Projet Kawa–LLVM
Document de travail v1.2

Florent Nicart

30 octobre 2014

1 Syntaxe du langage

— Les entités supportées seront : les classes, les classes abstraites, les interfaces
— leur visibilité (par rapport au paquetage) pourra être soit public, soit privée,
— les espaces de noms seront définis uniquement à l'aide de paquetage. Les paquetages peuvent contenir des entités et d'autres paquetages, mais les entités ne pourront pas êtres imbriquées (ex : classe/interface déclarée à l'intérieur d'une autre classe/interface),
— l'héritage pourra avoir lieu entre classes ou entre interfaces (mot clé **extends**),
— le mot clé **extends** appliquera la relation d'implémentation entre une classe et une interface,
— les interfaces seront composées uniquement de méthodes publiques (pour simplifier, elles ne pourront contenir des attributs),
— le système de types sera composé des types primitifs (**int**, **float**, etc.), des classes et des interfaces.
— les attributs des classes pourront être de n'importe quel type du système de type,
— la visibilité des membres (attributs et méthodes) pourra être : public, protégé, privé (avec la même sémantique que Java),
— ils pourront être des attributs de classe (mot clé **static**),
— les types primitifs seront toujours composés et transmis par valeur,
— les autres types seront composés et transmis par référence par défaut (comme en Java)
— les types non primitifs pourront être composés et transmis par valeur grâce au mot clé **value**
— les méthodes seront définies de manière identique à Java excepté que les paramètres pourront être préfixé du mot clé **value** (transmission par valeur au lieu de référence).
— le mot clé **final** empêche la dérivation d'une classe ou d'une interface, la surcharge d'une méthode ou l'affectation d'une variable après son initialisation.
— les constructeurs porteront le même nom que la classe
— les méthodes de finalisation seront appelées **finalize()**.
— l'opérateur point (kw .) permet l'accès à un membre (attribut ou méthode) à partir d'une référence.
— la présence de parenthèse après une référence à une méthode dénote un appel.
— les instructions de contrôle de flux supportées seront : **if/else**, **for**, **while/do**, **switch**, l'opérateur ternaire `?:`, **try/catch**, avec la même sémantique et syntaxe que Java.
— Les exceptions n'auront pas besoin d'être vérifiée comme en Java (checked).
— les opérateurs arithmétiques seront `+ - * / mod ++ - += -=`, et ne s'appliqueront qu'aux types primitifs. Les priorités usuelles s'appliqueront et des parenthèses dans les expressions permettront de les contraindre.

— L'affectation (opérateur =) donnera lieu à un contrôle de type (compatibilité type statique/type dynamique).

2 Fonctionnalités du code

2.1 Gestion de la mémoire

L'allocation et la libération de la mémoire sera automatique. Seule l'allocation sera explicite à travers l'opérateur **new**.

2.2 Polymorphisme

2.2.1 Polymorphisme *ad hoc*

C'est le polymorphisme qui consiste à choisir l'implémentation d'une méthode à l'appel en fonction de sa signature. Exemple :

```
public class Test {
        // Methode A
        public int add(int a, int b) {
                return a+b;
        }

        // Methode B
        public Integer add(Integer a, Integer b) {
                return a+b;
        }

        // Rappel :
        // public Integer add(int a, int b) {...}
        // Ne peut pas exister car meme signature que
        // int add(int a, int b)

        public test()
                int a=1;
                int b=2;
                Integer A=new Integer(1);
                Integer B=new INTEGER(2);

                add(a, b); // Appel de la methode A.
                add(A, B); // Appel de la methode B.
}
```

Le sous-type des paramètres peut intervenir sur le polymorphisme *ad hoc*. Exemple :

```
class A {  }
class B extends A {  }
class C extends A {  }
class D extends C {  }

public class Test {
        // Methode A
        public void fait(A a) { System.out.println("Appel A"); }

        // Methode B
        public void fait(C c) { System.out.println("Appel C"); }

        public static void main(String[] args) {
                A a=new A();
```

```
                B b=new B();
                C c=new C();
                D d=new D();
                A e=new D();
                Toto t=new Toto();

                t.fait(a); // Appel de la methode A.
                t.fait(b); // Appel de la methode A.
                t.fait(c); // Appel de la methode C.
                t.fait(d); // Appel de la methode C.
                t.fait(e); // Appel de la methode A. ???
        }
}
```

