École Normale Supérieure 2015–2016

projet de compilation

Petit Scala

partie 2 — à rendre avant le dimanche 17 janvier 18:00

version 1 — 7 décembre 2015

L'analyse syntaxique et le typage ont été réalisés dans la première partie du projet. Dans cette seconde partie, on va réaliser la compilation de Petit Scala vers l'assembleur X86-64.

1 Production de code

L'objectif est de réaliser un compilateur simple mais correct. En particulier, on ne cherche pas à faire d'allocation de registres mais on se contente d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible d'utiliser localement les registres. On ne cherche pas à libérer la mémoire.

1.1 Organisation des données

1.1.1 Représentation des valeurs

Types primitifs. Les entiers sont représentés de manière immédiate par des entiers machine 64 bits signés. Les booléens sont représentés par des entiers : 0 représente `false` et `1` représente `true`. La valeur `()` est représentée par l'entier 0.

Objets. Un objet est représenté par une adresse, pointant sur un bloc alloué sur le tas. Ce bloc contient

- un *descripteur de classe* (voir ci-dessous) ;
- les valeurs des champs de l'objet.

Les champs sont organisés de telle manière qu'un même champ d'une classe C se trouve dans le bloc à la même position pour tout objet de la classe C ou de toute sous-classe de C (organisation dite en *préfixe*). Ceci est possible car l'héritage est *simple*. Ainsi, les deux classes `A` et `B` suivantes :

```
class A { var x: Int = 0; var b: Boolean = true }
class B extends A { var d: Int = 42 }
```

donnent lieu à des objets ayant la forme suivante :

descr. `A`
`x`
`b`

descr. `B`
`x`
`b`
`d`

Le compilateur maintient donc une table donnant, pour chaque classe C et chaque champ de C, la position où trouver ce champ dans un objet de la classe C.

La valeur `null`. Elle est représentée par l'entier 0. En particulier, elle est différente de toute adresse d'un objet alloué sur le tas.

Les classes prédéfinies. Les objets de ces classes sont représentés exactement comme les autres. Un objet de la classe `String` contient un unique champ, qui est un pointeur vers la chaîne de caractères (allouée dans le segment de données).

1.1.2 Descripteurs de classes

Chaque classe est représentée par un descripteur de classe. Il est alloué dans le segment de données. Il contient :
- le descripteur de sa super-classe ;
- la liste des codes des méthodes.

Exactement comme pour la représentation des objets, la liste de ces codes obéit à une organisation préfixe : le code pour la méthode `f` doit se trouver au même endroit dans le descripteur de la classe `A` qui la définit et dans celui de la classe `B` qui la redéfinit. Ainsi le code suivant :

```
class A {
    def f() : Int = { return 0 };
    def g() : Boolean = { return true }
}
class B extends A {
    override def f() : Int = { return 1 };
    def h() : Boolean = { return false }
}
```

donne lieu à des descripteurs de la forme suivante :

descr. de `A` :

descr. de `AnyRef`
`A_f`
`A_g`

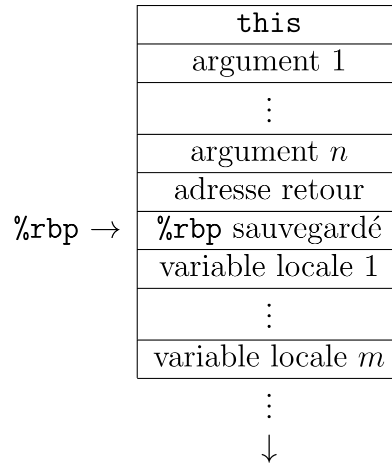
descr. de `B` :

descr. de `A`
`B_f`
`A_g`
`B_h`

De même que pour les champs, le compilateur maintient une table donnant, pour chaque classe C et chaque méthode de C, la position où trouver le code de cette méthode dans le descripteur de la classe C.

1.2 Schéma de compilation

Les méthodes et les constructeurs sont représentés par des fonctions (*i.e.*, du code assembleur appelé par un `call`). L'appelant place la valeur de l'objet (`this`) et les arguments sur la pile. L'appelé place la valeur de retour dans le registre `%rax`. Au retour, l'appelant se charge de dépiler `this` et les arguments.



Attention : les arguments d'un *constructeur* sont considérés comme des champs de l'objet et n'apparaissent donc pas comme des arguments sur la pile.

La valeur d'une expression est compilée en utilisant la pile si besoin et en plaçant sa valeur finale dans `%rdi` ou en sommet de pile (au choix).

Le code assembleur produit au final doit ressembler à quelque chose comme

```
              .text
              .globl main
main:         allocation d'un objet de la classe Main
              appel de sa méthode main
              xorq %rax, %rax
              ret
C_Main:       constructeur de la classe Main
M_Main_main:  méthode main de la classe Main
...           autres méthodes et constructeurs
              .data
D_Main:       .quad D_Any
              .quad M_Main_main
...           autres descripteurs de classes
...           chaînes de caractères
```

2 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. Votre projet doit se présenter sous forme d'une archive `tar` compressée (option "z" de `tar`), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé pscala. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format ASCII, PostScript ou PDF.

Étendre votre compilateur avec une nouvelle option de ligne de commande `--type-only`. Cette option indique de stopper la compilation après l'analyse sémantique (typage). En l'absence de cette option, et si le fichier d'entrée est conforme à la syntaxe et au typage décrits dans la première partie du projet, votre compilateur doit produire du code X86-64 et terminer avec le code de sortie 0 (`exit 0` explicite ou terminaison normale du programme), sans rien afficher. Si le fichier d'entrée est `file.scala`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.scala`). Ce fichier X86-64 doit pouvoir être exécuté avec la commande

```
gcc file.s -o file
./file
```

Le résultat affiché sur la sortie standard doit être identique à celui donné par l'exécution du fichier Scala `file.scala` avec

```
scalac file.scala
scala  Main
```

Remarque importante. La correction du projet est réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec l'instruction `print`, qui sont compilés avec votre compilateur et dont la sortie est comparée à la sortie attendue. Il est donc très important de correctement compiler les appels à `print`.