Rapport - Compilation : Petit Scala, première partie

Nathanaël Courant, Noémie Cartier

8 décembre 2015

1 Choix techniques

1.1 Analyseur lexical

Nous avons choisi, pour des raisons de performance et de minimisation du nombre d'états de l'automate de l'analyseur lexical, de stocker les mots-clef et le lexème correspondant dans une table de hachage. En revanche, comme il est possible de placer plusieurs opérateurs à la suite sans séparateur (mais que tous les opérateurs ne sont pas réduits à un signe) nous avons réalisé leur reconnaissance en écrivant un cas par opérateur.

1.2 Analyseur syntaxique

Nous avons beaucoup utilisé la bibliothèque de Menhir, en particulier les fonctions `option`, `delimited`, `list` et `separated_list`. La fonction `ioption` nous a permis de résoudre un conflit.

Les constantes valant précisément -2^{31} nous ont posé des problèmes : en effet, elles ne peuvent pas être reconnues immédiatement par l'analyseur lexical, puisque celui-ci n'est pas capable d'identifier si le symbole - devant la constante est celui de la constante, ou est un opérateur binaire. De plus, nous n'avons pas réussi à écrire une grammaire sans conflits qui permettrait de détecter ces constantes négatives lors de l'analyse syntaxique. Ce que nous avons dont fait a été de rajouter une phase de traitement après l'analyse syntaxique, où les - unaires précédant un entier sont transformés en constantes entières négatives, et où les débordements de constantes sont détectés.

1.3 Typeur

Dans la définition d'un type (visible au début du fichier `type_ast.ml`), nous avons créé trois constructeurs différents afin de mieux gérer la portée des différentes variables de type.

De la même façon, nous avons choisi d'utiliser une structure de type `Map` au lieu de `Hashtbl` pour avoir des environnements persistants.

Nous avons géré les paramètres de constructeur de classe comme des champs privés, auquels on ne peut accéder que si l'expression de l'objet est `this`. Ainsi, si `a` est un paramètre du constructeur de la classe T, les expressions `a` et `this.a` sont autorisées, tandis que `(new T(0)).a` et `{(); this}.a` ne le sont pas, comme en Scala. Une autre possibilité aurait été d'accepter les accès de ce type uniquement lorsque le type de l'expression de laquelle on essaie d'accéder au paramètre du constructeur est exactement celle de la classe (et non pas un sous-type de celle-ci), ce qui aurait préservé la sûreté du typage, mais nous avons choisi de faire comme Scala pour ce point-là.

Dans le calcul de la variance, nous avons dû prendre un double paramètre en permanence : la classe (ou le type, selon les arguments nécessaires pour chaque fonction) créée par le typeur et celle fournie par l'analyseur syntaxique, les premiers portant les informations de typage qui a déjà été fait (portée des variables de type, type des champs lorsque celui-ci n'est pas précisé en particulier), et les deuxième portant la localisation à afficher en cas d'erreur.

Nous décorons également l'arbre de syntaxe abstraite au cours du typage, en ajoutant à chaque sous-expression son type. Au cours de cette phase, on effectue également les opérations suivantes :

- détection de la portée des variables : une variable peut soit être une variable locale, soit un paramètre de la méthode appelée, soit un paramètre du constructeur de la classe à l'intérieur de laquelle elle apparaît ;
- détection de la méthode appelée : lors d'un appel de méthode, on identifie de quelle classe cette méthode provient (i.e. dans quelle classe cette méthode a été définie en premier, en remontant dans la hiérarchie des classes tant que cette méthode existe), afin de pouvoir compiler ces appels ultérieurement.

Modification – 08/12/2015 : Nous avons remarqué que l'instanciation de paramètres de types, ou de classes définies par défaut peuvent conduire à une perte de la sûreté du système de typage : en effet, cela permet en particulier une création d'objets de type `Nothing`. On ne peut pas non plus créer d'instance d'un paramètre de type, même si celui-ci est déclaré comme héritant d'une autre classe, puisque les arguments du constructeur peuvent différer (et même changer de nombre) entre une classe et une classe dont elle hérite ! Par conséquent, nous avons interdit toute instanciation d'un paramètre de type d'une classe, ainsi que des classes de base (pour celles-ci, l'instantiation devant soit être interdite, soit est inutile car correspond à créer des objets de type `Unit`, `Int`, `String` ou `Boolean`, pour lesquels il existe déjà d'autres manières de les créer, soit de type `Any`, `AnyRef` ou `AnyVal`, qui n'ont aucune utilité, car il n'est rien possible de faire avec ces objets).

2 Tests supplémentaires

Nous nous aperçus sommes que, bien que notre compilateur passe déjà tous les tests fournis pour le projet, il ne se comportait pas comme il aurait dû dans certains cas particuliers. Nous avons donc créé quelques nouveaux tests (ceux dont le nom est de la forme `extra-*.scala`).

Ces tests supplémentaires vérifient ces cas particuliers :

- vérification du fait que la constante -2^{31} est acceptée ;
- vérification de la variance dans un champ d'une classe dont le type n'est pas précisé dans le programme ;
- vérification de la bonne gestion des variables de type lorsque leur nom est commun à un paramètre de méthode et à un paramètre de classe, ou à une classe définie précédemment ;
- vérification de la bonne gestion de variables de type nommées comme une classe standard (`Int`, `Unit`, ...).
- *Modification – 08/12/2015* : vérification qu'il est impossible d'instancier des paramètres de types ou des classes de base.