# 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 commencer par réserver le mot clé FINAL dans le lexer, et l'intégrer dans la grammaire. Pour différencier les attributs dont la valeur peut être modifiée et ceux dont la valeur ne peut être modifiée, nous avons modifié la syntaxe abstraite. Nous avons décidé d'ajouter un nouveau champ `attributes_final` dans le type `class_def`. L'étape suivante consiste à modifier le typechecker pour s'assurer qu'aucune méthode à part le constructeur ne modifie la valeur de cet attribut. Cette modification a été faite au niveau de la fonction `type_mem_access`. Pour savoir si on a le droit de modifier ou pas un attribut, on a rajouté un paramètre `check_bool` à cette fonction. Ce paramètre vaut `false` si la méthode est appelée depuis le constructeur et `false` sinon.

Notre modification permet alors d'interdire les autres méthodes de changer la valeur d'un attribut ayant été déclaré comme `final`. Cependant, notre compilateur, à l'état actuel, permet au constructeur de changer la valeur d'un tel attribut, c'est-à-dire d'avoir deux instructions d'affectations successives. Un simple booléen indiquant si une instruction d'affectation est présente dans le constructeur ne suffit pas pour résoudre ce probléme. En effet, on pourrait avoir un branchement conditionnel avec deux affectations différentes suivant une certaine condition. Il faudrait plutôt explorer les différentes branches du code afin de s'assurer que chaque branche initialise cet attribut correctement.

## 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 non vide d'identifiants.

Voici la définition des différents symboles de la grammaire qui implémentent cette extension :

```
<var_decl> := VAR <type_decl> separated_nonempty_list(COMMA,IDENT) SEMI
<attribute_decl> := ATTRIBUTE <type_decl> separated_nonempty_list(COMMA,IDENT) SEMI;
```

Il s'agit ensuite de renvoyer une liste qui contient chaque nom de variable associé au type présent dans la déclaration.

## 6.3 Déclaration avec valeur initiale

La partie facile de cette extension concerne les variables locales et globales. En effet, il suffit dans ce cas de modifier la grammaire pour permettre d'initialiser directement la variable. Nous sommes inspirés de la syntaxe de `python` qui permet d'affecter des valeurs à plusieurs variables dans la même ligne, comme le montre l'exemple suivant :

```
int x, y = 1, 2;
```

Voici la grammaire modifiée :

```
<var_decl> := VAR type_decl separated_nonempty_list(COMMA,IDENT)
              SET separated_nonempty_list(COMMA,expression) SEMI
```

Si le nombre d'expressions présentes à droite du signe d'affectation ne correspond pas au nombre de variables, la fonction `map_ident_val_attributes` erreur indiquant à l'utilisateur s'il y a des valeurs manquantes, ou, au contraire, des valeurs supplémentaires qu'il faut retirer.

Ensuite, il s'agit de décomposer cette ligne de code en deux : déclaration et affectation. Pour les variables globales, nous avons rajouté les instructions des affectations correspondantes au tout début de la liste des instructions du `main`. De même pour les variables locales des méthodes.

Pour les attributs, la modification de la grammaire donne une nouvelle règle identique au cas des variables :

```
<attribute_decl> := ATTRIBUTE <type_decl> separated_nonempty_list(COMMA,IDENT)
                    SET separated_nonempty_list(COMMA,expression) SEMI
```

Le traitement des attributs est plus compliqué. En effet, on ne peut pas juste rajouter les instructions des affectations au code du constructeur pour plusieurs raisons. D'abord, il est possible de créer une nouvelle instance d'un objet sans appeler le constructeur. Dans ce cas, les attributs concernés ne seront pas même pas initialisés. Ensuite, il y a les questions de l'héritage. Une classe fille peut ne pas redéfinir le constructeur de la classe mère. Il faudrait donc commencer par remonter la hiérarchie des classes, récupérer le code du premier constructeur rencontré pour ensuite l'ajouter à la liste des méthodes de la classe courante en y introduisant les affectations.

