Projet de compilation du langage mini-scala

Joseph de Vilmarest et Ulysse Marteau-Ferey

le 17 janvier 2015

1 Lexer-parser

Nous avons suivi exactement la syntaxe de l'énoncé. Nous avons un peu compliqué les choses dans le parser, notamment car nous n'avons pas utilisé les outils permettant de faire des *?* facilement.

Il marche correctement sur les différents tests lexicaux.

La localisation fonctionne relativement bien, à un ou deux caractères près.

2 Typeur

2.1 Implémentation

Nous avons choisi de reprendre les données de l'arbre de syntaxe abstraite telles quelles, et donc une classe se représente comme un type

*string * (param_type_ class list) * (parametre list) * typ * (expr list) * (decl list)*

Nous avons codé notre environnement comme un type enregistrement avec deux champs mutables :

{ mutable classes : classaug Cmap.t ; mutable dec : dec_ var list }

où *dec_ var* représente la déclaration d'une variable dans un environnement et est de la forme

type dec_ var = Var of string type | Val of string * type*

et où classes est un *Map* qui à un identificateur de classe associe une classe augmentée définie par

type classaug = { mutable classe : classe ; borneinf : typ}

En variable globale, nous avons créé (tardivement) un map qui à une classe associe l'environnement propre *gamma_prime* créé pour typer cette classe. Il nous sert dans la production de code.

Nous avons choisi cette façon de représenter les classes pour que l'on puisse les modifier, notamment lorsque l'on teste le bon typage des méthodes et champs et que l'on doit les rajouter petit à petit dans la liste des déclarations. Cela nous permet donc de garder la classe sous sa forme originelle en *classe*, et donc d'avoir accès aux méthodes

définies dans cette classe, tout en ne rajoutant que celles typées pour l'instant dans la *classaug* associée.

De plus, tout ceci s'accompagne d'une décoration (on a donc un type enregistrement encore plus gros) qui prend la forme schématique suivante :

{ o : objet; lo : localiation de l'objet}

Pour les expressions, on a le schéma un peu différent suivant :

{ e : expression; le : localisation de l'expression; te = type de l'expression}

ce qui nous permet d'accéder au type d'une expression lors de la production de code.

Par ailleurs, nous avons suivi l'ordre du sujet.

2.2 Problèmes rencontrés, points à améliorer

Nous avons rencontré quelques problèmes, notamment car nous avons souvent créé des structures de données inadéquates : il a donc fallu en modifier quelques unes, ce qui a rendu une partie du code très peu lisible (en particulier sur des types enregistrement en cascade). La localisation des erreurs a également rendu le code particulièrement lourd.

Nous avons souvent du modifier des structures car nous avons eu du mal à comprendre comment fonctionnait la programmation objet au début. La compréhension de ce qu'on faisait n'est réellement venue qu'au moment du débuggage, où nous avons vu à quoi ressemblait du code scala et à quoi il pouvait servir. Les éléments où nous sommes le plus longtemps restés bloqués sont l'ajout de méthode à une classe (notamment avec l'override) et le typage des expressions d'accès, notamment parce que l'on change de classe. Les tests qui nous ont causé le plus de problèmes sont les tests de l'*exec*.

Finalement, nous sommes arrivés à d'assez bons résultats sur le typeur, seuls restent quelques tests que nous devrions rejeter (*typing/bad*).

La principale chose que nous avons modifiée est le fait que pour nous, toute classe hérite d'une autre, ce qui fait que *Any* hérite d'elle-même, et aussi que l'on ne peut avoir *var a $\bar{e}$xpression* : elle est toujours typée avec un type par défaut *Any*, ce qui fait que très souvent, on doit traiter ce cas à part. Nous devons donc apporter quelques modifications aux conditions d'unicité des héritages. De plus, nous n'avons pas traité les cas d'unicité des blocs. Nous vous enverrons peut-être une version où ces derniers problèmes sont réglés.

3 Production de code

3.1 Implémentation

Nous avons tenté de suivre quasiment à a lettre le cours et le sujet. Nous avons choisi d'implémenter les constantes (entiers, booléens, chaines ...) comme des objets comme les autres.

Nous avons utilisés quelques conventions tacites. Les registres suivants sont utilisés dans ces cas et uniquement dans ceux-là :

- dans les constructeurs de classes, $\%r12$ stocke l'adresse du prochain champ à remplir ;
- $\%r13$ et $\%r14$ sont utilisés dans le calcul de *Eop(e,o,f)* pour stocker les valeurs des objets calculés e et f ;
- $\%r15$: pour pouvoir accéder aux champs d'un objet, on met l'adresse de cet objet dans $\%r15$ avant d'appeler une méthode avec lui.

Nous avons aussi alloué la mémoire pour les variables locales en déplaçant $\%rsp$ et $\%rbp$ mais nous n'avons pas modifié l'ast comme dans le TD 10, nous nous sommes contentés d'écrire une fonction qui calcule l'allocation nécessaire et qui détermine pour chaque variable locale son emplacement dans l'espace alloué (nous avons jugé que rajouter un champ à notre ast aurait donné quelque chose d'encore plus illisible). Cette fonction *alloc_expr* est utilisée au début de la compilation de chaque méthode.

3.2 Problème rencontrés, points à améliorer

Il reste une quantité non négligeable de tests que nous ne passons pas. A priori, nous avons codé cette partie dans son intégralité, mais il reste des problèmes, notamment dans l'appel aux méthodes.

Une des principales difficultés dans l'écriture du code a été le passage d'expression simples du type *print(int)* (que l'on compilait sans constructeurs dans un premier temps et qui était relativement satisfaisante car donnait des résultats) à une représentation plus complexe. En effet la gestion des classes des objets et des méthodes est beaucoup plus difficile à appréhender abstraitement. Tout entier est donné sous la forme d'un pointeur vers un objet de descripteur *D_Int* et d'unique champ un entier. Cela ne semble pas très intéressant dans le cas des constantes mais nous avons estimé que traiter le cas des constantes à part serait source de disjonctions de cas.

Pour le débuggage, nous avons plusieurs problèmes. Le premier est l'absence de localisation qui permet de se perdre dans du code assembleur. En effet, le lien entre assembleur est code initial n'est pas toujours évident, et nous avons préféré débugger en regardant le code assembleur plutôt qu'utiliser gdb, outil que nous ne connaissons pas. De plus, les imbrications multiples posent beaucoup de problèmes pour trouver d'où vient réellement l'erreur.

Les offset et la gestion de la pile nous ont particulièrement dérangés au début, mais ces problèmes sont en grande partie réglés (dans le cas des offset en tout cas). Nos principaux problèmes sont dans les méthodes, et peut-être aussi dans les accès.