'envoyer des données à travers le réseau, et l'on veut bien sûr pouvoir récupérer les données, donc désérialiser.

Revenons à présent sur le modèle des jeux-vidéo pour illustrer le lien entre « sérialisation » et « génération dynamique de code ».

Considérons deux joueurs, appelons-les classiquement Alice et Bob, possédant des personnages évoluant dans un même monde virtuel[15]. Alice fabrique un objet unique décrit par des propriétés. Pour que Bob prenne connaissance de cet objet, Alice envoie les informations à Bob à travers le réseau. Deux choix se montrent pour la sérialisation. L'une consiste à générer dans un format spécial les données décrivant l'objet, par exemple en XML⁸. L'autre consiste à utiliser les fonctionnalités de sérialisation du langage de programmation utilisé. Ce qui fait que d'un côté on prime la portabilité ainsi que la sécurité, et de l'autre côté la facilité.

Pour vérifier la cohérence d'une donnée au format XML, on utilise les grammaires DTD⁹. Pour les programmes embarquant les types à l'exécution, on vérifie le contenu en même temps que les types. Pour les programmes statiquement typés oubliant les types à l'exécution, il y a parfois possibilité de les reconstruire partiellement pour vérifier la cohérence[21].

5 Compilation juste-à-temps

5.1 Compilation juste-à-temps *versus* interprète

La compilation juste-à-temps est utilisée pour améliorer les performances des langages compilés vers du code-octet en traduisant un programme pour

⁷ (Java Remote Method Invocation)

⁸ Extensible Markup Language (XML) <http://www.w3.org/XML/>

⁹ Document Type Definition

machine virtuelle en programme pour la machine réelle utilisée, autrement dit de l'assembleur de machines virtuelles. Ce procédé consiste à traduire les instructions de machines virtuelles en instructions de machine réelle. Cela se traduit en général par un gain de performances lorsque le coût de la traduction est amorti par le gain à l'exécution du résultat produit.

Ce procédé a été initié au début des années 80 par Smalltalk-80 - le pionnier de cette technique. Aujourd'hui, la plate-forme .Net l'utilise, un certain nombre d'implantations de la machine virtuelle Java sont également basées sur ce modèle. Bien sûr, cette technique est utilisée par d'autres langages et plate-formes. Par exemple, l'INRIA l'a expérimenté en 2004 pour le langage Objective Caml ; il s'agit du projet OCamlJitRun[3].

L'utilisation d'un interprète de code-octet par Sun pour Java lui a valu une réputation de lenteur ; depuis, Sun a fait bien des avancées pour les performances de sa machine virtuelle, notamment en implantant la compilation juste-à-temps.

5.2 Schéma de fonctionnement

La figure 4 présente un schéma de fonctionnement général de la compilation juste-à-temps (des composants peuvent être enlevés ou ajoutés).

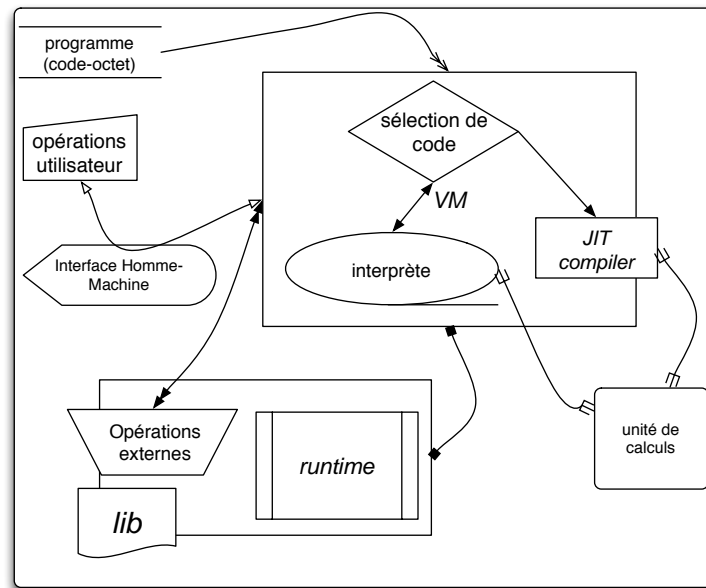


FIG. 4. Schéma de fonctionnement général de la compilation juste-à-temps

La première étape pour la machine virtuelle est de prendre en entrée un programme (code-octet). Ensuite, si la machine virtuelle comporte un interprète, une sélection de code est faite à l'aide d'un procédé d'instrumentation permettant de « surveiller » les actions du programme afin de compiler juste-à-temps vers du code machine (natif), c'est le cas de nombreuses machines virtuelles Java. Dans le cas où il n'y a pas d'interprète, la machine virtuelle se contente de compiler juste-à-temps le programme (code-octet) vers du code machine (natif), c'est le cas de la plate-forme .Net et d'OCamlJIT.

