

E P I T E C H

Programmation Orientée Objet Avancée
Documentation de conception du projet KOOC

Alexis Mongin	Géraldine Heuse	Christopher Paccard
Pierre Rousselle	Benoit de Chezelles	David Galy

13 novembre 2016

Table des matières

1	Introduction & fonctionnalités	3
1.1	Les modules	3
1.2	Le polymorphisme ad hoc	3
1.3	Programmation orientée objet	3
2	Modules	3
2.1	Import de module	3
2.2	Compilation des implémentations de module	3
3	Types abstraits : classes	4
3.1	Représentation en mémoire, fonctions membres & implémentation	4
3.2	Allocation & forme canonique	4
3.3	Héritage	5
3.3.1	Structure en mémoire	5
3.3.2	Métadonnées et polymorphisme	5
3.3.3	Type Object	5
4	Généralités	6
4.1	Compilation des fichiers .kh	6
4.2	Librairie de runtime	6
4.3	Décoration des symboles	6
4.4	Inférence de type	7
4.4.1	Invariance	7
4.4.2	Covariance et Contravariance	8
4.4.3	Résolution de type	9
5	Annexe	10
5.1	Diagramme FAST	10
5.2	Module	11
5.3	Class	13
5.4	Heritage	18
5.5	Test inferences	23

1 Introduction & fonctionnalités

Le présent document constitue la documentation de conception d'un compilateur d'un langage orienté objet étendant la grammaire du C, le KOOC (Kind Of Objective C) vers C. Nous détaillerons ici les différentes fonctionnalités du langage. En annexe, nous proposons un exemple de fichier à compiler.

1.1 Les modules

Un module désigne un ensemble de symboles (variables et fonctions) lui appartenant. Semblable au namespace du C++, il est composé en terme de code d'un bloc de définition `@module` contenant ses variables et ses prototypes de fonctions; et d'un bloc `@implementation` contenant cette dernière. Les blocs `@implementation` sont situés dans les fichiers `.kc` et les fichiers contenant les blocs `@module` ont pour extension `.kh`, l'équivalent des headers C. Nous utilisons, dans une unité de compilation, la déclaration `@import` afin d'importer les headers KOOC de notre choix

1.2 Le polymorphisme ad hoc

Les modules peuvent posséder des fonctions avec le même nom, mais des signatures différentes. La signature d'une fonction est l'ensemble des types de ses paramètres ainsi que son type de retour¹. On implémente ainsi une forme de polymorphisme ad hoc.

1.3 Programmation orientée objet

Le KOOC est un langage de programmation orienté objet, fournissant à l'utilisateur l'abstraction nécessaire afin de développer selon ce paradigme. Cela se traduit par la présence de types abstraits, possédant des fonctions et des variables membres; pouvant hériter les uns des autres et possédant des fonctions virtuelles.

2 Modules

2.1 Import de module

La directive `@import` sera remplacée par un `#include` du fichier `.h` généré par la compilation du fichier `.kh` concerné. Pour éviter la double inclusion, une macro protectrice sera définie dans chacun des fichiers `.h` généré. L'import de module, au moment de la compilation, importera aussi les types KOOC tel que les classes, afin de pouvoir reconnaître les déclarations.

2.2 Compilation des implémentations de module

La compilation des blocs `@implementation` contenus dans les fichiers `.kc` se fera simplement en traduisant le nom des symboles `kooc` vers le nom des symboles C générés.

Définitions de module

Les blocs `@modules` contenus dans les fichiers `.kh` seront compilés en calculant d'abord les noms décorés des fonctions dans chaque module présent dans le fichier, puis en copiant leurs prototypes dans le fichier `.h` de sortie. Lorsqu'un même module est importé par plusieurs unités de compilation, ses variables ne doivent pas être copiées dans chacune d'elles, le cas échéant

1. Il est intéressant de noter que le type de retour ne fait pas partie des signatures de fonction en C++.

provoquant de multiples définitions et de terribles erreurs de linkage. On mettra alors dans le fichier .h généré une référence externe vers le symbole décoré, la définition du symbole prenant place dans le fichier généré par la compilation de l'implémentation. On notera que cela implique de toujours fournir l'implémentation d'un module, même si elle est vide.

Par exemple, une telle déclaration :

```
$>cat pouet.kh
@module pouet
{
    int answer = 42;
}
```

produira un fichier .h ressemblant à :

```
#ifndef POUET_KMOD_H_
# define POUET_KMOD_H_
    extern A_B_n_W_H_B_o___E_E_MODULE_5_pouet_6_answer;
#endif
```

et ajoutera la déclaration suivante à pouet.c :

```
int A_B_n_W_H_B_o___E_E_MODULE_5_pouet_6_answer = 42;
```

3 Types abstraits : classes

3.1 Représentation en mémoire, fonctions membres & implémentation

Les classes seront implémentées via des structures contenant toutes leurs variables membres. Ces structures seront nommées `kc_{classe}_instance` et seront marquées de l'attribut `__attribute__((packed))` pour préserver leur ordre lors de la compilation. Les implémentations des classes héritent des règles de décoration des modules ; la différence concernant le support des fonctions membres. Ces dernières seront transcrites en C comme des fonctions prenant en premier paramètre un pointeur vers une instance de la classe. Il existe aussi la possibilité de déclarer les fonctions membres sans le mot-clé `member`, en prenant le pointeur d'une instance en tant que premier paramètre. Une telle fonction sera interprétée comme une fonction membre. Pour appeler une fonction membre, il faudra ainsi une instance valide de la classe. Il est à noter que les fonctions et variables non-membres des classes se comportent comme des fonctions et variables appartenant à un module.

