Projet de Typage Et Polymorphisme :

Étude de la surcharge en Java par le codage d'un compilateur Java → ZAM

Gregory Potdevin & Benoît Vaugon

Années 2010-2011

1 Introduction

La surcharge de méthode est une fonctionnalité de plusieurs langages de programmation comme par exemple Java. Le programmeur est alors autorisé à définir, dans une même classe, plusieurs méthodes ayant le même nom, mais des paramètres de type différents. La charge revient alors au compilateur de définir la signature de la méthode qui devra effectivement être appelée.

Ce trait, bien qu'il puisse être très utile, peut aussi être une source d'ambiguité, et donc de bug. Il est donc intéressant de voir comment le compilateur procède à la résolution de la surcharge.

Ce projet est écrit en OCaml, et se décompose en 2 axes majeurs :

- L'analyse d'un code écrit en mini-java, le typage, et la simulation de la surcharge.
- La génération de bytecode zam correspondant au code Java, permettant d'exécuter le programme.

2 Typage

A partir d'un fichier .java, on souhaite obtenir au final un arbre de syntaxe abstraite (Ast) qui pourra être manipulé aisément soit pour simuler le déroulement du programme, soit pour compiler le code. Afin d'obtenir cet arbre, le couple ocamllex/ocamlyacc a été utilisé. On obtient ainsi à partir des sources Java un arbre contenant différents types de noeuds pour les classes, les méthodes, les instructions, les expressions, etc...

2.1 Typage simple des expressions

Une fois l'Ast généré, il convient de l'analyser, d'en vérifier la validité, et de le transformer en une forme qui sera plus facile à manipuler pour le simulateur et le générateur de bytecode. On converti donc l'Ast d'origine en un Ast typé, dans lequel chaque expression expose son type. Ceci permet à la fois de vérifier que les types utilisés sont corrects, et aussi d'ajouter à l'Ast les informations qui seront nécessaires à l'algorithme de surcharge.

Les types étant explicites en Java, on dispose des types de toutes les feuilles (champs, valeurs constantes, valeurs de retour des méthodes, et appels de constructeurs). Il suffit alors de propager ces types dans les nœuds d'expression afin de typer l'ensemble de l'Ast. Certains types peuvent être implicitement convertis en un autre selon les règles suivantes :

- `int` peut être converti en `float` ou `double`
- tous les types peuvent être convertis en `String`, soit grâce à l'appel d'une méthode de conversion pour les types primitifs, soit par l'appel de la méthode `toString` sur les objects
- tout objet peut être casté dans un type correspondant à l'une de ses superclasses
- `null` peut être utilisé pour n'importe quelle classe

Les autres informations remontées dans l'Ast typé sont la résolution des accès à des variables (définir si c'est une variable locale, un paramètre de la méthode courante, ou un champs de l'objet) et la conversion des accès aux champs et aux méthodes en valeurs indexées.

2.2 Surchage des méthodes

Si l'accès au champs est non ambigu, l'accès aux méthodes ne l'est pas forcément. En Java, on peut en effet surcharger les méthodes (définir plusieurs méthodes ayant le même nom, mais avec des arités ou des paramètres différents). Le typage n'est alors pas automatique, puisqu'il y a un choix à faire.

On peut trouver la spéfication du langage Java à l'adresse suivante : `http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.12`

Ce choix peut être unique si une seule méthode convient, ou arbitraire le cas échéant. Le choix de la signature de la méthode se fait donc en plusieurs phases :

1. Présélection des méthodes parmi toutes les méthodes disponibles (ayant la bonne visibilité compte tenu du contexte) de la classe (et de ses superclasses), on ne va garder que les méthodes ayant le bon nom et la bonne arité. Dans le cadre de ce projet et dans un souci de simplicité, les appels de méthodes avec arité variable ne sont pas pris en compte.
2. Sélection des méthodes ayant les bons paramètres Il ne suffit pas d'avoir la bonne arité, il faut aussi que les paramètres des méthodes soient compatibles avec les arguments : pour un argument de type `A` donné, correspondant à un paramètre de type , il faut avoir `A=B` ou `A` sous-type de `B`.

   Exemple : soit une classe `C1`, ayant comme sous-classe `C2`. Qu'en est-il pour les méthodes ayant les signatures suivantes ?
   - `void foo(C1)` → Accepte un argument de type `C1` ou `C2`.
   - `void foo(C2)` → Accepte uniquement un argument de type `C2`.
   - `void foo(Object)` → Accepte tous les objets.

