Réalisation d'un compilateur Petit Java

Théo ZIMMERMAN Joris GIOVANNANGELI

14 janvier 2012

Introduction

Une remarque générale sur la réalisation de ce projet, qui se divisait en trois phases de travail, est que c'est lorsqu'on passe à une nouvelle phase que l'on se rend compte que la précédente ne convient pas.

Ainsi la programmation d'un compilateur pour **Petit Java** a nécessité de permanents va-et-vient entre les différents morceaux du code.

1 Analyses lexicale et syntaxique

Pour tout le projet, nous avons utilisé les outils présentés en cours ; pour les **analyses lexicale et syntaxique**, il s'agissait d'**ocamllex** et de **menhir**.

Les conflits qui sont apparus, lors de la compilation des *if* ou des expressions m'ont permis d'expérimenter deux méthodes de résolution. La première était de séparer chaque règle en plusieurs en utilisant les noms terminaux. Cela s'est avéré lourd mais efficace. Je l'ai conservée pour les *if*. La seconde consistait en l'usage des précédences, et j'ai pu constater l'élégance de ce procédé. Je l'ai utilisé pour les expressions lorsque j'ai dû en réécrire le parsage après avoir modifié l'AST pour les besoins des étapes utlérieures.

L'épineux problème du *cast* m'a fait découvrir, à travers la manière que j'ai choisi pour le résoudre, tout l'intérêt de lancer ses propres exceptions dans le parser.

Avec la méthode que nous avons adoptée pour traiter *System.out.print*, qui consiste à le séparer des autres appels de méthodes dès l'analyse syntaxique, et à définir trois «faux» mots-clefs pour les trois mots en jeu, cette écriture est obligatoire. Par exemple, *((((System)))).out.print* lève une erreur de syntaxe. Nous avons jugé que ce n'était pas grave, puisque de toute façon, cette «méthode statique» de **Java** était «parachutée».

Depuis le début, nous avons mal géré l'instruction vide. Au lieu de la considérer comme une instruction normale, nous avons tout fait pour l'éliminer, par exemple avec des types option. Du coup, nous nous sommes rendus compte seulement à la fin que le test *if* de *exec* ne passait pas l'analyse syntaxique. Pour régler ce problème, il aurait fallu changer l'AST et par conséquent modifier une très large partie de notre code.

2 Analyse sémantique et typage

L'analyse sémantique est réalisée en 3 passes successives sur l'ensemble des classes. L'entrée est l'arbre syntaxique issu du parsage, dont le type est donné par le module *Past*, dans le fichier *ast.ml*.

Première passe

Dans un premier temps, une fonction **buildClassMap** construit une Map des classes depuis la liste des classes issue du parsage, pour s'abstraire des contraintes de positions de la définition dans le fichier. Parallèlement, un graphe de l'héritage est construit, sous la forme d'une Map qui à chaque classe associe la liste des classes qui en héritent. La fonction vérifie que :

- chaque classe est définie une seule fois
- aucune classe n'hérite de *String*

Deuxième passe

Une deuxième passe est ensuite effectuée par la fonction **checkHerit** qui parcourt l'arbre représantant la relation d'héritage. Un parcourt en profondeur est utilisé. Cette passe transforme un arbre de syntaxe *Past* en un arbre de syntaxe *Oast* intermédiaire, et réalise de nombreux tests sémantiques :

- toute classe hérite d'une classe existante
- absence de cycle dans l'héritage
- toutes les méthodes et les constructeurs ont un profil bien formé, dont chaque argument porte un nom unique
- tout les constructeurs portent bien le nom de la classe
- dans chaque classe, il n'y a qu'une seule méthode et un seul constructeur de même signature
- les champs ont un type bien formé et sont uniques

En outre, on construit ici ce qui deviendra plus tard les descripteurs de classe. On conserve dans un tableau toutes les méthodes définies par les surclasses, leur profil, et leur type de retour. Une méthode sera donc identifiée par une classe et un index dans ce tableau. On réalise un tableau identique pour les constructeurs. Pour éviter la redondance, les corps des méthodes et des constructeurs sont déplacés dans un tableau global, et identifiés par un entier unique. Cette méthode permet de gérer simplement les redéfinitions et les surcharges.