Dans le code généré par le compilateur, jamais ne sera manipulée directement la structure générée par la compilation de l'implémentation de la classe, mais plutôt une interface, contenant une instance de cette structure précédée d'un pointeur vers l'ensemble des métadonnées de la classe.

3.2 Allocation & forme canonique

Les fonctions `init` permettent de générer des constructeurs, étant appelés automatiquement lors de l'instanciation des structures implémentant les classes. Si une fonction `init` précédée du mot-clé `member` prenant en unique paramètre un pointeur vers une autre instance de la classe, ou une fonction `init` non précédée par `member` prenant deux pointeurs vers des instances de la classe n'est pas définie, on générera par défaut un constructeur par copie faisant appel à la fonction `memcpy`. La fonction `alloc` consistera en l'appel de la fonction allocatrice de mémoire

malloc, prenant en paramètre la taille de la structure de la classe via le mot-clé `sizeof`. La fonction `new` générera le même code que les appels respectifs d'`alloc` et `init`. La fonction `delete` générera un appel à la fonction décorée représentant la fonction membre `clean` si elle existe, puis à `free` afin de libérer la mémoire.

3.3 Héritage

Les classes peuvent hériter d'une unique classe-mère, et ceci récursivement, ce qui permettra l'implémentation d'arbres d'héritage classiques, mais pas d'héritage multiple.

3.3.1 Structure en mémoire

La structure `kc_{classe}_interface`, générée dans le fichier `.h` implémentant la structure de la classe, est l'interface mentionnée plus haut. Ses membres sont nommés `meta` et `instance`. Elle implémente le fait que chaque type hérite d'`Object`. En ce qui concerne la structure en mémoire des classes filles, on copiera au début de la classe fille l'ensemble des variables membres de la classe mère.

3.3.2 Métadonnées et polymorphisme

Dans le `.h` généré par le fichier `.kh` contenant la description de la classe, on trouvera deux autres structures, nommées respectivement `kc_{classe}_vtable` et `kc_{classe}_metadata`. On trouvera dans `metadata` deux pointeurs sur `char*`, un nommé `name` et un autre `inheritance_list`, puis une instance de la structure `kc_{classe}_vtable` nommée `vtable`. Dans les fichiers `.c` générés par la compilation des fichiers `.kc`, on trouvera deux chaînes de caractères globales, une indiquant le nom de la classe et une autre la liste d'héritage de cette chaîne. On y trouvera aussi une instance unique et globale de la structure `kc_{classe}_metadata`. Les adresses des deux chaînes de caractères seront contenues dans les deux premiers champs de l'instance des métadonnées.

On doit bien entendu préserver l'ordre des fonctions virtuelles dans la `vtable` de chaque type appartenant à une même liste d'héritage. Pour ceci, on ajoutera les pointeurs sur fonctions virtuelles apparaissant dans une classe fille, triés par l'indice lexicographique de la signature, à la suite des pointeurs déjà présents dans la classe mère.

Par exemple, pour la liste d'héritage $A \rightarrow B \rightarrow C$, pour `C`, on placera d'abord les pointeurs des fonctions virtuelles de `A` triés, puis ceux de `B` triés, puis ceux des nouvelles fonctions virtuelles de `C`.

La structure implémentant la `vtable` de la classe sera remplie en dépilant la liste d'héritage de cette classe : on affectera les pointeurs sur fonctions aux premiers symboles rencontrés correspondant à leurs signatures, en partant du type le plus bas dans la liste d'héritage. Ainsi, l'instance possédera toujours les méthodes définies le plus bas dans la liste d'héritage. En upcastant l'instance par la suite, ce sera donc bien les fonctions correctes qui seront appelées.

3.3.3 Type Object

Le type `Object` et ses méthodes `isKindOf()` et `isInstanceOf()` seront implémentées dans une librairie de runtime. Ces deux méthodes se référeront aux métadonnées de chaque instance de classe en vérifiant les informations présentées dans les champs `kc_{classe}_name` et `kc_{classe}_inheritance_list` des instances desdites métadonnées. On considère que toutes les classes héritent d'`Object`, elles doivent alors toutes implémenter un destructeur virtuel.

4 Généralités

4.1 Compilation des fichiers .kh

Les macros protectrices seront générées à partir du nom du fichier .kh de la sorte :

```
[nom_du_fichier]_KMOD_H_
```

4.2 Librairie de runtime

On livrera avec le compilateur les sources de la librairie de runtime “libkrt.a”, ainsi qu’un makefile permettant de la compiler sous les architectures i386 et amd64. Cette librairie devra être liée avec les fichiers objets générés par les fichiers .c générés par le compilateur. Elle implémentera les fonctions membres du type `Object` .

4.3 Décoration des symboles

Pour supporter le polymorphisme ad hoc ainsi que l’appartenance des symboles aux modules, on décorera les symboles selon leur signature. Un symbole défini en KOOC sera ainsi “traduit” en C, et la plupart des mécanismes de compilation relatifs aux modules seront basés sur cette substitution. On propose cette norme afin de faire correspondre les symboles KOOC aux symboles traduits :

```
symbole_decoree ::= virtual? portee '_' type+ '_' origin '_' identifier '_' identifier
;
virtual ::= 'V' '_'
;
portee ::= ['S' | 'A'] #static ou auto
;
origin ::= ["CLASS" | "MODULE" | "OBJECT"]
;
identifier ::= number '_' identifier # taille puis nom
;
type ::= 'B' '_' node '_' 'E'
;
node ::= [primaryType | composedType | funcType | qualType |
          pointerType | arrayType | parenType | noneType]
;
primaryType ::= 'n' '_' final '_' type
;
composedType ::= 'c' '_' final '_' type
;
funcType ::= 'f' '_' final '_' type
;
qualType ::= 'q' '_' ['c' | 'l'] '_' type #respectivement const et volatil
;
pointerType ::= 'p' '_' '_' type
;
arrayType ::= 'a' '_' '_' type
;
noneType ::= 'o' '_' '_'
```

```

;
parenType ::= 't' '_' type* '_' type
;
primitif ::= 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' |
            'K' | 'L' | 'M' | 'N' | 'O' | 'X' | 'Z'
;
final ::= ['T' | 'V' | 'U' | 'S' | 'W'] '_' [primitif | identifier]
; #respectivement type utilisateur, enum, union, struct, natif