   Une fois les arguments typés, on peut donc tester toutes les méthodes sélectionnées pour ne garder que les méthodes valides pour ces arguments.
3. S'il ne reste plus de méthode, le typage échoue ; s'il ne reste qu'une seule méthode, alors on sait laquelle appeler, et s'il en reste plusieurs, il va falloir définir la méthode la plus adaptée, la plus précise, à savoir celle qui utilise les plus petits types possibles.

   On peut pour cela comparer les méthodes entre elles : si une méthode possède pour chacun de ces paramètres un type plus petit que ceux des autres méthodes, alors on la

garde. Ainsi, `foo(C1, C2)` est plus précise que `foo(C1, C1)`.

Mais parfois 2 méthodes ne sont pas comparables (comme par exemple `foo(C1, C2)` et `foo(C2, C1))`. Il faut alors avoir recours à un algorithme pour résoudre ces cas. Plusieurs algorithmes sont disponibles :

(a) *Java 1.2* : On affecte à chaque méthode une valeur correspondant au produit cartésien de sa classe de définition et des paramètres de la méthode. Les méthodes plus précises possèdent alors des valeurs plus grandes que les autres, ce qui permet de les sélectionner. A noter que plusieurs méthodes peuvent avoir la même valeur, ce qui peut mener soit à une impossibilité à typer, soit à une sélection purement arbitraire (au choix, par exemple la première méthode trouvée).

(b) *Java 1.5* : Fonctionne sur le même principe que précédemment, mais cette fois-ci la valeur est le produit cartésien des paramètres de la méthode. La classe fournissant la méthode n'est donc plus prise en compte. Cela évite qu'une méthode moins précise mais issue d'une sous-classe soit sélectionnée.

(c) *Perfect Match* : En cas d'ambigüité, ne garder que la méthode pour laquelle tous les types sont identiques aux expressions passées en paramètre. Cette méthode a l'avantage de n'avoir aucune ambigüité, mais est très restrictive et entrainera des erreurs de typage là où on aurait pu déterminer une méthode.

(d) *First Match* : Principe simple et peu efficace : on garde la première méthode trouvée.

(e) *Mult* : Multiplication des distances entre les arguments et les paramètres de la méthode. On définit la distance entre 2 types comme le nombre de lien de parentés nécessaires pour aller d'un type à l'autre : si **B** hérite de **A** et **C** hérite de **B**, alors la distance entre **A** et **C** est de 2. On calcule donc l'ensemble des distances, auxquelles on ajoute 1, et on les multiplie. Ainsi, si tous les arguments ont le même type que les paramètres de la méthode, on obtient la plus petite valeur possible : 1. On gardera bien entendue la méthode ayant la valeur minimale.

Ces différents algorithmes peuvent être testés grâce à l'option **-overloader**, suivie du nom de la méthode à utiliser : **first**, **java_1_2**, **java_1_5**, **mult**, ou **perfect**.

2.3 Simulation de l'exécution

Le simulateur permet d'évaluer l'Ast généré par le typeur, ce qui a pour intérêt, entre autres, de tracer les appels de méthodes et donc de simuler la surcharge. En effet, si la signature de la méthode à appeler est bien décidée par le typeur, une classe fille peut redéfinir une méthode de la classe mère. Ce n'est donc qu'à l'exécution que l'on peut savoir quelle méthode sera effectivement appelée. A noter que le code généré permet aussi de tester cela.

Plusieurs exemples sont livrés avec les sources. Afin d'illustrer la différence entre les différents algorithmes (et principalement Java 1.2 et Java 1.5), considérons l'exemple suivant :

```
class A {
    int m (A x) { System.out.println(1+"␣"); return (1); }
    boolean n (B x) {System.out.println(2+"␣"); return (true); }
}
```

```
class B extends A {
    int m (B x) { System.out.println(5+"␣"); return (5); }
    boolean n (A x) {System.out.println(6+"␣"); return (true); }
}
```

Nous utiliserons pour cet exemple ces objets :

```
A a1 = new A ();
B b1 = new B();
A a2 = b1;
```

Considérons les 2 appels suivants :

```
b1.m(b1);
b1.n(b1);
```

Bien que clairement définis en apparence, pour le premier appel on a 2 méthodes candidates : A.m(A x) et B.m(B x) La résolution ne posera ici aucun souci : la méthode B.m(B x) est dans la plus petite classe, et possède une signature qui colle parfaitement avec le type de l'argument. C'est donc celle qui sera sélectionnée.