5.3 Recompilation « simple »

La compilation juste-à-temps simple consiste à compiler juste-à-temps les parties rencontrées lors du parcours du code-octet. Les grandes zones qui ne sont jamais exécutées ne sont pas recompilées car on ne passe pas dedans. Les petites morceaux de programme non exécutés sont recompilés pour l'arithmétique des adresses. C'est ce qui est fait par OCamlJIT, et dans un cadre plus industriel et plus connu, par la plate-forme .Net [10] de Microsoft.

5.4 Recompilation sélective

La compilation juste-à-temps sélective [8] est plus complexe et demande d'instrumenter le code-octet afin de détecter les zones les plus exécutées pour déterminer une heuristique demandant à recompiler et à optimiser les parties les plus souvent exécutées. Cette technique se base sur des mesures faites sur un ensemble de programmes et ne peut pas être universelle, tout comme les algorithmes de ramasse-miettes.

5.5 OCamlJIT

En 2004, l'INRIA développe un compilateur juste-à-temps[3] pour l'interprète de code-octet du langage Objective Caml, pour les architectures x86 (linux), PPC (linux) et Sparc (32bit). Après avoir modifié légèrement certains points du générateur de code-octet, un compilateur juste-à-temps entièrement compatible avec l'interprète de code-octet existant a pu être développé et testé. Les mesures de performances ont montré qu'on gagnait généralement un facteur 2. Certains programmes pouvant être exécuté jusqu'à 5 fois plus rapidement, et d'autres étant perdant à cause du processus de traduction de code juste-à-temps vers du code natif qui n'est pas amorti.

Les contraintes de départ posées par le cahier des charges indiquaient une compatibilité complète entre OCamlJIT et la machine virtuelle (`ocamlrun`) existante. En particulier, la pile et le tas devaient être les mêmes pour OCamlJIT et `ocamlrun`. Cela a impliqué un partage important de code source (en C pour le *runtime*, en OCaml pour le compilateur). Les valeurs OCaml ont dû garder la même représentation.

Le procédé utilisé par OCamlJIT est la recompilation naïve : une boucle principale (autour un grand `switch` C avec une branche par instruction de code-octet) scanne le code-octet et le traduit vers du code natif. Cependant, les grands morceaux de code jamais exécutés ne sont jamais traduits car la traduction se fait sur demande.

L'utilisation de la pile OCaml au lieu de la pile C a fait que les performances d'OCamlJIT restent bien moins bonnes que celles d'`ocamlOpt`¹⁰. Un problème de gestion de la mémoire a été rencontré : l'espace pris par le code généré n'est pas récupéré par le ramasse-miettes d'Objective Caml, il faut donc le récupérer explicitement.

Finalement, le code produit pour la machine virtuelle actuelle d'Objective Caml est pensé pour être compact et facile à interpréter, et se base sur l'utilisation de la pile (caractéristique partagée avec la machine virtuelle Java) est remise en cause car la plupart des opérations requièrent un accès mémoire sur la pile. Il est pensé qu'une machine virtuelle basée sur l'utilisation des registres pourrait être plus performante, sauf pour les processeurs à peu de registres tels que les x86.

Une autre idée pour Objective Caml serait de garder une représentation de plus haut niveau (un langage intermédiaire un niveau au dessus) et laisser le système compiler entièrement vers du code natif.

5.6 Sun Java et Microsoft .Net

Pour palier au problème du temps passé à la compilation lors de l'exécution, une solution proposée consiste à compiler vers du code natif statiquement avant l'exécution, comme l'a fait Toba¹¹ pour Java JDK 1.1, mais cela sacrifie le caractère de portabilité d'une part, et entraîne un temps de compilation qui n'est pas forcément amorti par la suite. Cependant, il est possible de garder un environnement d'exécution, comme le font Caffeine¹²[25] et Harissa¹³.