```

Pour la règle “primitif”, on utilise le tableau de correspondance de décoration des types du C++, soit :

type	décoration
signed char	C
char	D
unsigned char	E
short	F
unsigned short	G
int	H
unsigned int	I
long	J
unsigned long	K
float	M
double	N
long double	O
void	X
elipsis	Z

4.4 Inférence de type

On pourrait se passer de préciser le type d’une expression en utilisant les casts KOOC via un mécanisme d’inférence de type. On présentera d’abord les différents types de résolutions de type, puis notre algorithme afin de déterminer le type d’une expression.

```

1 unsigned short f(unsigned short i)
2 {
3     return i;
4 }
5
6 int main()
7 {
8     unsigned short res_unsigned_short;
9
10    res_unsigned_short = f(a);
11
12    return 0;
13 }

```

4.4.1 Invariance

L’invariance représente le cas où un ou plusieurs symboles coïncident avec celui recherché et un seul a exactement le bon type.

Si nous prenons le cas où nous avons deux variables a telles que :

```
1 int a = 42;
2 unsigned short a = 42;
```

Nous allons chercher l'appel de la fonction `f` si une variable `a` existe avec le type demandé et donc lui donner la bonne variable avec le bon type, car celui-ci existe (la variable `a` *ligne 2* de type `unsigned short`).

4.4.2 Covariance et Contravariance

Types scalaires

Prenons, toujours dans le même contexte, le cas où nous avons deux variables `a` telles que :

```
1 char a = 42;
2 long a = 42;
```

Nous pouvons voir ici qu'aucune fonction `f` ne prend de variable de type `char` ou `long` et que par conséquent l'invariance ne marche pas.

Pour résoudre ce problème nous allons tout d'abord définir deux hiérarchies des types (une pour les entiers et une autre pour les décimaux) :

- `signed char < char < unsigned char < short < unsigned short < int < unsigned int < long < unsigned long`
- `float < double < long double`

Tout d'abord, nous allons chercher un type covariant dans la liste des entiers (le type initial attendu étant un entier), c'est-à-dire un type plus faible (une recherche vers la gauche de la liste). Pour chaque type, à partir du type demandé par la fonction, nous allons vérifier si une variable `a` existe pour le type en question. Dans ce cas nous allons chercher si la variable `a` existe pour les types `short`, `unsigned char`, `char`, `signed char`. Et nous allons donc trouver la variable `a` de type `char` que nous allons pouvoir passer en paramètre à la fonction `f` sans caster.

Dans le même cas, si la variable `a` de type `char` n'existe pas, nous avons uniquement la variable de type `long` de disponible. Nous allons donc effectuer dans un premier temps comme ci dessus la recherche d'un type covariant, mais cette fois sans trouver de variable correspondante. La recherche "vers la gauche" devient insuffisante. C'est là qu'intervient la contravariance. Pour nous la recherche de type contravariant revient à faire une recherche vers le plus fort (vers la droite). La recherche va donc chercher les types `int`, `unsigned int`, `long`, `unsigned long` et trouver la variable `a` de type `long`. Cependant, dans le cas de contravariance il convient de faire un cast.

Les types composés

Nous avons eu l'occasion de voir comment se passe l'inférence de type pour les types scalaires, mais rien sur les types composés.

Tout d'abord, qu'est-ce qu'un type composé? Ce sont des types instanciables composés de plusieurs autres types. Dans le cadre du KOOC, les types composés sont les classes (les modules n'étant pas instanciables).

Pour l'invariance, rien ne change par rapport aux types scalaires. En revanche, la problématique de la covariance et de la contravariance ne se pose qu'à partir du moment où nous instaurons un mécanisme d'héritage entre ces types. Au lieu d'avoir une liste de types scalaires, nous allons construire une liste d'héritage pour chaque type composé. Par exemple :


```
1 @class A
2 {}
3
4 @class B : A
5 {}
6
7 @class C
8 {}
```

L'équivalent de l'exemple ci-dessus est l'association de ces types en deux listes. Pour A et B la liste suivante : $A < B$

Cet exemple nous indique que le type de B est plus fort que le type de A (tout comme plus haut, le type unsigned int est plus fort que le type int).

C n'ayant de relation d'héritage avec aucun autre type, la liste ne contient donc que : C

Ce type ne peut donc pas se convertir en un autre.