- Le premier appel trouve une correspondance direct de type de paramètres.
- le seconde appel (type `B`) n'a pas de correspondance directe mais on peut voir statiquement qu'une implémentation existe pour un de ses ancêtres (en fait son parent immédiat),
- le troisième montre un choix à faire puisqu'une implémentation existe pour le même type amis aussi pour son parent, c'est le type le plus bas (le plus spécialisé) qui est choisi,
- le quatrième est similaire mais le type exacte n'est pas implémenté, là aussi, c'est la version la plus précise qui sera choisie.
- le cinquième montre que le choix, en java, est fait statiquement et ne prend pas en compte le type dynamique.

→ En *Kawa*, nous traiterons le polymorphisme *ad hoc* comme en *Java*, c'est à dire que, lorsqu'une méthode est surchargée dans une interface publique, le choix de la méthode à appeler sera déterminé à la compilation.

2.2.2 Polymorphisme de type

C'est le polymorphisme qui consiste à choisir l'implémentation d'une méthode à l'appel en fonction du type dynamique de la référence et non du type statique.

```
class A {
        public void fait() { System.out.println("Appel A"); }
}
class B extends A {
        public void fait() { System.out.println("Appel B"); }
}

public class Test {
        public static void main(String[] args) {
                A a=new A();
                B b=new B();
                A a2=new B();

                a.fait();  // Appel de la methode A.
                b.fait();  // Appel de la methode B.
                a2.fait(); // Appel de la methode B.
        }
}
```

Ce polymorphisme s'applique aussi à travers les interfaces car elles font parti du système de types :

```
interface Doing {
        public void fait();
}
```

```
class A implements Doing {
        public void fait() { System.out.println("Appel A"); }
}
class B implements Doing {
        public void fait() { System.out.println("Appel B"); }
}
class C extends B {
        public void fait() { System.out.println("Appel C"); }
}

public class Test {
        public static void main(String[] args) {
                Doing d1=new A();
                Doing d2=new B();
                Doing d3=new C();

                d1.fait();  // Appel de la methode A.
                d2.fait();  // Appel de la methode B.
                d3.fait();  // Appel de la methode C.
        }
}
```

2.3 Contrôle des types

Un système de types définit la manière dont sont catalogué les types des variables dans un langage, les règles permettant de les combiner et de déterminier leur compatibilité dans les affectations. En *Java*, le système de types est décomposé en deux grands espaces :

- un ensemble de types primitifs, comme **int, float,...**, qui ne peuvent être définit récursivement,
- un espace de classes qui sont organisées, par héritage, en une arborescence ayant pour racine la classe **Object**.
- Toutefois, les interfaces définissent un espace de type supplémentaire en dehors de cette arborescence. Il s'agit cette fois d'un graphe orienté sans cycle qui est créé par les relations d'héritage ou d'implémentation.

Les types ne sont compatibles que s'il existe une relation de sous-typage non stricte entre eux. Par exemple, l'instruction :

```
// B b;
A a=b;
```

n'est valide que si le type de b est un sous-type de A. Plus précisément :

- A et B sont des classes et B est un descendant direct ou indirect de A par héritage,
- A est une interface et B (ou un de ces ancêtres) implémente A,
- A et B sont des interfaces et B hérite (directement ou non) de A.

On précise que le fait de présenter toutes les conditions requises pour implémenter une interface ne suffit pas pour en être un implémenteur, il faut le dire explicitement :

```
public interface I {
        public int doThings(int x);
}

public class A {
        public int doThings(int x) { return x+1; }
}
...
// code de test :
A a=new A();
```

```
int y=a.doThings(10);    // Fonctionne parfaitement

// Mais A n'est pas un sous-type de I,
// car A n'implemente pas I :
I i=a;  // Erreur !
```

3 Implémentation en application binaire monolithique

Dans cette version, nous produisons un seul binaire exécutable. Tous les composants (*Kawa*) sont inclus et donc le binaire n'est pas dépendant de composant externe qui pourraient évoluer (exemple : changement de signature d'une méthode) et n'est donc pas tenu de faire certains contrôles à l'exécution (puisqu'il n'y a pas de processus de liaison dynamique du code).