Pour l'appel suivant, b1.n(b1), la situation n'est plus tout a fait la même. Les 2 candidats sont ici A.n(B x) et B.n(A x). Observons la différence selon les algorithmes :

- Pour Java 1.2, ces méthodes sont représentées par les couples (A, B) et (B, A), qui possède la même valeur, et on ne peut donc pas typer, il y a ambiguïté.
- Pour Java 1.5, les méthodes sont représentées par (B) et (A), et la méthode A.n(B x) est donc plus précise. C'est celle qui sera sélectionnée (et c'est d'ailleurs intuitivement celle qu'on voudrait appeler).
- Pour Perfect Match et Multiply, la même méthode sera sélectionnée.

On voit ici qu'au final tous les algorithmes (en ignorant volontairement "First") ont pris la "bonne" méthode, sauf Java 1.2 ! Cet exemple, d'apparence pourtant trivial, illustre tout le problème de la surcharge, et probablement la raison pour laquelle l'algorithme de Java a été changé. Le fait d'inclure la classe de la méthode dans le calcul du "poids" cause in fine plus de problèmes qu'il n'en résout.

Mais cet exemple illustre aussi tout le problème sous-jacent à la surcharge : quand le programmeur a ajouté la méthode B.n(A x), l'a-t-il fait en pensant qu'elle serait appelée préférentiellement par rapport à A.n(B x), ou juste pour gérer les cas où A.n(B x) ne pourrait être utilisée ? Or, aucun algorithme de surcharge ne peut connaitre l'intention du programmeur, et donc, par essence, toute résolution de surcharge va introduire un risque d'erreur (le programmeur pense que la méthode X va être appelée alors qu'en pratique ce sera la méthode Y).

3 Génération du code

L'application qui a été développée permet de compiler des programmes Java en exécutables pour la machine virtuelle Objective Caml. Ceci a différents intérêts :

- Porter Java sur toutes les architectures sur lesquelles de la machine virtuelle Objective Caml a été portée.
- Tester et comparer les performances des différentes techniques de compilation des programmes Java.

La génération du code pour les instructions et les expressions est très standard. Nous nous intéressons principalement ici à la gestion des objets Java.

3.1 La machine virtuelle Objective Caml

La machine virtuelle Objective Caml est une machine à pile. Elle permet en particulier d'exécuter le code-octet généré par le compilateur Objective Caml `ocamlc` distribué par l'INRIA.

Lors de l'exécution, elle gère différents éléments :

- *stack* : une pile d'évaluation.
- *accu* : un accumulateur pouvant être vu comme le sommet de la pile. Il permet principalement d'améliorer la vitesse d'exécution.
- *heap* : un tas contenant les valeurs allouées.
- *code* : un segment contenant le code-octet à exécuter.
- *pc* : un compteur ordinal pointant sur le segment contenant le code-octet.
- *data* : un tableau contenant les valeurs des variables globales.
- *env* : une variable pointant, lors de l'évaluation du corps d'une fonction, vers la fermeture associée. Celle-ci contient entre autres les valeurs des variables libres du corps de la fonction, ce qui permet d'y accéder rapidement.
- *trapSp* : un registre stockant la position dans la pile du dernier rattrapeur d'exception posé.
- *extraArgs* : un compteur stockant, lors d'un appel de fonction le nombre d'argument passé. Il est utilisé pour repérer les appels partiels.

Il existe 146 instructions de code-octet différentes. Cependant, 60% sont des alias permettant de factoriser le code et d'améliorer la vitesse d'exécution. Ces instructions peuvent être groupées en sept catégories :

- calculs arithmétiques et logiques.
- contrôle : branchements conditionnels et inconditionnels.
- gestion de la mémoire : accès à la pile et au tas, allocations dynamiques.
- gestion des fermetures : créations de fermetures, appels et retours de fonctions.
- gestion des exceptions : poses de rattrapeurs, lancements d'exceptions.
- gestion des objets : appels de méthodes.
- gestion des appels aux fonctions externes.

On peut se référer à la documentation de X. Clerc pour le projet Cadmium [1] pour le détail de chaque instruction.