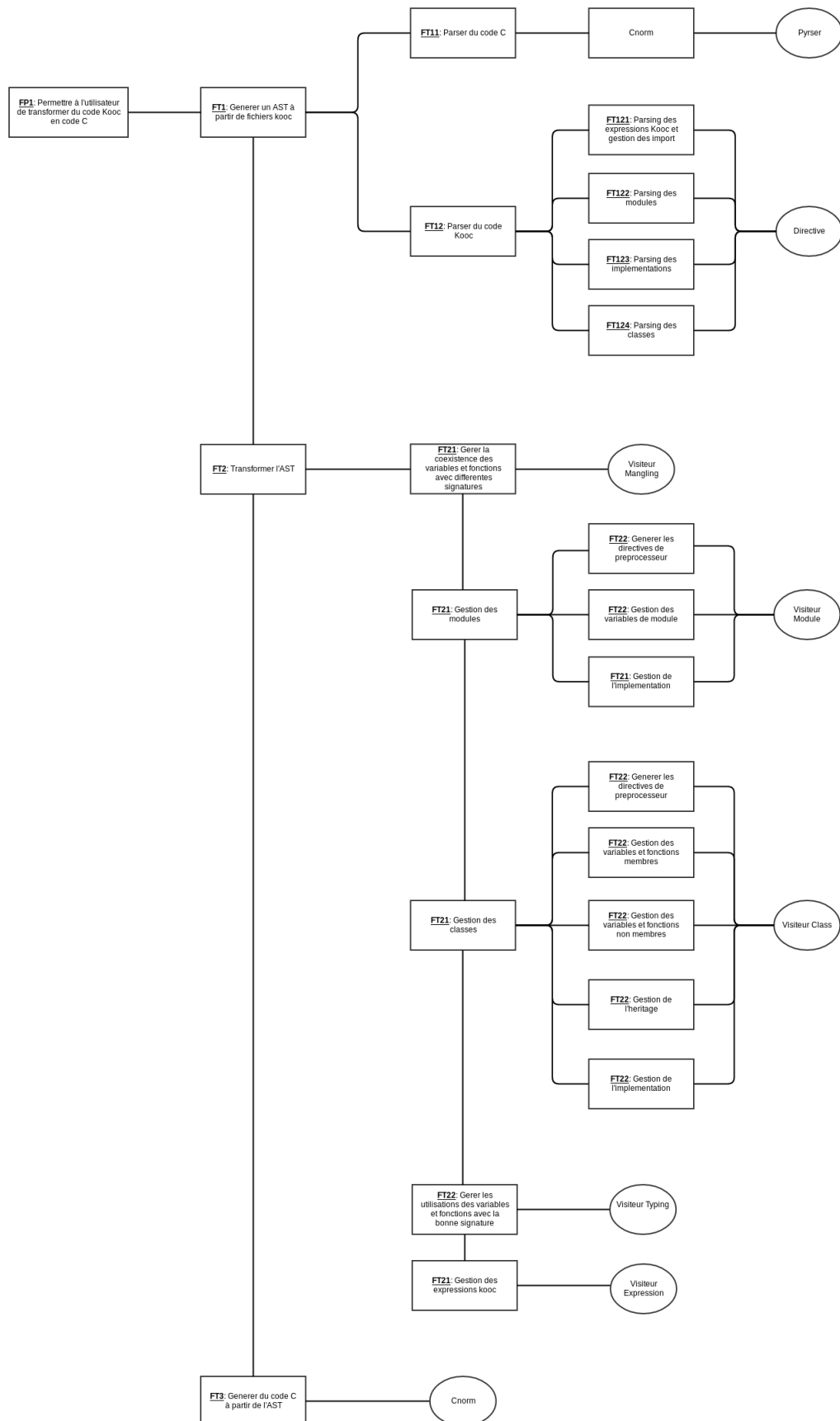
4.4.3 Résolution de type

Dans l'AST généré par un bloc de code, on remonte pour chaque opération toutes les signatures possibles jusqu'à la racine de l'expression. On essaiera de trouver une signature commune pour résoudre la racine, ce qui correspond à une résolution par invariance. Dans le cas où toutes les signatures remontées par les lvalue et rvalue ne donnent pas une signature commune, on testera une résolution par covariance. Si cela échoue, une résolution par contravariance sera testée, mais en affichant un message pour prévenir l'utilisateur de la perte potentielle de donnée due à la conversion. Puis la racine propagera le nouveau type de manière récursive à ses lvalue et rvalue, forçant ainsi le typage de l'AST.

Vous trouverez en annexe un test sur l'inférence de type scalaire.

5 Annexe

5.1 Diagramme FAST



5.2 Module

Module.kh

```
1  @module          Test
2  {
3      /* Variables relatif au module avec polymorphisme */
4      int           a;
5      char          a = 'a';
6      float         a = 42.0;
7      double        a;
8
9      static char    b = 'b';
10     static float   b = 21.0;
11
12     /* Fonctions relatif au module avec polymorphisme ad hoc */
13     int            f(int foo, float bar);
14     float          f(int foo, float bar);
15     void           f(const int *foo, const float *bar);
16     float          f(int foo);
17     float          f(float bar);
18     int            f(float bar);
19     int            f(int foo);
20     void           f(int *foo);
21     void           f(float *bar);
22 }
```

Module.h

```
1  /* Traduction du fichier Module.kh */
2
3  extern int       ~~~Test~~~int~~~a~~~;
4  extern char      ~~~Test~~~char~~~a~~~ = 'a';
5  extern float     ~~~Test~~~float~~~a~~~ = 42.0;
6  extern double    ~~~Test~~~double~~~a~~~;
7
8  int              ~~~Test~~~f~~~(int foo, float bar);
9  float            ~~~Test~~~f~~~(int foo, float bar);
10 void             ~~~Test~~~f~~~(const int *foo, const float *bar);
11 float            ~~~Test~~~f~~~(int foo);
12 float            ~~~Test~~~f~~~(float bar);
13 int              ~~~Test~~~f~~~(float bar);
14 int              ~~~Test~~~f~~~(int foo);
15 void             ~~~Test~~~f~~~(int *foo);
16 void             ~~~Test~~~f~~~(float *bar);
```

