Interprète Kawa

Nadine Hage Chehade, Lisa Ceresola

Table des matières

1 Résumé

L'objectif de ce projet a été de contruire un interprète pour un petit langage objet inspiré de Java. (à compléter)

2 Analyse syntaxique

Nous avons réalisé l'analyse syntaxique du programme et produit l'arbre de syntaxe abstraite en complétant les fichiers `kawalexer.mly` et `kawaparser.mly`. Le fichier `kawalexer.mly` représente la syntaxe concrète du langage Kawa. On y introduit des symboles terminaux : mots-clés, identifiants, constantes ; et des symboles non terminaux. Nous avons ajouté des règles de priorité : l'accès à un attribut obtient la priorité la plus élévée, vient ensuite les opérations arithmétiques et les opérations logiques.

3 Vérification des types

La vérification des types se fait dans le fichier `typechecker.ml` (à compléter)

4 Interprétation

L'interprétation se fait dans le fichier `interpreter.ml`. (à compléter, je pense qu'il faut attendre qu'on ait terminé car les fonctions changent tout le temps)

4.1 Valeurs et opérations de base

Fonctions utiles :
`eval` : évalue une expression
`exec` : exécute une instruction
`exec_seq` : exécute une séquence d'instructions.

4.2 Classes et objets

Fonction utile :

`evalnew` : permet de créer un nouvel objet. On ajoute les attributs de la classe courante et des classes mères s'il y a, dans notre environnement local. Si la création de notre nouvel objet fait appel à un constructeur, alors nous faisons appel à une méthode donc utilisons la fonction `eval_call`

4.3 Variables et attributs

Fonction utile :
`evalvar` : évalue si la variable ou l'attribut ont été déclarés.
Pour le cas d'une variable, on regarde si celle ci est dans l'environnement local ou global. Si ce n'est pas le cas, alors notre variable n'est pas définie.
Pour le cas des attributs d'une classe, il nous suffit de chercher le nom de notre attribut dans la Hashtbl représentant les attributs de la classe à l'aide la fonction `Hashtbl.find`.

4.4 Méthodes

Fonction utile :
`eval_call` qui évalue un appel de méthode. Cette fonction prend en argument : le nom de la méthode, l'objet qui appelle cette méthode, les arguments de cette méthode.
Nous avons donc dans un premier temps cherché la définition de cette méthode. Il a fallu ensuite créer un environnement local pour exécuter la fonction et lui ajouter notre objet passé en argument. Il a ensuite fallu associer les arguments aux différents paramètres de cette fonction et ajouter le tout à notre environnement

local. Nous ajoutons par la suite toutes les variables locales à notre environnement local.
parler du return !! → on a créé un pointeur

5 Helper.ml

Nous avons créé un nouveau fichier comprenant des fonctions régulièrement appelées dans notre code : Je ne sais pas si on dit lorsque les fonctions sont utilisées ?

- `aux` : fonction prenant en paramètre une liste de tuples comprenant eux-même des listes et deux accumulateurs de liste. Elle renvoie alors un tuple composé de deux listes. Cette fonction est utilisée pour effectuer des déclarations en série avec valeurs initiales.

- `map_ident_val_attributes` : Elle vérifie, lors de la déclaration avec valeur initiale, que le nombre de valeur donnée correspond au nombre de déclarations. Elle renvoie un tuple. Le premier élément un quadruplet donnant les particularités de l'attribut (nom,type,est-t-il final ?,visibilité). Le second élément constitue une liste d'instructions. Cette fonction est utilisée lors de la déclaration d'attributs avec valeur initiale.

- `map_ident_val_variables` : Effectue la même chose que la fonction `map_ident_val_attributes`, mais pour les variables. Cependant, le premier élément du tuple renvoyé est un tuple de la forme : (nom,type). Cette fonction est utilisée lors de la déclaration de variables avec valeur initiale.
- `find_cls_def` : fonction prenant en paramètre le nom d'une classe et le programme. Cette fonction vérifie que la classe existe et que deux classes n'ont pas le même nom. Si ces deux conditions sont vérifiées, alors la définition de la classe est renvoyée, sinon on renvoie une erreur.

- `find_mthd_def` : cette fonction prend en paramètre le nom d'une classe c, le nom d'une méthode m et le programme. La fonction va alors renvoyer la définition de la méthode m et qui est accessible depuis la classe c. On vérifie bien que la classe et la méthode existent, et on recherche aussi la méthode dans les classes parentes si elle n'existe pas dans la classe courante.