Cependant, le comportement dynamique du langage nécessite l'ajout de structures de données et de code, notamment en ce qui concerne les appels de méthodes polymorphes. Exemple : nous savons que le compilateur ne peut pas générer statiquement le code d'appel à une méthode sur une référence car il ne connaît pas le type dynamique. En revanche le type statique constitue une interface publique des opérations possibles (que ce type soit une interface ou une classe).

3.1 Appels de méthode

Si on laisse pour l'instant de coté le polymorphisme, l'appel d'une méthode sur un objet ne diffère pas beaucoup d'un apple classique dans une programme structuré. Il s'agit du branchement en mémoire à une adresse contenant le code compilé de la fonction. L'objet sur lequel la méthode s'applique est alors vu comme un paramètre implicite (en général le premier). Un appel de méthode en programmation orientée objets :

```
class A {
        private int x;
        public void doThing(int arg1, int arg2) {...}
}
...
A a=new A();
a.do(arg1, arg2);
```

est implémenté de manière équivalente à un appel de fonction en programmation non-objet :

```
struct A {
        int x;

};
void doThing(struct A* arg0, int arg1, int arg2) {...}
...
struct A a;
a.do(&a, arg1, arg2);
```

Ce qui se traduirait donc également en code machine (ou en représentation intermédiaire) par quelque chose ressemblant à :

```
push  arg2     % parametres par valeur
push  arg1
push  @a       % parametre implicite, par adresse
call  @do      % saut à l'adresse de la fonction do
```

3.1.1 Convention de nommage (name decoration/mangling)

Cependant, le paradigme objet permet à une méthode de recevoir plusieurs implémentations. Le cas le plus évident à prendre en compte est la surcharge :

```
public class A {
        ...
        public int next(int i) { return i+1; }
        public int next(Float f) { return f.intValue()+1; }

}
```

La classe `A` comporte deux implémentations (avec deux signatures différentes) de la méthode `next`. Il nous faut pouvoir les distinguer symboliquement. Pour cela, on utilise la décoration de noms (*name decoration* ou *name mangling* en anglais) pour dénoter leurs implémentations respectives dans le code intermédiaire, et même dans le binaire final (fichier `.elf`). On pourrait par exemple adopter la convention de suffixer le nom de la méthode par le type de tous ses paramètres avec un séparateur, par exemple le signe `$`, ce qui donnerait :

```
call  @next$int                                % pour int next(int i)
call  @next$kawa.lang.Float                    % pour int next(Float f)
call  @do$kawa.util.List$kawa.lang.Integer     % pour do(List, Integer)
```

Il faut maintenant également tenir compte du fait que plusieurs méthodes avec le même nom peuvent être définie dans des classes différentes (indépendamment de toute considération polymorphique). Par exemple si l'on a :

```
package toto;
public class A {
        public int next(int i) { return i+1; }
}
```

ainsi que

```
package titi;
public class B {
        public int next(Float f) { return f.intValue()+1; }
}
```

il semble raisonnable d'utiliser comme préfixe leurs espaces de nom respectifs pour les distinguer. Ce qui pourrait donner :

```
call  @toto.A.next$int                  % pour toto.a.next(int i)
call  @titi.B.next$kawa.lang.Float      % pour titi.B.next(Float f)
```

Note : d'après mes expériences avec le compilateur `gcc`, les caractères `_$.£§°`μ^2 sont exploitables dans le format `ELF`.

3.1.2 Appels polymorphes (proposition)

L'exemple de code machine précédent montre un appel de méthode lié statiquement à la compilation. Ceci peut être obtenu si l'on sait, à la compilation que le type statique et le type dynamique sont les mêmes (par exemple appel sur une valeur). Mais dans le cas général, nous ne pouvons supposer le type dynamique et il nous faudra produire du code pour le déterminer à l'exécution et procéder à l'appel de la méthode adéquate. On peut donc considérer que, dans l'appel précédent, l'adresse de branchement est le résultat d'une fonction à deux paramètres, la signature de la fonction [1] et le type dynamique de l'objet :

