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

21 octobre 2016

Table des matières

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 a 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 retour [1]. On implémente ainsi une forme de polymorphisme ad hoc.

1.3 Programmation orientée objet

Le KOOC est un langage de programmation orienté objet, fournissant a 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 chacunes 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 appeller 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 appellés 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 chaine. On y trouvera aussi une instance unique et globale de la structure `kc_{classe}_metadata`. Les addresses 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 libraire de runtime. Ces deux méthodes se réfèreront aux métadonnées de chaque instance de classe en vérifiant les informations présentent 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 linkée 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 tels 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 tels 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 :

```
@class A
{}

@class B : A
{}

@class C
{}
```

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.

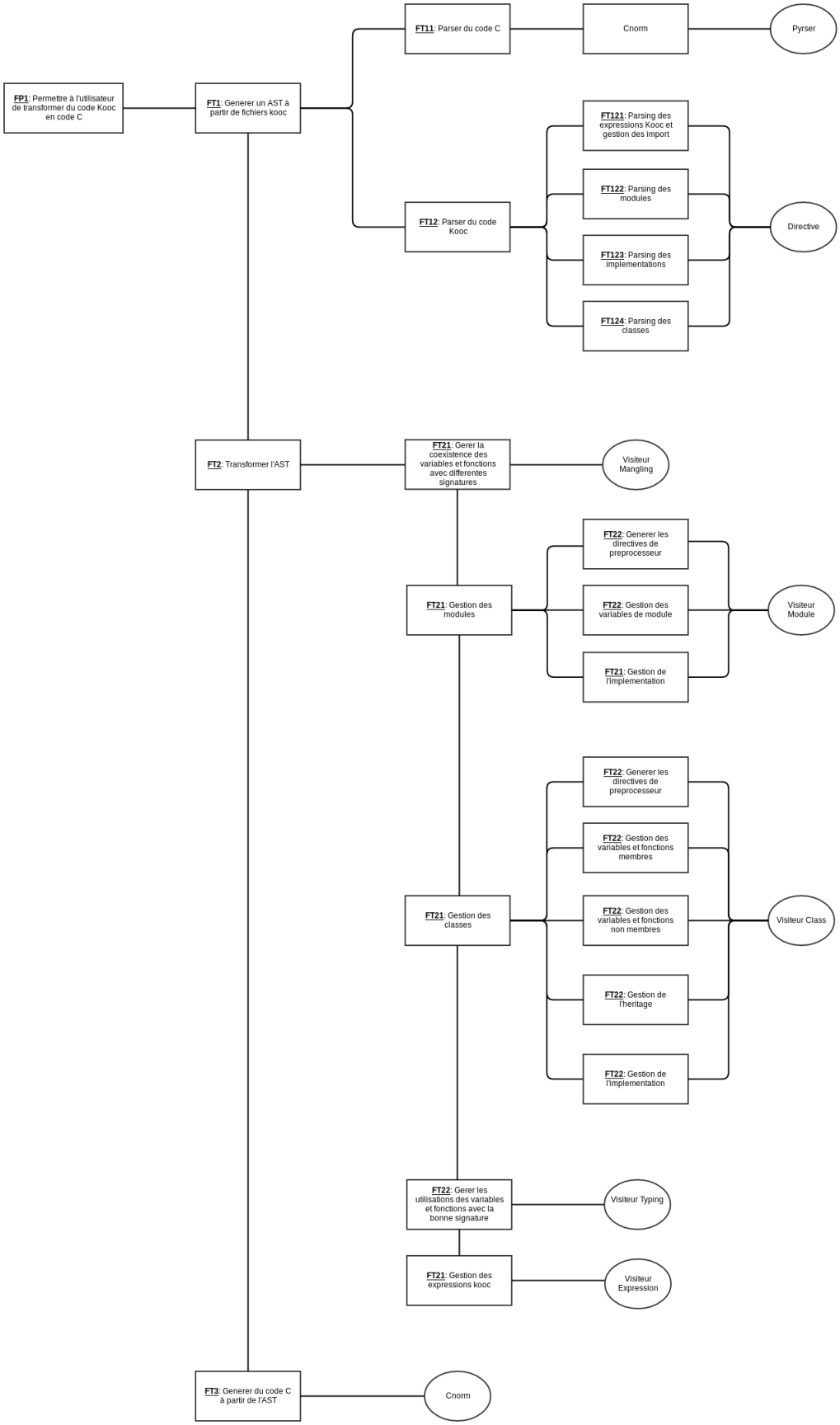
4.4.3 Résolution de type

Dans l'AST généré par un bloc de code, on remonte pour chaque opération toutes les signatures possible 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 recursive 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

```
@module          Test
{
  /* Variables relatif au module avec polymorphisme */
  int            a;
  char           a = 'a';
  float          a = 42.0;
  double         a;

  static char    b = 'b';
  static float   b = 21.0;

  /* Fonctions relatif au module avec polymorphisme ad hoc */
  int            f(int foo, float bar);
  float          f(int foo, float bar);
  void           f(const int *foo, const float *bar);
  float          f(int foo);
  float          f(float bar);
  int            f(float bar);
  int            f(int foo);
  void           f(int *foo);
  void           f(float *bar);
}
```

Module.kc

```
/* Gestion des import multiples */
@import "Test.kh"
@import "Test.kh" /* Gestion des commentaires en C */
@import "Test.kh" // Gestion des commentaires C++

@implementation Test
{
  int   f(int foo, float bar)
  {
    return foo+((int)bar);
  }

  float f(int foo, float bar)
  {
    return ((float)foo)+bar;
  }

  void  f(const int *foo, const float *bar)
  {
    printf("int: %i && float: %f\n", *foo, *bar);
  }

  float f(int foo)
  {
    return (float)foo;
  }

  float f(float bar)
  {
    return bar;
  }

```

```
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  }
```

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 differents 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    void                    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();
```

```
  float                 get_value();
  void                  set_value(int value);
  void                  set_value(float value);
  void                  get_class_name();
  static void           generate_new_secret();