Les travaux menés sur Java et les résultats obtenus sur la plateforme .Net laissent à penser que le choix de Microsoft qui est de compiler systématiquement juste-à-temps est le bon, puisque nombre de mesures de performances placent .Net (très) légèrement devant Sun Java.

Selon les informations que l'on peut trouver sur la prochaine version de Sun Java (version 6), la compilation juste-à-temps sera mieux intégrée que dans les versions précédentes.

5.7 Limitation des machines virtuelles

Les travaux sur la compilation juste-à-temps d'Objective Caml ont montré quelques problèmes de performances liées à au code-octet de la machine vir-

¹⁰ `ocamlOpt` est le compilateur natif d'Objective Caml

¹¹ Toba est un compilateur de code-octet Java vers C (devenu cependant obsolète) <http://www.cs.arizona.edu/projects/sumatra/toba/>

¹² Caffeine est un compilateur de code-octet Java vers du code natif

¹³ Harissa est un compilateur de Java vers C

tuelle. Des travaux [11] menés par Microsoft Research sur la compilation de C vers la plate-forme .Net ont montré certaines limitations de l'assembleur de leur machine virtuelle, telles que celle engendrée par l'absence des équivalents de `long jmp` et `set jmp` dans leur CLR¹⁴.

On peut également noter la difficulté de la gestion des exceptions dans les machines virtuelles de Java et .Net. [28] présente notamment des travaux effectués sur la prédiction des rattrapeurs d'exceptions.

On se demande alors s'il devient nécessaire de changer le langage cible, en le rapprochant d'un langage machine - ce qui privilégierait indéniablement une catégorie de processeurs¹⁵ qui serait celui se trouvant être le plus proche du langage choisi - ou en l'éloignant, donc en compilant vers un langage de plus haut niveau éventuellement plus long à compiler vers du code natif, mais en expectative plus efficace et plus facile à désassembler.

6 Compilation utilisant des modèles pour la spécialisation dynamique

Les modules et les classes génériques ou polymorphes savent appliquer des traitements sur plusieurs types de données différents. Un coût supplémentaire est alors engendré par ces fonctionnalités puisqu'il faut discriminer la bonne opération en fonction du type de donnée reçu. Il existe la détection statique ainsi que la détection dynamique. Statiquement, il est parfois possible de savoir tout de suite si telle ou telle autre fonction ou méthode sera appelée sur un seul type de données. Dans ce cas, on compile l'appel de la fonction spécialisée au lieu de la fonction générique. Seulement, dans certains cas, il est impossible de détecter statiquement ces possibilités de spécialisation. Cela arrive par exemple lorsqu'une fonction polymorphe est encapsulée dans une autre fonction elle aussi polymorphe 5.

En revanche, lors de l'exécution, il est possible d'instrumenter le code et de regarder avec quels types la fonction est le plus appelée et ainsi faire une spécialisation dynamique, ce qui n'est pas non plus sans coût.

6.1 Avec des annotations du programmeur

Il est aussi possible de mettre en place un mécanisme basé sur les annotations du programmeur afin de détecter les zones à optimiser dynamiquement. Ainsi, à l'exécution, lorsqu'on rencontre une zone dynamique, selon qu'on soit déjà passé par là ou pas, on applique les optimisations prévues ou bien on passe directement au code optimisé 6. La figure 7 présente un exemple d'annotations expérimentées pour le langage C.

¹⁴ *Common Language Runtime* est le nom du langage de Microsoft pour la machine virtuelle de la plate-forme .Net

¹⁵ il existe un classement des processeurs en deux catégories : les RISC (microprocesseurs à jeu d'instruction réduit, *Reduced instruction set computer* en anglais) et les CISC

Soit (=) un opérateur infix polymorphe :

```
val (=) : ∀α.α → α → bool
```

- spécialisation (monomorphisation) statique du (=)


```
let int_equal : int → int → bool = (=)
```

 Il est possible de spécialiser statiquement la fonction (=) appelée.
- spécialisation statique impossible du (=)


```
rec all_equal = function
  | [] → true
  | a :: b :: tl → a = b && all_equal (b :: tl)
  | _ :: _ → true
```

```
val (all_equal) : ∀α.α list → bool
```

 Il n'est pas possible de prévoir à l'avance avec quels types de données la fonction (=) sera appelée.

