Documentation: Projet Kooc

Victor Schuchmann, Alice Pesty, Nicolas Zordan et Arnaud Boulay

13 novembre 2016

Table des matières

Partie No. 1

Introduction

Le Kooc, ou Kind of objective C est un projet qui consiste en la création d'une surcouche au langage C, permettant d'y ajouter des fonctionalités présentes dans certains langages orienté object tels que le c++ ou le java. Ce projet est proposé dans le cadre du module KOOC, du 26 septembre au 20 novembre 2016. Nous avons realisé ce projet à l'aide des outils pyrser et Cnorm. Cette documentation permettra d'expliquer le fonctionnement de notre Kooc, ainsi que de partager les choix de conceptions qui ont été faits lors de sa réalisation.

Partie No. 2

Le fonctionnement

L'éxécutable Kooc est un compilateur qui va transformer des fichiers dont le langage source est le langage Kooc en langage C fonctionnel et compilable à partir d'un compilateur C tel que gcc.

Pour lancer la compilation, il suffit de passer en paramètre à l'éxécutable Kooc le nom d'un fichier Kooc (.kc ou .kh) tout autre extension sera ignorée.

Dans un premier temps, le compilateur va lire le fichier et le parser selon les différentes directives Kooc qui vont être rencontrées, pour générer un arbre de syntaxe comportant du code Kooc. Pour cela, on utilise

Une fois l'arbre généré, il va être transformé pour être rendu compatible au langage C.

Une fois l'arbre complètement transformé en fonctions des directives kooc, il va être utilisé pour générer du code C grace à l'outil to_c() de cnorm.

Partie No. 3

Les modules

3.1 Introduction

Dans un premier temps, nous avons dû réaliser l'implementation des directives kooc suivantes : @import, @module et @implementation.

Ce sont des directives qui sont propres à notre langage kooc et non au C, et donc non reconnues par Cnorm qui est un outil permettant de générer un arbre de syntaxe à partir de code C. Pour pouvoir inclure nos directives dans l'arbre généré par Cnorm habituellement, nous avons du modifier certaines règles de Cnorme et en ajouter d'autres.

Lorsque Cnorm parse un fichier écrit en C, la première règle appelée est la règle « translation_unit ». Elle appelle ensuite la règle « declaration » qui elle vérifie les déclarations en c, en assembleur ou bien les directives de préprocessing. Nous avons donc modifié cette règle en lui ajoutant les règles qui nous seront nécéssaires pour reconnaitre nos directives kooc. On a alors :

```
declaration = [
        Declaration.declaration
        | import_decl
        | module_decl
        | implem_decl
        | class_decl
]
```

Les 4 règles que nous avons ajouté nous permettent de récupérer les différentes directives kooc ainsi que le code écrit dans le scope de ces directives.

Les trois premières règles seront traitées dans cette premère partie, mais la suivante sera abordée dans les parties qui suivent.

3.2 @import

La directive Kooc @import est en quelque sorte l'équivalent Kooc de la directive de préprocessing #include en C, c'est à dire qu'elle permet d'importer un fichier à l'extension .kh dans un fichier à l'extension .kc, de la même manière que #include permet d'importer un fichier à l'extension .h dans un fichier d'extension .c. La différence entre les deux directives est que la directive @import doit empêcher d'elle même la double inclusion, ce que ne fait pas #include.

La syntaxe de la règle dans la BNF qui permet de récupérer la directive se présente de la manière suivante :

```
import_decl = [
        '@import' string :fichierAImporter #import_file(_, fichierAImporter)
]
```

Lorsqu'une directive @import est rencontrée, la première chose à faire est de remplacer dans l'arbre généré la directive kooc par la directive de préprocessing correspondante, soit #include suivi du nom de fichier que l'on a récupéré après la directive kooc en remplaçant l'extension .kh par l'extension .h.

Il faut ensuite lancer le parsing du fichier .kh grace à la fonction parse_file de pyrser, ce qui nous permettra d'obtenir un arbre à partir des elements du fichier .kh ainsi qu'un dictionnaire des modules et des classes importés. C'est à partir de ce dictionnaire que nous allons pouvoir vérifier que rien n'est inclu plusieurs fois.