Module.kc

```
1  /* Gestion des import multiples */
2  @import "Test.kh"
3  @import "Test.kh" /* Gestion des commentaires en C */
4  @import "Test.kh" // Gestion des commentaires C++
5
6  @implementation Test
7  {
8      int    f(int foo, float bar)
9      {
10         return foo+((int)bar);
11     }
12
13     float f(int foo, float bar)
```

```

14 {
15     return ((float)foo)+bar;
16 }
17
18 void f(const int *foo, const float *bar)
19 {
20     printf("int: %i && float: %f\n", *foo, *bar);
21 }
22
23 float f(int foo)
24 {
25     return (float)foo;
26 }
27
28 float f(float bar)
29 {
30     return bar;
31 }
32
33 int f(float bar)
34 {
35     return (int)bar;
36 }
37
38 int f(int foo)
39 {
40     return foo;
41 }
42
43 void f(int *foo)
44 {
45     *foo = [Test.b];           /* Cast implicite */
46 }
47
48     f(float *bar)
49 {
50     *bar = [Test.b];
51 }
52 }

```

Module.c

```

1  /* Gestion des import multiples - traduction de Module.kc */
2  #include "test.h"
3  #include "test.h"
4  #include "test.h"
5
6  int             ~~~Test~~~int~~~a~~~;
7  char           ~~~Test~~~char~~~a~~~ = 'a';
8  float          ~~~Test~~~float~~~a~~~ = 42.0;
9  double         ~~~Test~~~double~~~a~~~;
10
11 static char     ~~~Test~~~char~~~b~~~ = 'b';
12 static float    ~~~Test~~~float~~~b~~~ = 21.0;
13
14 int             ~~~Test~~~f~~~(int foo, float bar)
15 {
16     return foo+((int)bar);
17 }
18
19 float           ~~~Test~~~f~~~(int foo, float bar)

```

```

20 {
21     return ((float)foo)+bar;
22 }
23
24 void      ~~~Test~~~f~~~(const int *foo, const float *bar)
25 {
26     printf("int: %i && float: %f\n", *foo, *bar);
27 }
28
29 float      ~~~Test~~~f~~~(int foo)
30 {
31     return (float)foo;
32 }
33
34 float      ~~~Test~~~f~~~(float bar)
35 {
36     return bar;
37 }
38
39 int      ~~~Test~~~f~~~(float bar)
40 {
41     return (int)bar;
42 }
43
44 int      ~~~Test~~~f~~~(int foo)
45 {
46     return foo;
47 }
48
49 void      ~~~Test~~~f~~~(int *foo)
50 {
51     *foo = ~~~Test~~~char~~~b~~~;
52 }
53
54 void      ~~~Test~~~f~~~(float *bar)
55 {
56     *bar = ~~~Test~~~float~~~b~~~;
57 }

```

5.3 Class

Class.kh

```

1 class Test
2 {
3     /* Member attributes */
4     @member
5     {
6         int            value = 42;
7         float          value = 42.0;
8     }
9     @member int        secret = 42;
10
11     /* Member functions with the 3 different syntax */
12     @member
13     {
14         int            get_value();
15         float          get_value();
16     }

```

```

17  @member void      set_value(int value);
18  @member void      set_value(float value);
19  char              *get_class_name(Test *);
20  void              generate_new_secret(Test *);
21
22  /* Some member functions for coplien form */
23  @member void      init(int value_int, float value_float);
24  void              init(Test *, float value_float, int value_int);
25
26  /* ! Polymorphism works also with member ! */
27
28  /* Class attributes */
29  static int         value = 42;
30  static float       value = 42.0;
31  int                secret = 42;
32
33  /* Class functions */
34  int                get_value();
35  float              get_value();
36  void               set_value(int value);
37  void               set_value(float value);
38  char               *get_class_name();
39  static void        generate_new_secret();
40 }

```

Class.h

```

1  /* Traduction du fichier Class.kh */
2  #ifndef TEST_H_
3  # define TEST_H_
4
5  typedef struct    kooc_s_Test {
6      int          ~~~Test~~~int~~~value~~~;
7      float        ~~~Test~~~float~~~value~~~;
8      int          ~~~Test~~~int~~~secret~~~;
9  }                Test;
10
11  int              ~~~Test~~~get_value~~~(Test *test);
12  float            ~~~Test~~~get_value~~~(Test *test);
13  void             ~~~Test~~~set_value~~~(Test *test, int value);
14  void             ~~~Test~~~set_value~~~(Test *test, float value);
15  char             *~~~Test~~~get_class_name~~~(Test *test);
16  static void      ~~~Test~~~generate_new_secret~~~(Test *test);
17
18  void             ~~~Test~~~init~~~(Test *test, int value_int, float value_float);
19  void             ~~~Test~~~init~~~(Test *test, float value_float, int value_int);
20  void             ~~~Test~~~init~~~(Test *test);
21
22  Test             *~~~Test~~~alloc~~~();
23  Test             *~~~Test~~~new~~~(int res_int, float res_float);
24  Test             *~~~Test~~~new~~~(float res_float, int res_int);
25  void             ~~~Test~~~delete~~~(Test *test);
26
27  extern int       ~~~Test~~~int~~~secret~~~;
28
29  int              ~~~Test~~~get_value~~~();
30  float            ~~~Test~~~get_value~~~();
31  void             ~~~Test~~~set_value~~~(int value);
32  void             ~~~Test~~~set_value~~~(float value);
33  char             *~~~Test~~~get_class_name~~~();
34  static void      ~~~Test~~~generate_new_secret~~~();

```

```

35
36 #endif /* !TEST_H_ */

```

Class.kc

```

1  @import "Test.kh"
2
3  @implementation Test
4  {
5      @member
6      {
7          int          get_value()
8          {
9              return [self.value];
10         }
11
12         float        get_value()
13         {
14             return [self.value];
15         }
16     }
17
18     @member void      set_value(int number)
19     {
20         [self.value] = number;
21     }
22
23     @member void      set_value(float number)
24     {
25         [self.value] = number;
26     }
27
28     void              get_class_name(Test *this)
29     {
30         [this generate_new_secret];
31         return "Test";
32     }
33
34     void              generate_new_secret(Test *that)
35     {
36         [that.secret] *= 2;
37     }
38
39     @member void      init(int value_int, float value_float)
40     {
41         [self.value] = value_int;
42         [self.value] = value_float;
43     }
44
45     void              init(Test *self, float value_float, int value_int)
46     {
47         [self.value] = value_float;
48         [self.value] = value_int;
49     }
50
51     int               get_value()
52     {
53         return [Test.value];
54     }
55
56     float             get_value()