FIG. 5. Monomorphisation d'une fonction polymorphe

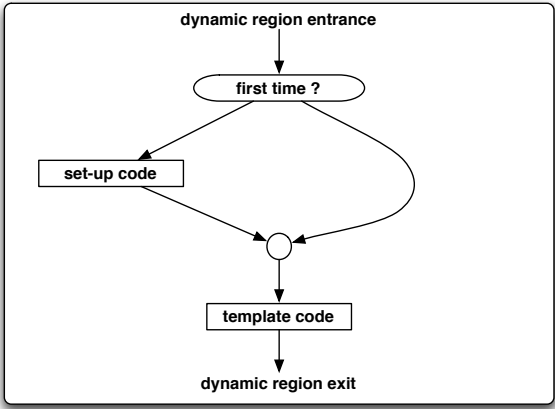


FIG. 6. Schéma de compilation partielle dynamique [4]

```

cacheResult cacheLookup(void *addr, Cache *cache) {
    dynamicRegion(cache) { /* cache is runtime constant */
        unsigned blockSize = cache->blockSize;
        unsigned numLines = cache->numLines;
        unsigned tag = (unsigned) addr / (blockSize * numLines);
        unsigned line = ((unsigned) addr / blockSize) % numLines;
        setStructure **setArray = cache->lines[line]->sets;
        int assoc = cache->associativity;
        int set;
        unrolled for (set = 0; set < assoc; set++) {
            if (setArray[set]dynamic->tag == tag)
                return CacheHit;
        }
        return CacheMiss;
    } /* end of dynamic region */
}

```

les mots-clefs noirs en gras sont les annotations

FIG. 7. Annotations du programmeur (langage C) [4]

7 Implantation des fermetures basée sur la génération dynamique de code

Des travaux récents [7][6] (2006) ont été menés pour expérimenter l'implantation des fermetures basée sur la génération dynamique de code, à l'université technique de Berlin par Martin Grabmüller.

La compilation classique des langages fonctionnels crée certaines indirections lors de la création de fermeture pour les applications partielles, en construisant une structure de donnée (fig. 8) pour la représenter. Classiquement, la

Soit $add = \lambda x. \lambda y. x + y$

- x est une variable libre dans l'abstraction intérieure
- la représentation de la fonction va de paire avec l'environnement au moment de la création
- l'environnement contient une valeur pour chaque variable libre de la fonction

La fermeture est ainsi construite :

$$add\ 2 \rightarrow (\lambda x. \lambda y. x + y)\ 2 \rightarrow \lambda y. x + y, \{x = 2\}$$

FIG. 8. Technique classique d'implantation des constructions de fermetures lors des applications partielles

(microprocesseurs à jeu d'instruction étendu, *Complex instruction set computer* en anglais)

structure de donnée est composée d'une liste de variables (autrement dit un environnement) et d'un pointeur de code (vers la fonction). [7] présente une implantation de techniques envisagées (voire préjugées) il y a quelques années comme inefficaces. Cependant la donne a changé suite aux évolutions technologiques, ce qui permet de laisser envisager un gain possible dans ces techniques d'implantation des fermetures. L'idée est alors de supprimer ces indirections en créant dynamiquement des fonctions complètes au lieu de fermetures.

8 Conclusion

Nous avons montré différentes techniques de génération dynamique de code appliquées à de nombreux domaines. La génération et le chargement dynamiques de classes et la sérialisation sont les techniques les plus abouties et les moins complexes à mettre en oeuvre. La compilation juste-à-temps, en étude depuis plus d'une quinzaine d'années, devient peu à peu « indispensable » pour les langages compilés pour des machines virtuelles dès que les performances ont un poids important. La génération dynamique de code n'a pas fini de trouver des domaines d'application et d'expérimentation, que ce soit pour des calculs répartis ou pour des propriétés réflexives des langages de programmation.

Références

1. Jacques Malenfant : *Architectures Logicielles pour l'Autoadaptabilité Dynamique*. Cours. Université Pierre et Marie Curie, Paris (2006).
<http://www-master.ufr-info-p6.jussieu.fr/>
2. Basile Starynkevitch : *OCAMLJIT a faster Just-In-Time Ocaml Implementation*. MetaOCaml Workshop 2004.
3. Basile Starynkevitch. *OcamljitRun software*.
<http://cristal.inria.fr/~starynke/ocamljit.html>
4. Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad : *Fast, Effective Dynamic Compilation*. Department of Computer Science and Engineering, University of Washington. (1996)
5. D. Engler : *VCODE : A Retargetable, Extensible, Very Fast Dynamic Code Generation System*, PLDI (1996).
<http://citeseer.ist.psu.edu/engler96vcode.html>
6. Martin Grabmüller : *A Generic Model of Functional Programming With Dynamic Optimization*, Technische Universität Berlin, 2006.
7. Martin Grabmüller : *Implementing Closures using Run-time Code Generation*, Technische Universität Berlin, research report, 2006.
8. Gilles Fedak : *Recompilation sélective à la volée du Byte code Java*, rapport DEA, 1997.
9. Sun Microsystems : *Le Langage Java*.
<http://java.sun.com/>