3.3 @module

La directive Kooc @module sert à définir un scope dans lequel seront groupées dans un même ensembles des variables et des fonctions. Le module permet d'ajouter au C la possibilité de déclarer des variables et des fonctions possédant le même nom, tant que la signature est differente.

Pour récupérer la directive @module, on ajoute à la BNF la règle suivante :

```
module_decl = [
        '@module' id :nomModule
        '{'
        [
                [ c_decl ] :content
                #add_content(_, content)
        ]*
        '}'
]
```

Lorsque l'on rencontre la directive @module, la première étape est de parser le bloc de déclaration kooc pour générer un ast ensuite. Lors de ce parsing, plusieurs dictionnaires vont être créés

en plus de l'arbre : un contenant les noms des differentes variables, un second contenant le noms des differentes fonctions et enfin un contenant les variables définies dans le module. En effet ces dernières ne peuvent qu'être déclarées dans le module, la déclaration, elle, ne peut être fait que dans l'implémentation du module (grace à la directive @implementation).

La deuxième étape consiste en la transformation de l'arbre obtenu en un arbre traduisible en langage C, ce qui n'est pas le cas à la fin de l'étape de parsing. En effet, il est impossible en C de déclarer plusieurs variables ou fonctions du même nom, même lorsque les signatures diffèrent. C'est pourquoi les noms doivent êtres modifiés, c'est ce que nous appelons le mangling ou décoration des noms. Ce mangling suit une certaine grammaire :

_tailleNomModule
nomModule
_type

{Si c'est une fonction}
tailleNomFonction
nomFonction
_[typeParam]*

{Si c'est une variable}
tailleNomVar
nomVar

Et voici un exemple de variables et de fonctions manglées (toutes les fonctions et les variables sont contenues dans un module que l'on appelera test) :
int var1 –> _4test_i4var1
*volatile char *const var2 –> _4test_KPVc4var2*
short int func1(int p1) –> _4test_s5func1_i

Pour une structure :
*Struct s{int i; char *str;}*
*void *func2(struct s p1) –> _4test_Pv5func2_TiPc*

Enfin, une fois manglées, il faut ajouter aux variables déclarées le mot clé "extern" qui permet de définir ces variables en tant que variables globales au programme. Certains qualifiers tels que "const" ou volatile, lorsqu'ils ne sont pas rattachés à un pinteur ne sont pas pris en compte dans la décoration, car ils ne sont pas differanciables Pour un exemple de traduction de module en langage C ainsi que plus de précision sur les symboles de mangling, se référer à l'annexe 1.

3.4 @implementation

La directive kooc @implementation correspond à la définition des variables et des fonctions qui ont été déclarées dans le module ou la classe correspondante. Les variables qui avaient été définie dans le module/classe doivent donc être "déplacées" dans la partie implementation.

On récupère cette directive grace à la règle suivante :

```
implem_decl = [
        '@implem' id :nomModuleAImplem
        compound_statement :content
        #implementation(_, nomModuleAImplem, content)
]
```

Lors de l'implementation se posent plusieurs problemes. En effet, cette directive est ce qui va permettre d'utiliser ce qui à été déclaré dans un module ou une classe, or il a été dit que le module et la classe (dont nous parlerons dans la partie suivante) permettaient de déclarer plusieurs fonctions ou variables possèdant le même nom (mais pas la même signature). Cela étant impossible en C, nous avons en partie pallié au problème avec le mangling des noms, il se pose cependant toujours le problème de savoir quelle variante appeler et à quel moment.

En kooc, l'accès au variables et fonctions d'un module/classe se fait grace aux opérateurs crochets, et c'est à cet endroit précis que doit se déterminer le type de l'attribut auquel on veux accèder (voir la partie typage).

