

Projet de compilation

Joseph de Vilmaest et Ulysse Marteau-Ferey

le 17 janvier 2015

1 Parser et lexer

Nous avons suivi exactement la syntaxe de l'énoncé. Nous avons un peu compliqué les choses dans le parser, notamment car nous n'avons 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 à 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 global, nous avons créé (tardivement) un map qui à une classe associe l'environnement propre Γ' créé pour typer cette classe. Il nous sert dans la production de code et pour gérer des accès propres à une classe. De la même façon, nous avons créé un environnement similaire pour gérer les méthodes : un map qui à un nom de classe associe un map qui à un nom de methode associe l'environnement Γ'' correspondant.

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 :

$\{ o : \textit{objet}; lo : \textit{localiation de l'objet} \}$

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

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

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

Nous avons réalisé tardivement une grosse erreur dans notre implémentation qui nous a conduit à refaire une partie du typeur : nous n'avions pas changé d'environnement. Nous n'avions pas vraiment de Γ' et Γ'' , nous rajoutions juste des éléments à Γ . Nous avons donc ajouté des structures adéquates sur notre typeur, ce qui rend quelques éléments obsolètes, ou plutôt mal faits : l'intérêt des classes augmentées diminue fortement une fois ce problème trouvé. Nous avons cependant modifié le code pour nous adapter à cela.

2.2 Problèmes rencontrés, choses non faites

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

Nous avons souvent dû 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 venu qu'au moment du debug, 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 méthodes 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 modifié est le fait que pour nous, toute classe hérite d'une autre, ce qui fait que par *Any* hérite d'elle-même, et aussi que l'on ne peut avoir *var a : expression* : elle est toujours typée avec un type par défaut *Any*, ce qui fait que très souvent, on doit traiter ce cas à part. Nous avons donc apporté quelques modifications aux conditions d'unicité des héritages. Nous avons désormais traité le cas d'unicité des blocs. A priori, le typeur fonctionne correctement.

3 Production de code

3.1 Implémentation

Nous avons tenter de suivre quasiment à la lettre le cours et le sujet. Nous avons choisi d'implémenter les entiers et les chaînes de caractère comme des objets comme les autres.

Nous avons utilisés quelques conventions tacites pour les registres "locaux" :

- dans les constructeurs de classes, %r12 stocke l'adresse du prochain champ à remplir ;
- %r13 et %r14 sont utilisés dans le calcul de type $Eop(e, o, f)$ pour stocker les valeurs des objets calculés ;
- %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 (nous avons jugés que rajouter un champ à notre ast aurait donné quelque chose d'encore plus illisible).

3.2 Problème rencontrés, choses non faites

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.

Les principales difficultés que nous avons rencontrées lorsque nous avons codé la production de code est le passage d'expression simples du type $print(int)$ qui pouvait se faire sans constructeurs dans un premier temps et qui était relativement satisfaisante car donnait des résultats, à la gestion des classes des objets et des méthodes, plus difficiles à appréhender abstraitement.

Pour le debug, 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. 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. Nos principaux problèmes sont dans les méthodes, et peut-être aussi dans les accès.

Nous savons également qu'il y a un problème de typage, dans le sens où lorsque l'on type une variable de la forme $var x : \tau = e$, le typage lui donne le type τ ce qui est normal. Par contre à la compilation, il faut plutôt considérer qu'elle a le type de l'expression, ce que nous ne gérons pas. Pour cela, il faudrait modifier le typeur pour que lorsque l'on type une expression, il renvoie non seulement le type mais aussi le type effectif. Nous n'aurons malheureusement pas le temps de faire cela.