$$fadr = f(sign, type)$$

1. Puisque nous avons décidé que la surcharge sera traitée statiquement comme en *Java*.

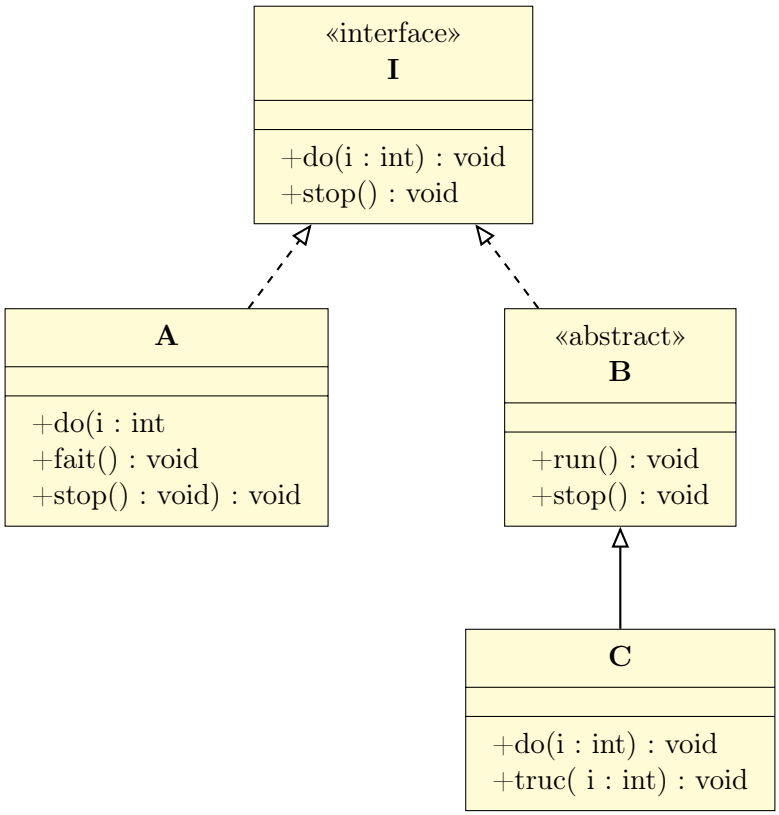
où *sign* représente la signature de la méthode, déterminée à la compilation et donc statique, et *type* est une référence à la classe de l'objet manipulé, obtenue à la compilation. Une technique couramment employée (cf C++) consiste à doter chaque objet d'un pointeur vers sa classe.

Les appels de méthode étant fréquents, il serait trop coûteux de faire appel à une sous-fonction pour déterminer l'adresse de branchement dynamique. En général on utilise une indexation de table en s'arrangeant pour que le paramètre qui sert d'indice soit contigu. On aurait le choix entre :

— indexer une table associée à chaque signature de méthode par un identifiant de classe : si l'on peut imaginer affecter à chaque classe un identifiant numérique séquentiel au chargement de celle-ci, on se rend bien compte que chaque table devra avoir autant d'entrées qu'il y a de classes chargées alors que très peu de ces dernières implémentent des méthodes avec la même signature. De plus le chargement dynamique d'une classe doit provoquer la mise à jour de toutes les tables.

— indexer une table associée à la classe de chaque instance avec un identifiant de signature. Les signatures peuvent être considérées par sous-ensemble, dans le cadre d'interfaces publiques ou nommées et donc leur nombre est connu et fixé au moment de la compilation.

La seconde approche est plus réaliste. Dans ce cas, on fera correspondre à chaque classe non pas une table, mais autant de table quelle présente d'interfaces déclarées, y compris son interface publique propre. Soit la hiérachie suivante :



Le code suivant met en œuvre le polymorphisme de type sur cette hierarchie et les commentaires indiquent la structure d'indexation utilisée. Par exemple `A.tableI` dénote la table de la classe `A` correspondant à l'interface `I` et `C.tableB` la table de la classe `C` correspondant à l'interface publique de la classe `B`.

```
public interface I {
        public void do(int i);
        public void stop(); ...
}
```

```
public class A implements I {
        public void do(int i) { ... }
        public void fait() { ... }
        public void stop() { ... }  ...
}

public abstract class B implements I {
        public void run() { ... }
        public void stop() { ... }  ...
}

public class C extends B {
        public void do() { ... }
        public void truc(int i) { ... }
}

... main() {
        I i=new A();
        i.do();          // appel de A.tableI[do] -> A.do()
        i=new B();
        i.do();          // appel de B.tableI[do] -> B.do()

        B b=new B();
        b.run();         // appel de B.tableB[run] -> B.run()
        b=new C();
        b.do();          // appel de C.tableB[do] -> C.do()
        b.run();         // appel de C.tableB[run] -> B.run()
}
```