Partie No. 4

Les classes

4.1 Introduction

Dans cette deuxième partie, nous avons du réaliser les classes. A l'instar de la directive module, la directive @class ouvre un scope dans le quel il va etre possible de déclarer des variable et des fonction, à la seule différence que ces attributs vont maintenant avoir la possibilité d'être déclarés en tant qu'attributs membres, cela grace à la directive @member.

4.2 Définition

La classe est définie en Kooc par la directive @class et les attributs membres grace à @member (@member peut s'appliquer à un seul attribut ou à un bloc de déclaration, de la même manière que @module ou @class). Ces deux directives sont récupérées réspectivement grace aux règles suivantes :

```
classe_decl = [
        '@class' id :nomClasse
        [
                ' :' id :nomClasseParent
        ]
        '{'
        [
                [ c_decl | member_decl | virtual_decl ] :content
                #add_content(_, content)
        ]*
        '}'
]

member_decl = [
        '@member'
        '{'?
```

```
        [
                [ c_decl ] :content
                #add_content(_, content)
        ]*
        '}'?
]
```

Lorsque la directive @class est rencontre, le parsing va être éffectué de manière similaire à celui du module, à la différence que dans une classe, les attributs ayant été marqués comme membres vont êtres sauvegardés dans un autre dictionnaire, ce qui permettra de pouvoir créér une structure de donnée contenant les variables membres lors de la transformation de l'arbre pour en faire un arbre à partir du quel il est possible de générer du code C.

Les fonctions membres quant à elles sont traitées de la même manière que les attributs non membres éxcepté que ces fonctions on toutes pour premier paramètre un pointeur sur la classe, qui sera appelé __self, et que nous rajouterons nous même.

En ce qui concerne la partie mangling que nous avions vu lors de la récupération d'un module, elle est appliquée exactement de la même manière pour les attributs non membres ainsi que pour les fonctions membres, les variables membres quant à elles seront manglées à l'interieur de la structure, dont le nom sera, lui aussi, manglé.

4.3 Allocation et initialisation

Outre le fait de pouvoir déclarer des attributs membres, la classe possède une autre particularité, qui réside dans la manière dont elle est allouée et initialisée.

En effet, 5 fonctions doivent toujours être mise à disposition de l'utilisateur de la classe :

La fonction **alloc** :
: La fonction alloc quant à elle ne sert qu'à allouer de la mémoire à une instance de classe, et n'initialise pas le contenu de l'objet.

La fonction **init** :
: Cette fonction doit toujours être appelée lors de la création d'une instance d'une classe. Elle n'alloue pas de mémoire et sert uniquement à initialiser les attributs de la classe. L'utilisateur peut la définir en créant sa classe, lui faire prendre les paramètres de son choix ainsi que l'implémenter comme il le souhaite, toutefois, si l'utilisateur ne définie pas la fonction lui même, une fonction init par défaut doit être crée. Elle correspond à ce que l'on appelle un constructeur. C'est dans la fonction init que le pointeur sur la vtable sera initialisée à la valeure de la vtable correspondant au type de l'instance.

La fonction **new** :
: Cette fonction a pour but d'allouer de la mémoire à une instance de classe puis à

l'initialiser. Elle prend les mêmes paramètres que la fonction init() (qu'elle appelle) et retourne un pointeur sur une nouvelle instance de la classe en appelant la fonction alloc(). Il doit donc y avoir autant de fonction new que de fonctions init (une pour chaque signature)

La fonction **clean** :

Cette fonction est la fonction qui est appelée lorsqu'une instance de classe est détruite, elle correspond au destructeur. Dans cette fonction, les attributs auquels il a été attribué de la mémoire doivent être libérés. Cependant cette fonction n'a pas pour but de libérer la mémoire prise par l'instance de la classe, seulement son contenu.

La fonction **delete** :

La fonction delete est l'inverse de la fonction new. Là où la fonction new alloue et initialise, la fonction delete détruit les données de l'instance et libère l'espace qu'elle prend. Tout comme la fonction new appelle la fonction init, la fontion delete appelle la fonction clean.

Sont ajoutées à ces fonctions :

La fonction (name_of_interface) Cette fonction ne prend rien en paramètre et renvoie le nom du type réel de l'instance sous forme de chaine de char.