- `separate_attributes` : cette fonction prend en paramètre une liste de quadruplet correspondant aux informations de chaque attributs. Elle renvoie alors 4 listes :
(i) `attr_type_list` : de la forme (nom,type) ;
(ii) `attr_final_names` : comporte le nom des attributs `final`
(iii) `attr_private_names` : comporte le nom des attributs `private`
(iv) `attr_protected_names` : comporte le nom des attributs `protected`

- `is_sub_class` : fonction prenant en paramètre le nom de deux classes et renvoie vrai si la première classe est une sous-classe de la deuxième.

6 Extensions

6.1 Champs immuables

Pour donner la possibilité de déclarer un attribut `final` nous avons d'abord ajouté le mot clé FINAL dans le lexer. Il a fallu l'ajouter en temps que token dans le parser. Pour différencier les attributs dont la valeur peut être modifié et ceux dont la valeur ne peut être modifié, il a fallu modifier la syntaxe abstraite. Nous avons décidé d'ajouter un nouveau champ `attributes_final` dans le type `class_def`.

6.2 Déclaration en série

Pour réussir à déclarer simultanément plusieurs variables de la forme : `var int x,y,z` nous avons modifié uniquement le fichier `kawaparser.mly` en précisant que le champ `names` n'est plus un identifiant mais un liste d'identifiants.

6.3 Déclaration avec valeur initiale

Pour que l'initialisation des attributs des classes mères se fasse, nous avons créé un nouveau champ `init_instr` dans le type `class_def`. Celui ci contient une liste d'instructions, en réalité ces instructions sont uniquement des affectations. Ainsi, grâce à ce champ, nous pouvons stocker les affectations et ensuite les utiliser pour les effectuer lorsque nous ajoutons les attributs aux environnements locaux. Ils seront alors initialisés. Ceux des classes-mères également.

6.4 Égalité structurelle

Pour cette extension, il a fallu dans un premier temps rajouter les opérateurs `===` et `=/=` dans le fichier `kawalexer.mly`. Il nous a donc fallu ajouter ce token dans `kawaparser.mly` et modifier la grammaire de `binop` en y ajoutant les opérateurs nécessaires et ajouter l'associativité à gauche.
Pour la vérification des types, l'idée est la même que pour la cas d'égalité et d'inégalité. Pour compléter le fichier `interpreter.ml`, l'idée est aussi la même que le cas d'égalité et d'inégalité, seulement, nous remplaçons l'égalité physique des champs par l'égalité structurelle des champs.

6.5 Super

Rajouter les mots-clés. rajouter une expression de la forme super.f(...) dans la grammaire. L'idée est simple, il faut chercher dans la classe-mère (si elle existe) une fonction nommée f. Ainsi, on appelle cette fonction sur l'objet de la classe-fille. Ajout d'un paramètre à la fonction `eval_call` afin de savoir si l'on cherche la méthode dans la/les classes-mères ou dans la classe "d'origine" (très mal dit).

6.6 Transtypage

On suppose que le nom des classes commence par une majuscule et le nom des variables et attributs commence par une minuscule. En effet, nous avions avant d'implémenter cette règle, un conflit du type reduce/reduce pour une expression de la forme (*ident*). On pouvait soit, réduire ident comme un type soit le réduire comme une variable. (à compléter)

6.7 Visibilité

Ajout d'un nouveau champ `visib` dans `method_def` pour la visibilité. Un enrichissement de la grammaire a été nécessaire et un nouveau type somme a été créé : `type visibility = Private | Protected | Public`.
Suivant la valeur du champ `visib`, l'attribut ou méthode est alors accessible :

- depuis toutes les classes (public) ;
- depuis la classe courante ainsi que ses sous-classes (protected) ;
- depuis la classe courante uniquement (private).

6.8 Test de type

(à compléter) `instanceof`

7 Idées et notes

→ Tableaux en cours (à faire) :
- TAB t name[] = new t[n] ; (création du tableau de type t et de n éléments
à voir car il semblerait que la création ait été implémentée de la sorte : [e1,e2,e3].
- .add(type t) qui ajoute un élément à la fin du tableau ;
- .remove() qui retire le dernier élément du tableau ;

- accès : t[i] où i est l'indice → vérifier que t est un tableau et i est un entier compris entre 0 et la taille du tableau (exclu).
- écriture : t[i] = n ;
- il faut bien vérifier que les expressions soient du même type, notamment lors de l'écriture ou ajout d'un élément.
- peut-etre faut il créer un nouveau type pour les tableaux ? car on ne peut pas avoir un tableau de TVoid...