1 Introduction

Je vais présenter dans ce rapport le travail réalisé pour la conception d'un interpréteur du langage *Kawa*. L'ensemble de la partie minimale du projet a été réalisée. Une partie des extensions a également été réalisée. Ce rapport sera donc organisé en deux partie: la première présentera le travail réalisé pour la partie minimale, la seconde présentera les différentes extensions réalisées.

L'ensemble du code se trouve sur le **Git**. On peut retrouver la partie minimale (sans extensions) dans la branche **Kawa_v1** et la partie avec entensions dans la branche **Kawa**.

2 Implémentation minimale

J'ai d'abord commencé par compléter le lexer et le parser. Une fois le parser complété, j'ai compilé puis corrigé les quelques erreurs qu'il pouvait y avoir. Cette partie terminée, j'ai pu commencé à implémenter l'interpréteur.

L'interpréteur gère les différentes instructions et expressions du langage. Les expressions d'opérations sont plutôts simples, on évalue les expressions puis on effectue les opérations. D'autres expressions et instructions demandent plus d'opérations. L'expression **new** doit créer un nouvel objet. Si l'objet n'hérite de rien, on peut créer l'environnement et rendre un nouvelle objet avec cet environnement. Si l'objet hérite, on va créer l'objet parent (qui lui même va créer ses parents s'il en a), on récupère l'environnement de l'objet parent pour y ajouter les attributs de l'objet fils. Une partie intéressante de l'interpréteur est l'appel de méthodes. Pour l'appel de méthode on va d'abord recherche la méthode à partir du nom donné. On peut ensuite appeler *eval_call* avec la méthode, l'objet et les paramètres de la méthode. Il faut ajouter l'objet lui-même à l'environnement qui permettra l'utilisation du mot clé *this* et également les paramètres donnés. Une fois l'environnement de l'appel complet, on peut exécuter la séquence d'instructions de la méthode. Pour terminer avec l'interpréteur j'ai ajouté une table de hachage pour pouvoir récupérer rapidement les classes à partir du nom de la classe et ainsi que la possibilité d'afficher tous les types avec **Print**.

Une fois l'interpréteur terminé j'ai pu commencé à le tester. À ce niveau il est possible d'exécuter du code mais une erreur de typage va lever une erreur sans donner plus d'information ou bien le programme pourrait avoir un comportement non souhaité. Je me suis ensuite penché sur la vérification des types du programme.

La vérification du typage passe d'abord par la vérification du typage des classes puis par la vérification du *main*. Comme pour l'interpréteur il faut vérifier les instructions et les expressions. Pour la plupart c'est plutôt simple, on vérifie le typage et si tout est bon on renvoie le type produit. C'est la cas par exemple pour toutes les opérations binaires. D'autres opérations demandent un peu plus de vérification. C'est le cas de **SET** qui pour un objet doit vérifier que l'objet

donné est du même type que la variable ou bien est un sous-type de la variable. De manière similaire, pour l'accès mémoire en notation pointée et l'appel de méthode il faut vérifier que l'attribut est dans la classe ou dans l'une de ses classes mères.

3 Extensions

J'ai implémenté les extensions suivantes:

- Champs immuables
- Visibilités
- Déclarations en série
- Déclarations avec valeur initiale
- Champs statiques
- Test de type
- Transtypage
- Super
- Tableaux
- Egalité structurelle
- "Did you mean 'recursion' ?"
- Le processus ne peut pas aboutir en raison d'un problème technique
- "Missing Semicolon"

Ces extensions ont posées plus ou moins de difficultés et ont nécessitées différents changements dans le code. Nous allons revenir sur les différents changements faits pour mettre en place ces extensions.

3.1 Champs immuables

Les champs immuables ont nécessités la modification du typechecker. Il faut en effet vérifier que les accès mémoire sur des attributs *Final* ne se fassent pas avec **SET**. On peut cependant y accéder avec **GET**. C'est pourquoi j'ai créé une exception **Final** qui indique que la variable demandée est final. C'est ensuite à *SET* et *GET* de gérer cette exeption (pour SET on lève une erreur). L'information qu'une variable est immuable ou non est liée à la visibilité de la variable (voir 3.2).

3.2 Visibilités

La visibilité des attributs a nécessité plusieurs modifications. D'abord j'ai ajouté un type **visibility** qui contient les différentes visibilités possibles pour une variable. Les visibilités telles que **FPRIVATE** commencent par un **F** pour indiquer que l'attribut est immuable (Final). Les variables globales ont une visibilité **UNDEFINED**. Cela permet de traiter les variables globales comme les attributs. Ensuite j'ai ajouté un type **global_def** qui permet de récupérer les attributs et les méthodes. En effet si on ajoute les mots clés PRIVATE et PROTECTED et qu'on récupère les attributs puis les méthodes on obtient un conflit à cause des listes. Avec ce système on récupère une liste d'attributs et de méthodes donc plus de conflits. Les attributs et les méthodes sont ensuite trié quand on récupère une classe. On revient au comportement qu'on avait avant d'ajouter les visibilités et il suffit ensuite de vérifier lors de l'accès mémoire qu'on a le droit d'y accéder avec la visibilité associé à la variable.