3.2 Gestion des objets Java

Différentes représentations des objets Java sont possibles. Il aurait par exemple été possible de les représenter par des objets Objective Caml. Cette solution a été rejeté pour des raisons d'efficacité. En effet, l'implantation actuelle de l'appel de methode sur des objets Objective Caml s'effectue en temps logarithmique par rapport au nombre de methodes de l'objet. Pour les objets Java, il est possible d'utiliser de simples tables de methodes pour obtenir un accès en temps constant.

Dans la représentation qui a été choisie, les objets sont représentés par des fermetures. Les valeurs des attributs des objets sont stockés dans l'environnement de ces fermetures. Aucune table des methodes n'est allouée dans le tas. Lorsque l'on appelle une methode sur

un objet, on effectue un appel de fonction sur la fermeture représentant l'objet. L'identifiant (entier) de la méthode à appeler est passé en premier argument. Le code pointé par le pointeur de code de la fermeture représentant un objet consiste donc en une indirection vers le code des différentes méthodes en fonction de la valeur du premier argument. Cette indirection est faite grâce à l'instruction SWITCH de la machine virtuelle.

Cette représentation permet d'obtenir de bonnes performances. En effet, l'appel de methode s'effectue en temps constant, et l'indirection vers le code de la méthode à appeler se fait en une seule instruction de la machine virtuelle. Ainsi, aucun calcul n'est effectué à l'initialisation du programme, et le tas ne stocke que les objets et tableaux alloués explicitement.

3.3 Étude de performance

Trois jeux de tests ont été effectués pour comparer les temps de calcul.

1. **Simple boucle `for`**

Technique de compilation	Temps de calcul (s)
Programme OCaml compilé avec ocamlopt	10
Programme Java avec javac/java de sun	12
Programme OCaml compilé avec ocamlc	490
Programme Java compilé avec notre compilateur	520
Programme Java simulé par notre interpréteur	14000

2. **Calcul de nombres premiers**

Technique de compilation	Temps de calcul (s)
Programme Java avec javac/java de sun	11.2
Programme Java compilé avec notre compilateur	42

3. **Opérations arithmétiques sur les nombres de Péano**

Technique de compilation	Temps de calcul (s)
Programme Java avec javac/java de sun	2.5
Programme Java compilé avec notre compilateur	17

4 Utilisation

Une fois compilée, javaz s'utilise en ligne de commande de la façon suivante :
`javaz [OPTIONS] source.java`

Les options disponibles sont :

`-o <filename>` → définir le nom de fichier à utiliser pour l'exécutable
`-verbose` → afficher un maximum d'informations
`-silent` → ne rien afficher
`-print-ast` → afficher l'Ast généré par l'analyse syntaxique
`-print-typed-ast` → affecter l'Ast type
`-simulate` → évaluer le code dans le simulateur après l'avoir typé. Attention, les performances du simulateur sont loin d'égaler celles de l'exécutable généré.
`-no-compile` → ne pas compiler
`-overloader <technique>` → définir l'algorithme de surcharge à utiliser (java_1_2, java_1_5, first, perfect, mult)

5 Conclusion

Au niveau de la surcharge, dans la plupart des cas, il n'est pas difficile de déterminer statiquement quelle méthode devrait être appelée. L'algorithme utilisé dans Java 1.5 est à la fois simple et efficace. Il reste certes des cas d'ambiguité, mais en pratique, si l'algorithme ne permet pas de déterminer précisément quelle méthode devrait être appelée, c'est que le programme n'est probablement pas assez lisible. Il semble nécessaire que les algorithmes de résolution ne soient pas trop compliqués, car tout programmeur doit être en mesure de déterminer lui-même quelle méthode devrait être appelée. Toute erreur dans le choix de la méthode peut entraîner des bugs qu'il sera difficile à tracer par la suite. Refuser de compiler en cas d'ambiguité apparait donc comme le choix le plus sûr.

Concernant la génération de bytecode ZAM, bien que les performances soient en retrait par rapport à la JVM, les performances restent tout de même plus qu'acceptables (la JVM utilisée possède un JIT). Avec une implémentation plus aboutie, on pourrait même imaginer d'utiliser ce compilateur pour l'intégration de sources Java au sein de sources OCaml. En couplant le bytecode généré avec OCaml for PIC, ce projet ouvre aussi la voie à l'utilisation de Java sur microcontroleur PIC, sans avoir toute la lourdeur d'une machine virtuelle Java.

Table des matières

Références

[1] Xavier Clerc. *Cadmium*, February 2010. `http://cadmium.x9c.fr/distrib/cadmium.pdf`.