Dans un premier temps, je n'avais pas compris que les attributs pouvaient être redéfinis, chose qu'il a fallu ré-implémenter par la suite. En outre, les profils d"arguments sont conservés sous la forme de liste, ce qui implique ordonnée. La première implémentation du test de sous-typage tenait compte de cet ordre, ce qui impliquait une différence entre

```
(int a; boolean b)
```

et

```
(boolean a; int b)
```

. Il a fallu ajouter une relation d'ordre sur les types, et trier les profils avant d'en tester le sous-typage, pour avoir un vrai test de différence sur les signatures. En outre, j'ai perdu beaucoup de temps sur des erreurs assez incompréhensibles qui venaient d'un mauvais usage des opérateurs de comparaison. J'utilisais != au lieu de <>, ce qui levait des exceptions alors même que les valeurs étaient réellement différentes.

Troisième passe

La troisième passe parcourt les classes et type le corps des méthodes. Il s'agit d'une bête implémentation des règles de typage du sujet. Le point important ici est le choix de la méthode effectivement appelée en fonction du profil des arguments. Pour cela, on construit l'ensemble des méthodes portant le bon nom, et appartement aux surclasses de la classe courante, en parcourant le descripteur de méthodes. Puis, cet ensemble, conservé sous forme de liste, est triée selon la relation de sous-typage. Le minimum est choisi s'il existe. Une méthode est alors transformée en un nom de classe et un entier, représentant sa position dans le descripteur. Un autre point à tester est la présence d'une instruction *return* dans toutes les branches d'exévutions possibles, ce qui est fait lors du typage des instructions. Ce parcours est réalisé par la fonction **typProg** et par l'ensemble des fonctions du module **typInstr**. Il renvoit un programme tel que défini dans l'arbre de syntaxe du module **Sast**.

Un bon nombre de problème a été soulevé par le traitement d'exceptions réservées à *Object* et à *String*. En effet, leur absence dans la Map des classes contraint à prévoir tout les cas où il faut traiter les choses différemments pour ces deux types. Leur traitement particulier a été motivé par le fait qu'il s'agit de classes particulières :

- *Object* n'hérite de rien
- *String* est incodable en Petit Java à cause de l'absence de tableau.

Toutefois, a posteriori, il semble qu'il aurait été plus souhaitable de les traiter le plus normalement possible.

3 Production de code

Cette partie est très clairement la plus «magique» de tout le compilateur. Programmer en assembleur est relativement ardu. Mais ça l'est encore plus lorsqu'il s'agît de manipuler des choses que l'on ne connaît pas vraiment.

J'ai dès le début mis dans mes fonctions de compilation un accumulateur pour éviter la concaténation de code **MIPS** (stocké dans une liste). C'était une mauvaise idée. Même si je l'ai conservé jusqu'à la fin, cet accumulateur

allourdit le travail et n'évite pas d'être contraint de faire régulièrement des concaténations.

Contrairement à ce que spécifiait l'énoncé, j'ai associé le pointeur *null* à la valeur zéro. Cela permet de faire un grand nombre de tests plus facilement avec les instructions de **MIPS**. Cela permet aussi de simplifier l'affectation par défaut des valeurs des attributs des objets instanciés. En effet, on peut tout mettre à zéro, que ce soit les entiers, les booléens ou les pointeurs.

Il n'y a pas de test pour empêcher l'accès à un champ de *null*. Il aurait été mieux de le rajouter, mais vu la représentation de *null* adoptée, une telle action fait déjà échouer **SPIM**.

L'implémentation de *String_concat* a soulevé des difficultés. Il fallait penser à ce que les caractères ne tiennent que sur des octets et pas sur des mots usuels de 32 bits et utiliser les fonctions adéquates.

A l'exécution, il y a encore certains tests qui ne passent pas.

Notamment la fonction *String_ofint*, codée en **MIPS** pour les besoins de la concaténation ne fonctionne correctement qu'avec les entiers positifs. Pourtant, on avait fait en sorte de gérer les deux cas.