Chaque interface publique ou nommée définit une indexation de ses méthodes :

IndexI	
do$int	0
stop	1

IndexA	
do$int	0
fait	1
stop	2

IndexB	
do$int	0
run	1
stop	2

IndexC	
do$int	0
fait	1
stop	2
truc$int	3

et chaque classe fournie une table correspondant à cette indexation :

— pour la classe `A` :

A.tableI	
0	A.do$int
1	A.stop

A.tableA	
0	A.do$int
1	A.fait
2	A.stop

— pour la classe `B` : cette classe étant abstraite, il est inutile de maintenir ces tables en mémoire car aucune instance n'y fera référence.

— pour la classe `C` :

C.tableI	
0	C.do$int
1	B.stop

C.tableB	
0	C.do$int
1	B.run
2	B.stop

C.tableC	
0	C.do$int
1	B.run
2	B.stop
3	C.truc

Il manque cependant une étape dans le processus de liaison dans ce qui vient d'être montré : comment retrouver une table à partir d'une référence/un type dynamique. Par exemple, comment retrouver `C.tableB` à partir de `b` dans la ligne :

```
        b.do();          // appel de C.tableB[do] -> C.do()
```

Le type statique de `b` (c'est à dire la classe `B`) nous indique bien que nous sommes à la recherche de `tableB`, et nous ne savons pas (statiquement) pour quelle classe. À l'exécution, nous pouvons obtenir la classe de `b` mais nous ne pouvons pas faire de supposition quant à la position d'un quelconque attribut `tableB` dans celle-ci puisque les classes implémentent un nombre variable d'interfaces. Comme précédement, il n'est pas envisageable d'indexer les tables de chaques classes avec des identifiants contigu d'interfaces. L'utilisation d'une table de hachage serait également contre-performant.
On propose de faire reposer la responsabilité sur les références.

3.1.3 Références à typage dynamique

Traditionnellement, une référence se résume à une adresse en mémoire (même en *Java*). Supposons que l'on définisse une référence comme étant un couple $(type, adresse)$ où *type* est l'adresse des méta-données d'un type, ici simplement l'adresse de la table des méthodes de ce type. Ainsi, chaque affectation choisirait la table dans le terme droit en fonction du type statique du terme gauche et un appel de méthode n'aura plus besoin d'indirection à travers l'objet pour atteindre sa classe mais pourra se faire directement à partir de la référence :

```
... main() {
        I i=new A();    // i=(@A.tableI, @newA)
        i.do();         // appel de i.type[do] = A.tableI[do] -> A.do()
        i=new B();      // i=(@B.tableI, @newB)
        i.do();         // appel de i.type[do] = B.tableI[do] -> B.do()

        B b=new B();    // b=(@A.tableB, @newB)
        b.run();        // appel de b.type[run] = B.tableB[run] -> B.run()
        b=new C();      // B=(@C.tableB, @NewC)
        b.do();         // appel de b.type[do] = C.tableB[do] -> C.do()
        b.run();        // appel de b.type[run] = C.tableB[run] -> B.run()
}
```

Nous devons faire face à une dernière [2] complication : lorsque le terme de droite est dynamique, alors le compilateur ne peut choisir la valeur du membre *type* de la référence affectée :

```
... main() {
        B b=new B();    // b=(@A.tableB, @newB)

        I i=b;          // i=(?, b.adresse) = (@B.tableI, @newB)
}
```

Comme nous savons statiquement que `B` est un sous-type de `I`, nous pourrions doter `B` d'une fonction implicite donnant accès à la table de `B` correspondant au type parent. Plus généralement : tout type (classe ou interface) imposera dans son interface publique la présence d'une méthode donnant accès à la table qui lui correspond et tout sous-type devra implémenter cette méthode. Par transitivité, cela s'applique également aux relations indirectes. Par exemple dans le cas de la classe `C` de l'exemple précédent, qui est la plus riche en terme d'interfaces implémentées (en code un peu mélangé) :

```
public interface I {
        ...
        // Methodes implicites :
        TypeTable[] getTableI();
}

public class C extends B { // et donc "implements I"
        ...
        // Methodes implicites :
```