La fonction (isKindOf)

La fonction (isInstanceOf)

C'est après le parsing du bloc de la classe que l'on s'occupe de créer ces fonctions. Grace au dictionnaire de fonctions que l'on récupère après le parsing, il est possible de vérifier que la fonctions init a été déclarée au moins une fois. Si elle n'a jamais été déclarée, alors elle est créée par défaut avec les fonctions new et delete, clean et alloc puis elles sont injectées dans l'arbre.

Si au contraire elle à été définie une fois ou plus, alors pour chaque fonction init une fonction new est créée avec la même signature, c'est à dire avec les même paramètres. Elles sont ensuite injectées dans l'arbre avec les autres fonctions.

Pour un exemple de traduction de code Kooc en code C, se referer à l'annexe 2.

Partie No. 5

L'héritage et le polymorphisme

5.1 Introduction

Dans cette troisième partie, les fonctionalités que nous avons ajoutées au C consistent en la possibilité de faire hériter une classe d'une autre ainsi que le polymorphisme. Nous ne parlerons dans cette partie que d'héritage simple.

La syntaxe pour indiquer l'héritage d'une classe est " : nomClasseParent" après le nom de la classe fille. Pour le polymorphisme, on va utiliser la directive Kooc @virtual qui sera définie par la règle suivante :

```
virtual_decl = [
        '@virtual'
        '{'?
        [
                [ c_decl ] :content
                #add_content(_, content)
        ]*
        '}'?
]
```

Tout comme la directive @member, la directive @virtual peut s'appliquer à un bloc de fonctions ou bien à un seul attribut, cependant, à la différence de la directive @member, @virtual ne s'applique qu'aux fonctions, qui deviennent des fonctions membre.

5.2 Héritage

Lorsque nous récuperons une classe, la première chose qui va être éffectuée est de vérifier si elle hérite ou non d'une autre classe. Si tel est le cas, le nom de la classe parente vas être récupéré dans un noeud de l'arbre au moment du parsing. On vérifie également l'existence de cette classe parente

dans le dictionaire de classe que l'on a récupéré lors du parsing. Si la classe parente n'existe pas, on arrete le parsing, sinon le parsing continuera ensuite sur le reste de la classe.

Le parsing fini, on va créer un typdef sur le nom manglé de la structure avec son nom initial pour que le type existe dans d'autres fichiers. Lors de la création de la structure contenant les attributs membres de la classe, il sera vérifié si la classe hérite d'une autre classe et, si tel est le cas, les premiers éléments qui seront mis dans la structure seront les attributs de la classe parents, puis ceux de la classe fille, en ayant vérifié qu'aucun attribut ne soit déclaré plusieurs fois à l'interieur de chaque classe.

Une classe qui hérite d'une autre hérite non seulement de ses attributs mais aussi de ses fonctions. C'est pourquoi ce qui suit la création de la structure est la concaténation des dictionaires de fonctions membres de la classe parente et celui de la classe fille

5.3 Polymorphisme

Il se pose cependant un problème de redéclaration de fonctions, dans le cas ou la classe fille souhaiterait réimplémenter une fonction de la classe mère. En effet une classe fille peut réimplémenter une fonction de la classe mère, toutefois, il est possible de se retrouver dans le cas ou l'on possède un pointeur dont le type apparent est celui de la classe mère mais dont le type réel est en fait celui de la classe fille.

Dans ce cas la, si une fonction est redéfinie dans la classe fille, c'est à celle ci que nous voulons acceder, mais ayant un pointeur sur la classe mère, c'est à celle définie dans la classe mère que nous accedons.

Pour pallier à ce problème, on va utiliser la directive kooc @virtual, qui signifie que l'implémentation de la fonction déclarée en virtuelle écrase celle de la classe mère.

Afin de garder trace de quelle fonction appeller selon le type de la classe, chaque classe va contenir un pointeur sur une structure de pointeur sur fonction que l'on appelle vtable. Cette structure va contenir les addresses de toutes les fonctions virtuelles qui correspondent à l'instance et seulement ces fonctions. C'est au niveau de la fonction init que la vtable est initialisée, ainsi quel que soit le type apparent, ce sera toujours la fonction correspondant au type réel qui sera appelée.