Une solution plus simple consiste à modifier la définition d'une classe dans le fichier `kawa.ml` Nous avons ajouté un nouveau champ `init_instr` dans `class_def`. Il s'agit d'une liste d'instructions qui sont, en réalité, uniquement des affectations. Ainsi, grâce à ce champ, nous pouvons stocker les affectations et les exécuter lors de la création d'une nouvelle instance d'un objet, que le constructeur soit appelé ou pas. Les attributs seront alors initialisés par la fonction `eval_new` de l'interprète. Ceux des classes-mères également, comme le montre l'extrait de code suivant :

```
let rec exec_init init_instr_list obj =
    match init_instr_list with
    | [] -> ()
    | Set(Field(This, x), e)::suite ->
        let () = Hashtbl.replace obj.fields x (eval e)
        in exec_init suite obj
```

```
        | _ -> failwith "eval_new : cas non atteignable - que des sets dans
    init_instr_list"

    in let rec init_attributes_current_and_parents class_name obj =
        let cls_def = find_cls_def class_name p in
        let () = exec_init cls_def.init_instr obj in
        match cls_def.parent with
        | None -> ()
        | Some parent_class_name -> init_attributes_current_and_parents parent_class_name
    obj
```

Dans le typechecker, on commence par vérifier la cohérence de cette liste d'instructions dans la fonction `check_class` avant de passer à la vérification des méthodes.

## 6.4 Égalité structurelle

Pour cette extension, il a fallu dans un premier temps rajouter les opérateurs `===` et `=/=` dans le fichier `kawalexer.mly` ainsi que les tokens `EQSTRUCT` et `NEQSTRUCT` dans `kawaparser.mly`. Ensuite, nous avons modifié la grammaire de `binop` en y ajoutant les opérateurs nécessaires et ajouter l'associativité à gauche.

Pour la vérification des types, de même que pour la cas d'égalité et d'inégalité, on vérifie que les deux expressions comparées sont de même type.

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 et des tableaux par l'égalité structurelle en utilisant l'opérateur `==` d'`ocaml`.

## 6.5 Super

Après avoir réservé le mot clé `super`, nous étendons le type `expr` en rajoutant le constructeur suivant pour stocker le nom de la méthode appelée ainsi que le liste des paramètres :

```
| Super of string * expr list
```

Nous introduisons une nouvelle règle dans la grammaire pour le symbole `expression` :

```
<expression> := SUPER DOT IDENT LPAR separated_list(COMMA,expression) RPAR
```

Ensuite, pour vérifier la cohérence de cette expression dans le typechecker, il s'agit de vérifier que la méthode appelée est bien définie dans la classe parente dont hérite la classe contenant cet appel. On ne remonte pas plus loin dans la hiérarchie des classes. Afin de localiser facilement la classe courante pour laquelle on se trouve lors de la vérification d'une certaine classe, on crée une référence vers le nom de la classe courante :

```
let class_level = ref ""
```

Cette variable est ensuite modifiée dans la fonction `check_class`. On accède donc facilement au nom de classe courante, on récupère sa définition, on vérifie si elle hérite bien d'une autre classe qui définit la méthode appelée avec `super`. Le type de cette expression sera donc le type de cette méthode tel que précisé dans la classe parente.

En ce qui concerne l'évaluation d'une telle expression, il suffit d'appeler `eval_call` avec le nom de la méthode, en précisant le paramètre implicite comme étant `this` en en ajoutant un paramètre `super` pour indiquer à la fonction de chercher la définition de la méthode dans la classe parente.

## 6.6 Transtypage

Afin d'utiliser la syntaxe de `Java` pour le transtypage qui est la suivante :

```
(T) e
```

Nous avons introduit une nouvelle règle pour le transtypage :

```
LPAR t=type_decl RPAR e=expression
```

Cependant nous avons été confrontées à un conlit reduce/reduce qu'on illustre par l'exemple suivant :

```
(Point) origine
```

D'une part, cette expression peut être réduite en une expression de transtypage suivant la règle au-dessus. D'autre part, `Point` peu être considérée comme le nom d'une variable et réduit en un accès mémoire. Différentes solutions sont envisageables pour résoudre ce probléme. La première que nous avons implémenté dans un premier temps pour effectuer des tests a été de proposer une alternative à la syntaxe `Java` comme suit `cast(T, e)`. Une autre solution serait de garder une liste dynamique des types définis dans un programme et vérifier si `T` en fait partie.