```

```

57 {
58     return [Test.value];
59 }
60
61 void          set_value(int value)
62 {
63     [Test.value] = value;
64 }
65
66 void          set_value(float value)
67 {
68     [Test.value] = value;
69 }
70
71 void          get_class_name()
72 {
73     [Test generate_new_secret];
74     return "Test";
75 }
76
77 void          generate_new_secret()
78 {
79     [Test.secret] *= 2;
80 }
81
82 }

```

Class.c

```

1  #include "Test.h"
2
3  static int      ~~~Test~~~int~~~value~~~ = 42;
4  static float    ~~~Test~~~float~~~value~~~ = 42.0;
5  int             ~~~Test~~~int~~~secret~~~ = 42;
6
7  int             ~~~Test~~~get_value~~~(Test *test)
8  {
9      return(test->~~~Test~~~int~~~value~~~);
10 }
11
12 float           ~~~Test~~~get_value~~~(Test *test)
13 {
14     return (test->~~~Test~~~float~~~value~~~);
15 }
16
17 void            ~~~Test~~~set_value~~~(Test *test, int value)
18 {
19     test->~~~Test~~~int~~~value~~~ = value;
20 }
21
22 void            ~~~Test~~~set_value~~~(Test *test, float value)
23 {
24     test->~~~Test~~~float~~~value~~~ = value;
25 }
26
27 char            *~~~Test~~~get_class_name~~~(Test *test)
28 {
29     ~~~Test~~~generate_new_secret~~~(test);
30     return ("Test");
31 }
32

```



```

33 static void   ~~~Test~~~generate_new_secret~~~(Test *test)
34 {
35     test->~~~Test~~~int~~~secret~~~ *= 2;
36 }
37
38 void         ~~~Test~~~init~~~(Test *test, int value_int, float value_float)
39 {
40     ~~~Test~~~float~~~value~~~ = value_float;
41     ~~~Test~~~int~~~value~~~ = value_int;
42 }
43
44 void         ~~~Test~~~init~~~(Test *test, float value_float, int value_int)
45 {
46     ~~~Test~~~int~~~value~~~ = value_int;
47     ~~~Test~~~float~~~value~~~ = value_float;
48 }
49
50 void         ~~~Test~~~init~~~(Test *test)
51 {
52     ~~~Test~~~int~~~value~~~ = 42;
53     ~~~Test~~~float~~~value~~~ = 42.0;
54     ~~~Test~~~int~~~secret~~~ = 42;
55 }
56
57 Test         *~~~Test~~~alloc~~~()
58 {
59     Test *test;
60     test = malloc(sizeof(Test));
61     ~~~Test~~~init~~~(test);
62     return (test);
63 }
64
65 Test         *~~~Test~~~new~~~(int res_int, float res_float)
66 {
67     Test *test;
68     test = ~~~Test~~~alloc~~~();
69     ~~~Test~~~init~~~(test, res_int, res_float);
70     return (test);
71 }
72
73 Test         *~~~Test~~~new~~~(float res_float, int res_int)
74 {
75     Test *test;
76     test = ~~~Test~~~alloc~~~();
77     ~~~Test~~~init~~~(test, res_float, res_int);
78     return (test);
79 }
80
81 void         ~~~Test~~~delete~~~(Test *test)
82 {
83     free(test);
84 }
85
86 int          ~~~Test~~~get_value~~~()
87 {
88     return (~~~Test~~~int~~~value~~~);
89 }
90
91 float        ~~~Test~~~get_value~~~()
92 {

```

```

93     return (~~~Test~~~float~~~value~~~);
94 }
95
96 void      ~~~Test~~~set_value~~~(int value)
97 {
98     ~~~Test~~~int~~~value~~~ = value;
99 }
100
101 void      ~~~Test~~~set_value~~~(float value)
102 {
103     ~~~Test~~~float~~~value~~~ = value;
104 }
105
106 char      *~~~Test~~~get_class_name~~~()
107 {
108     ~~~Test~~~generate_new_secret~~~();
109     return ("Test");
110 }
111
112 static void ~~~Test~~~generate_new_secret~~~()
113 {
114     ~~~Test~~~int~~~secret~~~ *= 2;
115 }

```

5.4 Heritage

Heritage.kh

```

1  /*
2  ** Les peaux vertes sont des creatures belliqueuses.
3  ** Les orques sont primitifs et pratiquent le chamanisme.
4  ** Les gobelins sont plus intelligents, disciplines et
5  ** pratiquent la magie de bataille.
6  */
7
8  @class PeauxVertes
9  {
10
11      @member
12      {
13          int    pv;
14          int    attaque;
15          int    agilite;
16      }
17
18      @member void    init();
19
20      @virtual void    clean();
21      @virtual void    attaque();
22      @virtual void    magie();
23  }
24
25  @class Orque : PeauxVertes
26  {
27
28      @member void    init();
29
30      @virtual void    clean();
31      @virtual void    attaque();
32      @virtual void    magie();

```

```

32 }
33
34 @class Gobelins : PeauxVertes
35 {
36     @member
37     {
38         int parchemins; //necessaire pour jeter une boule de feu
39     }
40
41     @member void init();
42     @virtual void clean();
43
44     @virtual void magie();
45 }