10. Microsoft Research : La Plateforme .Net.
<http://www.microsoft.com/net/>
11. David R. Hanson : *lcc.NET : Targeting the .NET Common Intermediate Language from Standard C*, Microsoft Research (2002), MSR-TR-2002-112.
12. Hirotaka Ogawa, Kouya Shimura , Satoshi Matsuoka , Fuyuhiko Maruyama , Yuki-hiko Sohda , and Yasunori Kimura : *OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java*, Elisa Bertino (Ed.) : ECOOP 2000, LNCS 1850, pp. 362–387, 2000.
<http://www.openjit.org/>
13. Sami Jaber : *Génération de dynamique de code .NET et Java*
<http://www.dotnetguru.org/articles/EmitGrandFormat.html>
14. Java Remote Method Invocation (Java RMI)
<http://java.sun.com/javase/technologies/core/basic/rmi/>
15. Anne-Gwenn Bosser : *Réplication distribuée pour la définition des interactions de jeux massivement multi-joueurs*, Thèse, Université Paris 7, 2005
16. Christopher W. Fraser (AT&T Bell Laboratories) and David R. Hanson (Princeton University) and Todd A. Proebsting (The University of Arizona) : *Engineering a Simple, Efficient Code Generator Generator*. ACM Letters on Programming Languages and Systems 1. 1992.
17. François-Nicola Demers and Jacques Malenfant : *Reflection in logic, functional and object-oriented programming : a Short Comparative Study*. IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. 1995.
18. Dennis Sosnisky : *Java programming dynamics, part 2 : Introducing reflection*
<http://www-128.ibm.com/developerworks/java/library/j-dyn0603>
19. Pierre Cointe : *Metaclasses are First Classes : the ObjVlisp Model*. 1987.
20. Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim : *Compiling for Runtime Code Generation*. 2000.
21. Grégoire Henry, Michel Mauny et Emmanuel Chailloux : *Typing la désérialisation sans sérialiser les types*. Journées Francophones des Langages Applicatifs 2006.
22. Andreas Rossberg (Universität des Saarlandes) : *The Missing Link – Dynamic Components for ML*, ICFP 2006.
<http://www.ps.uni-sb.de/Papers/abstracts/missing-link.pdf>
23. H. Tatsuzawa and H. Masuhara and A. Yonezawa : *Aspectual caml : an aspect-oriented functional language*. In Workshop on Foundations of Aspect Oriented Languages, page 3950, March 2005.
24. Sun Microsystems : *JavaServer Pages Technology*.
<http://java.sun.com/products/jsp/>
25. Cheng-Hsueh A. Hsieh and John C. Gyllenhaal and Wen-mei W. Hwu : *Java Bytecode to Native Code Translation : The Caffeine Prototype and Preliminary Results*, (1996).
<http://www.crhc.uiuc.edu/IMPACT/ftp/conference/micro-96-bytecode.pdf>
26. Ankush Varma and Shuvra S. Bhattacharyya : *Java-through-C Compilation : An Enabling Technology for Java in Embedded Systems*, University of Maryland, CollegePark (2004)
27. Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth *Fast, Effective Code Generation in a Just-In-Time Java Compiler*, Intel Corporation, (1998).
<http://www.cs.cmu.edu/~stichnot/pldi98.ps>

28. SeungIl Lee, ByungSun Yang, Suhyun Kim, Seongbae Park, SooMook Moon, Kemal Ebcioğlu, Erik Altman : *Efficient Java Exception Handling in Just-in-Time Compilation*, (2000).
29. Miscellaneous Java JIT compiler publications
<http://schmidt.devlib.org/java/jit-compilers.html>
http://www.trl.ibm.com/projects/jit/pub_int.htm
<http://latte.snu.ac.kr/publications/>