La solution que nous avons implémentée est de suivre les mêmes convention de nommage présentes dans `Ocaml`. On suppose que le nom des classes commence par une majuscule et le nom des variables et attributs commence par une minuscule. On introduit un nouveau token pour les noms des classes et on modifie la définition du symbol `typ` pour intégrer ce token.

Pour la vérification des types, il s'agit de s'assurer que les deux classes concernées se trouvent dans la même branche. De plus, on interdit le transtypage vers un type de base. On ne s'attarde pas non plus sur le cas des tableaux, qui pourraient être intéressant dans le cas d'un tableau contenant des objets.

A l'interprétation, on vérifie que le type ciblé est une sur-classe de la classe de `e`. Dans le cas contraire, on déclenche une erreur indiquant que le downcast est interdit.

## 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) Il s'agit pour cette extension de modifier le code de notre projet afin de permettre à l'utilisateur de tester le type dynamique d'une variable à l'aide de l'opérateur `instanceof`.

D'abord, nous avons ajouté une nouvelle règle pour le symbole non terminal `expression`

```
<expression> := expression INSTANCE_OF type_decl
```

Pour résoudre le conflit shift/reduce suite à l'extension de la grammaire, nous avons affecté au token `INSTANCE_OF` une priorité plus élevée

```
instanceof
```

## 6.9 Tableaux

Nous avons rajouté dans notre language la possibilité de manipuler des tableaux de taille fixe. Nous résumons dans ce qui suit les étapes de l'implémentation

### 6.9.1 Un type pour les tableaux

Nous modifions la définition des types comme suit :

```
type typ =
    | TVoid
    | TInt
    | TBool
    | TClass of string
    | TArr of typ
    | EmptyArr
```

Cette définition devient donc une définition récursive. Le dernier type ne sert que dans le typechecker. Il intervient dans la vérification des instructions comme la suivante où la liste vide reste cohérente avec une liste d'entiers.

```
var int[] t;
t = [];
```

### 6.9.2 Modification de la grammaire

Il faut intégrer dans la grammaire les éléments de syntaxe suivant :

— création d'un nouveau tableau en énumérant ces éléments : `[1, 2, 3]`

```
<expression> := LBRACKET separated_list(COMMA, expression) RBRACKET
```

— création d'un tableau en indiquant sa taille `new int[n]`

```
<expression> := NEW t=type_decl LBRACKET e=expression RBRACKET
```

Cette nouvelle règle nous donnait un conflit reduce/reduce avec la règle d'accès à une case d'un tableau (voir plus bas). En effet, une expression `new Point[5]` peut être réduite en expression suivant la règle au-dessus. Mais une autre possibilité est de réduire d'abord `new Point` en expression, puis toute cette phrase en un accès mémoire. Afin de contourner ce problème, nous avons opté pour une variante de cette syntaxe en utilisant un autre mot-clé `new_arr` spécifique aux tableaux. Il suffit donc de remplacer le token `NEW` par `NEWARR` et la grammaire obtenue ne présente aucun conflit.

— accès mémoire pour la lecture ou l'écriture `t[i]` :

```
<mem_access> := expression LBRACKET expression RBRACKET
```

### 6.9.3 Extension de la syntaxe abstraite

On étend le type `expr` en ajoutant les constructeurs suivant :

```
| Array of expr array
| ArrayNull of typ * expr
```

Le premier correspond à la création d'un tableau en listant les éléments, et le second à un tableau dont seule la taille est connue.

### 6.9.4 Règles de typage pour les tableaux

— L'expression `[e1, ..., eN]` est bien typée et de type `T[]` si chaque élément est de type `T`
— L'expression `t[e]` est bien typée et de type `T` si `t` est un tableau de type `T[]` et `e` est un entier
— L'instruction d'affectation `t = e` est bien typée dans le cas où `t` est un tableau de type `T[]` si l'expression `e` est un tableau vide ou un tableau de type `T[]`

### 6.9.5 Interprétation

On représente les tableaux par des `Array` qui sont des structures de données mutables afin de faciliter la modification d'un tableau. L'accès à une case du tableau renvoie une erreur si l'indice est négatif ou supérieur ou égal à la taille du tableau. Un tableau créé avec `new_arr` contient les valeurs `Null` dans toutes ses cases au moment de la création.

# 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...