```

Heritage.h

```

1  /* Traduction du fichier Heritage.kc */
2
3  #ifndef PEAUXVERTES_KMOD_H_
4  # define PEAUXVERTES_KMOD_H_
5
6  typedef struct __attribute__((packed)) kc_PeauVertes_instance
7  {
8      int ~~~PeauxVertes~~~int~~~pv~~~;
9      int ~~~PeauxVertes~~~int~~~attaque~~~;
10     int ~~~PeauxVertes~~~int~~~agilite~~~;
11 }                                     kc_PeauxVertes_instance;
12
13 typedef struct kc_PeauxVertes_vtable
14 {
15     void (*~~~PeauxVertes~~~attaque~~~)(PeauVertes *self);
16     void (*~~~PeauxVertes~~~clean~~~)(PeauVertes *self);
17     void (*~~~PeauxVertes~~~magie~~~)(PeauVertes *self);
18 }                                     kc_PeauxVertes_vtable;
19
20 typedef struct kc_PeauxVertes_metadata
21 {
22     char *name;
23     char *inheritance_list;
24     kc_PeauxVertes_vtable vtable;
25 }                                     kc_PeauxVertes_metadata
26
27 typedef struct kc_PeauxVertes_interface
28 {
29     kc_PeauxVertes_metadata *meta;
30     kc_PeauxVertes_instance instance;
31 }                                     kc_PeauxVertes_interface
32
33 void ~~~PeauxVertes~~~init~~~(PeauxVertes *self);
34 void ~~~PeauxVertes~~~clean~~~(PeauxVertes *self);
35 void ~~~PeauxVertes~~~attaque~~~(PeauxVertes *self);
36 void ~~~PeauxVertes~~~magie~~~(PeauxVertes *self);
37
38 /* Orque */
39 typedef struct __attribute__((packed)) kc_Orque_instance
40 {
41     int ~~~PeauxVertes~~~int~~~pv~~~;
42     int ~~~PeauxVertes~~~int~~~attaque~~~;
43     int ~~~PeauxVertes~~~int~~~agilite~~~;
44 }                                     kc_Orque_instance;

```

```

45
46 typedef struct    kc_Orque_vtable
47 {
48     void (*~~~Orque~~~attaque~~~)(Orque *self);
49     void (*~~~Orque~~~clean~~~)(Orque *self);
50     void (*~~~Orque~~~magie~~~)(Orque *self);
51 }
52     kc_Orque_vtable;
53
54 typedef struct          kc_Orque_metadata
55 {
56     char                *name;
57     char                *inheritance_list;
58     kc_Orque_vtable     vtable;
59 }
60     kc_Orque_metadata
61
62 typedef struct          kc_Orque_interface
63 {
64     kc_Orque_metadata    *meta;
65     kc_Orque_instance    instance;
66 }
67     kc_Orque_interface
68
69 void    ~~~Orque~~~init~~~(Orque *self);
70 void    ~~~Orque~~~clean~~~(Orque *self);
71 void    ~~~Orque~~~attaque~~~(Orque *self);
72 void    ~~~Orque~~~magie~~~(Orque *self);
73
74 /* Gobelins */
75 typedef struct __attribute__((packed)) kc_Gobelin_instance
76 {
77     int    ~~~PeauxVertes~~~int~~~pv~~~;
78     int    ~~~PeauxVertes~~~int~~~attaque~~~;
79     int    ~~~PeauxVertes~~~int~~~agilite~~~;
80     int    ~~~Gobelin~~~int~~~parchemin~~~;
81 }
82     kc_Gobelin_instance;
83
84 typedef struct    kc_Gobelin_vtable
85 {
86     void (*~~~Gobelin~~~attaque~~~)(Gobelin *self);
87     void (*~~~Gobelin~~~clean~~~)(Gobelin *self);
88     void (*~~~Gobelin~~~magie~~~)(Gobelin *self);
89 }
90     kc_Gobelin_vtable;
91
92 typedef struct          kc_Gobelin_metadata
93 {
94     char                *name;
95     char                *inheritance_list;
96     kc_Gobelin_vtable    vtable;
97 }
98     kc_Gobelin_metadata
99
100 typedef struct          kc_Gobelin_interface
101 {
102     kc_Gobelin_metadata    *meta;
103     kc_Gobelin_instance    instance;
104 }
105     kc_Gobelin_interface
106
107 void    ~~~Gobelin~~~init~~~(Gobelin *self);
108 void    ~~~Gobelin~~~clean~~~(Gobelin *self);
109 void    ~~~Gobelin~~~magie~~~(Gobelin *self);
110
111 #endif

```

```

1  #include <stdio.h>
2
3  @implementation PeauxVertes
4  {
5      @member void init()
6      {
7          this->pv = 0;
8          this->attaque = 0;
9          this->agilite = 0;
10     }
11
12     @virtual void clean() {}
13
14     @virtual void attaque()
15     {
16         puts("Le peau-verte attaque !");
17     }
18
19     @virtual void magie()
20     {
21         puts("Le peau-verte lance un sort !");
22     }
23 }
24
25 @implementation Orque : PeauxVertes
26 {
27     @member void init()
28     {
29         this->pv = 16;
30         this->attaque = 15;
31         this->agilite = 8;
32     }
33
34     @virtual void clean() {}
35
36     @virtual void attaque()
37     {
38         puts("L'orque attaque avec une massue !");
39     }
40
41     @virtual void magie()
42     {
43         puts("L'orque fait un rituel chamanique !");
44     }
45 }
46
47
48 @implementation Gobelin : PeauxVertes
49 {
50     @member void init()
51     {
52         this->pv = 10;
53         this->attaque = 9;
54         this->agilite = 18;
55         this->parchemins = 3;
56     }
57
58     @virtual void clean() {}
59

