Interprte Kawa

Nadine Hage Chehade, Lisa Ceresola

Table des matières

1 Résumé

L'objectif de ce projet a été de construire un interprète pour un petit langage objet inspiré de Java. Nous avons complété et modifié les fichiers `kawalexer.mll`, `kawaparser.mly`, `typechecker.ml` et `interpreter.ml` pour la base du projet. En ce qui concerne les extensions, les fichiers précédents ont égalememt été modfiés, nous avons de plus apporter quelques modifications aux fichiers `kawai.ml` et `kawa.ml`.

2 Analyse syntaxique, vérification des types et interprétation

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, opérateurs... 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. La vérification des types se fait, elle, dans le fichier `typechecker.ml` et l'interprétation se fait dans le fichier `interpreter.ml`. Nous avons pris le soin de commenter le code en détail et n'expliquons donc que certaines parties relatives aux extensions. Dans la suite du rapport, nous présentons le fichier `Helper.ml`, quelques difficultés rencontrées, et l'implémentation des différentes extensions.

3 Fichier `helper.ml`

Nous avons créé un fichier `helper.ml` contenant des fonctions fréquemment appelées et que nous allons détaillées :

- `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. La première (respectivement deuxième) est la concaténation des listes du premier (respectivement deuxième) élement de chaque tuple.

- `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.

- `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).

- `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.

4 Difficulté rencontrée lors du travail de base : le return

Le problème que nous avons rencontré se situait, dans la fonction `exec`, lorsque qu'il fallait exécuter une instruction de la forme `return e` où `e` était une expression. Trouver une solution afin d'avoir accès à ce que devait renvoyer notre méthode n'a pas été immédiat.
Une première idée consistait à stocker une variable `return` dans l'environnement. Comme ce mot-clé est réservé, il est garanti qu'aucune variable n'aura le même nom.

Une autre solution, celle que nous avons adopté, est de créer une variable globale `return_exp` qui est une référence mutable et de l'initialiser à `Null`.

```
let return_exp = ref Null
```

Ainsi, lorsque nous devons exécuter une instruction de la forme `return e` il nous faudra donc évaluer l'expression et mettre à jour la valeur stockée dans `return_exp` en y plaçant le résultat de notre évaluation. Comme un return marque la fin du méthode, il faut, après avoir exécuté toutes les instructions qui s'y trouvent, déréférencer la référence pour renvoyer la valeur qui s'y trouve.

```
!return_exp
```

Nous nous sommes également posées de nombreuses questions en travaillant sur le projet ; en voici quelques unes et comment nous y avons remédiées :

- `Return manquant` : si le type de la méthode n'est pas `void`, vérifier qu'il y a bien une instruction `return` (sauf si la méthode est abstraite).
- Classes ou méthodes ayant le même nom : on choisit de déclencher une erreur au lieu de sélectionner la méthode/classe définie en dernier pour éviter toute ambiguïté.
- Types bien définis : vérifier que les types apparaissant dans une déclaration existent.
- Héritage et redéfinition : rien dans notre compilateur dans sa version actuelle n'interdit de redéfinir des attributs (redéclarer un attribut ayant le même nom et potentiellement un autre type) dans une sous-classe. Tout simplement, ces redéfinitions masquent celles des classes parentes, et lors de l'évaluation, l'interprète ne remonte pas dans la hiérarchie des classes puisqu'il a déjà trouvé un candidat.

5 Extensions

Nous avons implémenté différentes extensions. Nous expliquons dans ce qui suit notre démarche pour chacune d'entre elles.

5.1 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.

5.2 Déclaration en série

Pour réussir à déclarer simultanément plusieurs variables de la façon suivante : `var int x,y,z` nous avons modifié 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
```

On renvoie ensuite une liste qui contient chaque nom de variable associé au type présent dans la déclaration.

5.3 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` lance une 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 cependant 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 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.

5.4 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`.

5.5 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 dans 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.

5.6 Test de type

Le but est 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` la même priorité que les opérateurs de comparaison.

Dans le typechecker, nous avons interdit l'usage de l'opérateur `instanceof` sur les types de base (int, bool, void). Pour les tableaux, nous avons également opté d'interdire l'usage de cet opérateur. Nous avons tenté d'adapter l'évaluation de cette expression quand le type testé est un tableau. Mais s'est posé le problème des tableaux vides. En effet, si un tableau est vide, on ne dispose pas d'assez d'information pour déterminer le type d'un tableau vide (on n'a pas la possibilité de remonter à l'étiquette de la variable ou de la signature de la méthode qui a renvoyé ce tableau vide comme résultat). Dans le cas d'un objet, l'expression `e instanceof t` a simplement le type `TBool`.

Dans l'inteprète, il s'agit finalement de vérifier que la classe ciblée est bien une sur-classe du type dynamique de l'expression `e` en utilisant la fonction ci-dessous où `target_class` est le type contenu dans `t` :

```
1     let rec is_instance_of class_name =
2         if class_name = target_class then true (*on verifie si le type dynamique de e = t
      *)
3         else let class_def = find_cls_def class_name p in  (*on regarde s'il est sous-type
      *)
4         match class_def.parent with
5         | None -> false
6         | Some parent_class_name -> is_instance_of parent_class_name
```

5.7 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 :

```
1     <expression> := 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 néanmoins être intéressant dans le cas d'un tableau contenant des objets.

Pour 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.