}
```

Class.kc

```
@import "Test.kh"

@implementation Test
{
  @member
    {
      int       get_value()
      {
        return [self.value];
      }

      float     get_value()
      {
        return [self.value];
      }
    }

  @member void  set_value(int number)
  {
    [self.value] = number;
  }

  @member void  set_value(float number)
  {
    [self.value] = number;
  }

  void          get_class_name(Test *this)
  {
    [this generate_new_secret];
    return "Test";
  }

  void          generate_new_secret(Test *that)
  {
    [that.secret] *= 2;
  }

  @member void  init(int value_int, float value_float)
  {
    [self.value] = value_int;
    [self.value] = value_float;
  }

  void          init(Test *self, float value_float, int value_int)
  {
    [self.value] = value_float;
    [self.value] = value_int;
  }

  int           get_value()
  {
```

```
    return [Test.value];
  }

  float          get_value()
  {
    return [Test.value];
  }

  void           set_value(int value)
  {
    [Test.value] = value;
  }

  void           set_value(float value)
  {
    [Test.value] = value;
  }

  void           get_class_name()
  {
    [Test generate_new_secret];
    return "Test";
  }

  void           generate_new_secret()
  {
    [Test.secret] *= 2;
  }

}
```

5.4 Heritage

Heritage.kh

```
/*
** Les peaux vertes sont des creatures belliqueuses.
** Les orques sont primitifs et pratiquent le chamanisme.
** Les gobelins sont plus intelligents, disciplines et
** pratiquent la magie de bataille.
*/

@class PeauxVertes
{
                @member
        {
          int    pv;
          int    attaque;
          int    agilite;
        }

        @member void    init();

        @virtual void   clean();

        @virtual void   attaque();
        @virtual void   magie();
}

```

```
@class Orque : PeauxVertes
{
        @member void    init();

        @virtual void   clean();

        @virtual void   attaque();
        @virtual void   magie();
}

@class Gobelin : PeauxVertes
{
        @member
        {
          int   parchemins; //necessaire pour jeter une boule de feu
        }

        @member void    init();
        @virtual void   clean();

        @virtual void   magie();
}
```

Heritage.kc

```
#include <stdio.h>

@implementation PeauxVertes
{
        @member void init()
        {
          this->pv = 0;
          this->attaque = 0;
          this->agilite = 0;
        }

        @virtual void   clean() {}

        @virtual void   attaque()
        {
          puts("Le peau-verte attaque !");
        }

        @virtual void   magie()
        {
          puts("Le peau-verte lance un sort !");
        }
}

@implementation Orque : PeauxVertes
{
        @member void init()
        {
          this->pv = 16;
          this->attaque = 15;
          this->agilite = 8;
        }

        @virtual void   clean() {}

        @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 gobelin jette une boule de feu !");
65                this->parchemins -= 1;
66              }
67            else
68              puts("Plus de parchemins !");
69          }
70  }
```

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  }
```