```

```

60     @virtual void    magie()
61     {
62         if (this->parchemins > 0)
63         {
64             puts("Le goblin jette une boule de feu !");
65             this->parchemins -= 1;
66         }
67         else
68             puts("Plus de parchemins !");
69     }
70 }

```

Heritage.c

```

1  #include <stdio.h>
2  #include "peauxVertes.h"
3
4  kc_PeauxVertes_metadata    PeauxVertes_metadata =
5  {
6      "PeauxVertes",
7      "",
8      {
9          &~~~PeauxVertes~~~attaque~~~,
10         &~~~PeauxVertes~~~clean~~~,
11         &~~~PeauxVertes~~~magie~~~
12     }
13 };
14
15 kc_Orque_metadata    Orque_metadata =
16 {
17     "Orque",
18     "PeauxVertes",
19     {
20         &~~~Orque~~~attaque~~~,
21         &~~~Orque~~~clean~~~,
22         &~~~Orque~~~magie~~~
23     }
24 };
25
26 kc_Gobelin_metadata    Gobelin_metadata =
27 {
28     "Gobelin",
29     "PeauxVertes",
30     {
31         &~~~Gobelin~~~attaque~~~,
32         &~~~Gobelin~~~clean~~~,
33         &~~~Gobelin~~~magie~~~
34     }
35 };
36
37 void    ~~~PeauxVertes~~~init~~~(kc_PeauxVertes_interface *obj)
38 {
39     obj->~~~PeauxVertes~~~int~~~pv~~~ = 0;
40     obj->~~~PeauxVertes~~~int~~~attaque~~~ = 0;
41     obj->~~~PeauxVertes~~~int~~~agilite~~~ = 0;
42 }
43
44 void    ~~~PeauxVertes~~~clean~~~(kc_PeauxVertes_interface *obj){}
45
46 void    ~~~PeauxVertes~~~attaque~~~(kc_PeauxVertes_interface *obj)
47 {

```

```

48     puts("Le peau-verte attaque !");
49 }
50
51 void   ~~~PeauxVertes~~~magie~~~(kc_PeauxVertes_interface *obj)
52 {
53     puts("Le peau-verte lance un sort !");
54 }
55
56 void   ~~~Orque~~~init~~~(kc_Orque_interface *obj)
57 {
58     obj->~~~PeauxVertes~~~int~~~pv~~~ = 16;
59     obj->~~~PeauxVertes~~~int~~~attaque~~~ = 15;
60     obj->~~~PeauxVertes~~~int~~~agilite~~~ = 8;
61 }
62
63 void   ~~~Orque~~~clean~~~(kc_Orque_interface *obj) {}
64
65 void   ~~~Orque~~~attaque~~~(kc_Orque_interface *obj)
66 {
67     puts("L'Orque attaque avec une massue!");
68 }
69
70 void   ~~~Orque~~~magie~~~(kc_Orque_interface *obj);
71 {
72     puts("L'orque fait un rituel chamanique !");
73 }
74
75 void   ~~~Gobelin~~~init~~~(kc_Gobelin_interface *obj)
76 {
77     obj->~~~PeauxVertes~~~int~~~pv~~~ = 16;
78     obj->~~~PeauxVertes~~~int~~~attaque~~~ = 15;
79     obj->~~~PeauxVertes~~~int~~~agilite~~~ = 8;
80     obj->~~~Gobelin~~~int~~~parchemin~~~ = 3
81 }
82
83 void   ~~~Gobelin~~~clean~~~(kc_Gobelin_interface *obj) {}
84
85 void   ~~~Gobelin~~~magie~~~(kc_Gobelin_interface *obj)
86 {
87     if (obj->~~~Gobelin~~~int~~~parchemin~~~ > 0)
88     {
89         puts("Le goblin jette une boule de feu !");
90         obj->~~~Gobelin~~~int~~~parchemin~~~ -= 1;
91     }
92     else
93         puts("Plus de parchemins !");
94 }

```

5.5 Test inferences

testInference.kh

```

1  @module Test
2  {
3      int    a = 42;
4      float a = 42.0;
5
6      void   f();
7      void   f(int foo);

```

```

8   void f(float bar);
9   void f(int foo, float bar);
10  void f(float bar, int foo);
11
12  int f();
13  int f(int foo);
14  int f(float bar);
15  int f(int foo, float bar);
16  int f(float bar, int foo);
17
18  float f();
19  float f(int foo);
20  float f(float bar);
21  float f(int foo, float bar);
22  float f(float bar, int foo);
23 }

```

testInference.kc

```

1  @import "testInference.kh"
2
3  int main()
4  {
5      int res_int;
6      float res_float;
7
8      [Test f]; /* void f() */
9      [Test f :42]; /* void f(int foo) */
10     [Test f :42.0]; /* void f(float bar) */
11     [Test f :42 :[Test.a]]; /* void f(int foo, float bar) */
12     [Test f :42.0 :[Test.a]]; /* void f(float bar, int foo) */
13
14     res_int = [Test f]; /* int f() */
15     res_int = [Test f :42]; /* int f(int foo) */
16     res_int = [Test f :42.0]; /* int f(float bar) */
17     res_int = [Test f :42 :[Test.a]]; /* int f(int foo, float bar) */
18     res_int = [Test f :42.0 :[Test.a]]; /* int f(float bar, int foo) */
19
20     res_float = [Test f]; /* float f() */
21     res_float = [Test f :42]; /* float f(float foo) */
22     res_float = [Test f :42.0]; /* float f(float bar) */
23     res_float = [Test f :42 :[Test.a]]; /* float f(float foo, float bar) */
24     res_float = [Test f :42.0 :[Test.a]]; /* float f(float bar, float foo) */
25
26     return 0;
27 }

```