5.8 Visibilité

Pour préciser la visibilité d'une méthode, nous avons commencé par définir un nouveau type somme a été créé :

```
type visibility = Private | Protected | Public
```

Ensuite nous avons ajouté un nouveau champ `visib` dans `method_def` pour la visibilité. Si la signature d'une méthode ne contient pas le mot clé `private` ou `protected`, on suppose alors qu'elle est publique par défaut et accessible par `main` ou d'autres classes. Un enrichissement de la grammaire a été nécessaire pour déclarer une méthode comme privée ou protégée en utilisant le mot clé correspondant dans la signature après `method`.

Le typechecker se charge ensuite de vérifier les deux conditions suivantes dans la fonction `type_mthcall` comme le montre le code ci-dessous :

— une méthode privée ne peut pas être appelée depuis l'extérieur de la classe courante, et ne peut pas être redéfinie dans une sous-classe (cette dernière condition est vérifiée directement dans `check_mdef` en cherchant dans les classes parentes, la signature d'une méthode ayant le même nom)

— une méthode protégée ne peut être appelée que depuis la classe elle-même ou l'une de ses sous-classes

```
let () = match mthd.visib with
    | Private -> if obj_class_name <> (!class_level) then  error (Printf.sprintf "The
    method %s is private and can't be accessed outside the class" mthd_name)
    | Protected -> if (is_sub_class (!class_level) obj_class_name p) = false then error (
    Printf.sprintf "The method %s is protected and is not accessible" mthd_name)
    | Public -> ()
```

L'approche est très similaire pour les attributs. Comme pour les attributs finaux, nous ajoutons les champs suivant dans `class_def` :

```
attributes_private: string list;
attributes_protected: string list;
```

ainsi que le possibilité de déclarer un attribut comme privé ou protégé avec la syntaxe suivante :

```
attribute private int x;
attribute protected bool b;
```

Comme pour les méthodes, un attribut qui n'est ni privé ni protégé est public par défaut. Le typechecker vérifie que l'accès aux attributs est cohérent avec sa visibilité. Voici par exemple le code qui vérifie qu'un attribut privé est correctement utilisé :

```
    let check_private is_private attr_name found_class =
          match obj with
          | This -> (*si l'attribut est prive, il ne doit pas etre herite*)
                    if (is_private && class_name != found_class) then
                          error (Printf.sprintf "The attribute %s is private, cannot be
    used in a subclass" x)
          | _ ->  (*si l'attribut est private alors on n'a pas le droit de l'utiliser si
    on est a l'exterieur de la classe *)
                    if is_private && (!class_level <> class_name) then
                        error (Printf.sprintf "The attribute %s is private, cannot be used
     outside the class" x)
```

Si on accède à l'attribut avec `this`, alors l'attribut privé doit être défini dans cette même classe (et non dans une classe parente). Dans les autres cas, il faut que l'objet par lequel on accède à l'attribut soit une instance de la classe dans laquelle on se trouve.

5.9 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

5.9.1 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 = [];
```

5.9.2 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 type_decl LBRACKET 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
```

5.9.3 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.

5.9.4 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[]`

5.9.5 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.

5.10 Classes et méthodes abstraites

Le mot-clé `abstract` utilisé après `class` ou `method` permet de définir des classes et des méthodes abstraites.

Une méthode privée ne peut pas être abstraite car elle ne peut pas être redéfinie dans une sous-classe. De plus, une méthode abstraite ne contient aucune instruction, d'où les règles suivantes dans la grammaire :

```
<method_def> := METHOD ABSTRACT <type_decl> IDENT LPAR separated_list(COMMA,arg) RPAR
    BEGIN END
    | METHOD PROTECTED ABSTRACT <type_decl> IDENT LPAR separated_list(COMMA,arg) RPAR
    BEGIN END
```

Une classe abstraite ne pouvant pas être instanciée, le typechecker vérifie, dans la fonction `type_expr`, que la création d'un nouvel objet avec `new` ne concerne pas une classe abstraite. Dans la fonction `check_class`, on vérifie les conditions suivantes :

— si une classe est abstraite, toutes ses classes parentes sont abstraites

— si une classe n'est pas abstraite, alors aucune de ses méthodes héritées n'est abstraite

Cependant une vérification que nous n'avons pas effectuée est la cohérence entre la signature de la redéfinition et et celle de la méthode abstraite.

5.11 "Missing semicolon"

Nous avons implémenté cette extension en deux temps. Pour détecter les points-virgules manquant entre deux déclarations de variables ou d'attributs, nous avons extrait le mot qui suit directement l'endroit où l'erreur a été déclenchée dans le parser. Si ce mot correspond à main, var, attribute, method, class, abstract ou une accolade fermante, c'est qu'il manque un `;` pour compléter la déclaration.

Ensuite pour détecter les `;` manquant dans les instructions, nous avons introduit de nouveaux symboles non-terminaux pour parser une instruction sans `SEMI`, mais suivie d'un token marquant le début d'une nouvelle instruction, ou la fin du bloc de code. Si une telle instruction mal formée est parsée, alors une erreur spécifique à l'absence d'un point-virgule est déclenchée, et un message d'erreur correspndant est affiché.

```
    <missing_semi> :=
    PRINT LPAR <expression> RPAR <follow>
    | <expression> <follow> {}
    | RETURN <expression> <follow> {}
    | <memory_access> SET <expression> <follow> {}
    ;

    <follow> :=
    | PRINT
    | IF
    | WHILE
    | RETURN
    | IDENT
    | NEW
    | SUPER
    | END
    | THIS
    | NEWARR
```