3.3 Déclarations en série

La déclaration en série se fait grâce au type *var_decl* qui renvoie une liste de déclaration. Si une seule variable est déclarée on renvoie une liste avec un seul élément. Sinon on renvoie une liste de variables qui sera ensuite ajouté aux déclarations.

3.4 Déclarations avec valeur initiale

Pour la déclaration avec valeur initiale j'ai ajouté une nouvelle instruction **SETDEF** qui permet de traiter les déclarations avec valeur initiale différemment pour les classes. Pour les variables globales il suffit de récupérer les variables avec une expression puis d'ajouter au code un SET(variable, expression) (cette opération est réalisée dans le Parser). Ainsi quand le code s'exécute on donne bien une valeur à la variable. Cette méthode ne fonctionne plus avec les classes car par exemple si notre attribut est statique et immuable on ne va pas pouvoir ajouter un SET pour donner la valeur. C'est la raison pour laquelle j'ai ajouté SETDEF qui permet d'ignorer ces paramètres des variables.

3.5 Champs statiques

Un nouveau type de variable a été créé. Il permet de mettre ces variables à part pour les stocker non pas avec les autres mais dans un champs spécial de la classe. Ils sont ensuite géré comme une variable globale dans l'interpréteur. Les champs statiques ont pour clé dans l'environnement **NomClasse.attribut**. Comme les variables ne peuvent pas contenir de point on sait que ce nom sera unique dans l'environnement. Cela simplifie également l'accès car lorsqu'on a récupéré le nom de la classe et de l'attribut on peut directement y accèder dans l'environnement.

3.6 Test de type

Le test de type ajoute une expression **InstanceOf** qui prend une expression et une chaine de caractère. On vérifie que la chaine de caractère est un nom de classe valide et que l'expression est de type *Tclass*. L'interpréteur vérifie ensuite si l'objet ou l'un des parents de l'objet est du type donné.

3.7 Transtypage

Le transtypage ajoute une expression **Cast** qui prend deux expressions. Le nom de classe pour le Cast est une expression et non pas une chaine de caractères. La raison est qu'une chaine de caractères cause un conflit avec l'expression parenthésée. J'ai donc décidé de mettre une expression qui doit être *Get (Var classe, ...)* avec "classe" la classe du cast (on vérifie ça dans le typechecker). Une fois ces informations récupérer, le typechecker n'ayant accès qu'au type statique il doit vérifier que la classe du cast est une classe fille ou une classe mère de la classe à cast. Enfin l'interpréteur doit également vérifier que la classe du cast est une classe fille du type dynamique. Si toutes ces vérifications sont valides l'interpréteur renvoie l'objet.

3.8 Super

Super est une nouvelle expression. Cette expression fonctionne comme le mot clé *This*. La différence est qu'au lieu de faire référence à l'objet courant ("this" dans l'environnement), cette expression va récupérer "this" puis rendre la classe mère. Ainsi un accès avec Super va rechercher directement dans la classe mère ou ses classes mères.

3.9 Tableaux

Les tableaux sont créés grâce à un nouveau type **TArray a**. Les tableaux sont stockés via les tableaux ocaml. Un tableau est créé à partir d'une taille et d'un type. L'interpréteur va initialiser les éléments du tableau à Null. Pour un tableau à plusieurs dimensions, tous les tableaux sont initialisés et il est possible d'accéder et de donner une valeur directement aux éléments des tableaux. Le typechecker va vérfier que tous les éléments du tableau sont du même type.

3.10 Egalité structurelle

L'égalité structurelle vérifie que les attributs des deux objets ont la même valeur.

3.11 "Did you mean 'recursion' ?"

Cette extension a été très intéressante à implémenter. Il a fallu trouver un moyen de rechercher le mot le plus proche dans une liste. Pour trouver le mot le plus proche j'ai utilisé la distance de Levenshtein. J'ai créer un module *wordcomp.ml* qui gère les comparaisons dans lequel j'ai implémenté les fonctions necéssaires.

Il suffit ensuite de faire des appels lorsqu'un identifiant n'est pas trouvé dans l'environnement.

3.12 Le processus ne peut pas aboutir en raison d'un problème technique

Cette extension ajoute simplement un champ **Lexeme.position** qui est récupéré via le parser. On peut ensuite récupérer la ligne dans le typechecker et l'afficher en cas d'erreur.

3.13 "Missing Semicolon"

L'interpréteur indique les erreurs de point-virgule et les erreurs d'accolades. Pour cela j'utilise le symbole *error* qui permet de lever une exception en cas d'erreur.