2. Enfin j'espère ;-).

```
        TypeTable[] getTableI() { return @B.TableI; }
        TypeTable[] getTableB() { return @B.TableB; }
        TypeTable[] getTableC() { return @B.TableC; }
}
```

Ainsi, tout type dérivé de `I` comporte une méthode `getTableI()` qui lui est propre et présente dans toutes les tables des sous-types de `I` qui lui sont associées. Cette fois, on peut transtyper dynamiquement la référence :

```
... main() {
        B b=new B();    // b=(@A.tableB, @newB)

        I i=b;          // i=(b.type[getTableI](), b.adresse) = (@B.tableI, @newB)
}
```

Un inconvénient de cette technique est que les références ont une taille double. Cependant, si le code produit n'autorise pas de type de références sans type, alors on peut se dispenser d'équiper les objets d'un pointeur donnant accès à sa classe. Le coût mémoire est déplacé de l'objet vers la référence, ce qui signifie qu'il n'y a surcoût qu'en cas de références multiples vers une même objet.
Enfin, le maintient des fonctions d'accès complexifie légèrement la gestion des tables de méthodes mais le gain en efficacité des appels de méthodes devrait être significatif.

3.2 Agencement des objets en mémoire (layout)

L'agencement des attributs d'un objet en mémoire doit être défini une bonne fois pour toute à la compilation. De la connaissance de cet agencement va dépendre :

— l'accès aux attributs publiques,

— la construction de l'agencement des classes dérivées.

L'agencement doit prendre en compte le problème d'alignement en mémoire. Par exemple la plus petite unité d'allocation de mémoire est l'octet, même pour un booléen. On choisira pourtant de lui allouer plus de mémoire (ou plus précisément de le rembourrer) afin que la variable suivante démarre sur une adresse multiple de 16 ou 32 (voire 64 bits) afin que le processeur n'ai pas besoin de lire deux mots en mémoire au lieu d'un seul.

3.2.1 Attributs publiques

Les attributs publiques sont des variables d'un objet pour lesquelles ont accepte que du code externe obtienne une référence. Pour cela, l'adresse de la variable est en général calculée à partir de l'adresse de l'objet qui la contient augmenté de la position de la variable à l'intérieur de l'objet. Cette position est définie par la classe. Imaginons la classe suivante :

```
public class C {
        public char c;
        private int i;
        public bool b;
        public float f;
}
```

Ses attributs ont un décalage (offset) défini par le compilateur :

attributs de C	
0	c (1 octet)
1	bourrage (3 octets)
4	i (4 octets)
8	b (1 octet)
9	bourrage (3 octets)
12	f (4 octets)

Ce décallage sera utilisé par le compilateur pour calculer l'adresse d'accès direct :

```
... main {
        C c=new C();
        c.i++;  // (@c+4) = (@c+4)+1
}
```

3.2.2 Héritage

L'héritage entre deux classes va non seulement provoquer l'augmentation des interfaces par la classe dérivée, mais également donner lieu à une augmentation de l'agencement mémoire avec la contrainte que l'agencement de la partie héritée doit être conservée à l'identique puisque :

- les décallages des attributs publiques de la classe de base doivent rester les mêmes puisqu'ils peuvent être accédés à travers uen référence ayant pour type statique la classe de base ;
- les méthodes héritées de la classe de base doivent pouvoir trouver leurs attributs avec le même décallage par rapport à `this`.

L'héritage produit un système d'agencement fonctionnant comme des poupées russes. En dérivant la classe précédente :

```
public class D {
        private double d;
        public bool bb;
}
```

on obtient l'agencement suivant :

attributs de D		
super	0	c (1 octet)
	1	bourrage (3 octets)
	4	i (4 octets)
	8	b (1 octet)
	9	bourrage (3 octets)
	12	f (4 octets)
D	16	d (8 octet)
	24	bb (1 octet)
	25	bourrage (3 octets)

4 Implémentation en code partagé

A suivre ...