Seront ajoutées à la vtable : le destructeur, la fonction name_of_interface(), les fonctions isKindOf() et isInstanceOf().
pour un exemple de traduction de code Kooc en code C, se référer à l'annexe 3.

Partie No. 6

Le typage

6.1 Introduction

Le Kooc, contrairement au C, permet la surcharge de variables et de fonctions. Nous devons donc fournir un moyen de trouver la bonne version manglée de la fonction ou de la variable à utiliser dans une situation précise en fonction du type attendu. C'est le rôle du typage.

Soit deux variables "nb" dans un module "M", l'une de type "int" l'autre de type "float". Si l'utilisateur veut affecter à cette variable nb la valeur entière "42" et qu'il ne précise pas le type de la variable à l'aide de la syntaxe "@!(TYPE)", alors, le compilateur Kooc doit comprendre que l'utilisateur cherche à stocker un type "int" et donc automatiquement fournir la version "int" manglée de la variable nb. Si l'utilisateur utilise la variable "nb" à l'intérieur d'un "if", "while" ou "for", la version "int" de la variable "nb" sera utilisée afin de simuler au plus proche un type booléen.

Cette démarche étant la même pour les fonctions et leurs types de retour, nous pouvons nous retrouver dans des situations de ce genre :

Dans le .kh

```
@module M
{
	int nb;
	float nb;

	int func(int, float);
	float func(float, float);
}

@implementation M
{
	int func(int i, float f)
	{
		return (i + (int)f);
```

```
        }

        float func(float f1, float f2)
        {
                return (f1 + f2);
        }
}
```

Dans le .kc

```
@import M.kh

int main(void)
{
        [M.nb] = 42;
        [M.nb] = 4.2;

        [M.nb] = [M func :42 :[M func :[M.nb] :4.2]];
        return (0);
}
```

Le compilateur Kooc doit alors déterminer dans quelle version de la variable "nb" stocker 42, 4.2 ainsi que le retour d'une version de la fonction "func". Cette démarche s'effectuera en deux étapes.

6.2 Stockage de données

La première étape consiste à garder trace de chaque variable et fonctions déclarées ainsi que lerus équivalents manglés. Pour cela nous utilisons plusieurs couches de dictionnaire que nous remplissons au moment du parsing afin de construire un arbre.

Concept d'arbre :
[ ] = dico
" " = str
( ) = liste

Soit :

Modules["module name" : (functions[ ], vars[ ])]

Un dictionaire de modules ayant pour clef le nom du module et pour valeur une liste de dictionnaire chacun décrit ci dessous.

fucntions["func name" : ["mangled func" : ["ret" : "ret type", "p1" : "p1 type", "p2" : "p2 type"]]]

Un dictionnaire de fonctions ayant pour clef le vrai nom de la fonction (pour accès rapide) et pour valeur un autre dictionnaire. Chaque sous dictionnaire porte comme clef une version manglée

du nom de la fonction et contient en valeurs les types de chaque paramètres et celui de la valeur de retour (sous forme d'un troisième sous dictionnaire).

Vars[“var name” : [“mangled var” : “type”, “mangled var” : “type”]]

Un dictionnaire de variable ayant pour clef le vrai nom de la variable (pour accés rapide)) et pour valeur un autre dictionnaire. Chaque sous dictionnaire porte comme clef une version manglée du nom de la variable et contient en valeurs le type de cette variable manglée.

Un fois l'arbre rempli, nous pouvons facilement accéder à chaque version manglée de chaque variable ou fonctions simplement grâce à son nom.

6.3 Intersection des types

Afin de retrouver la bonne version d'une variable ou fonction, il suffit de réaliser une intersection entre les types possibles et les types de notre arbre. Reprenons l'exemple en annexe.

Ce procédé se doit d'être récursif afin de traiter les cas les plus éloignés dans une expression en premier (la variable “nb” passé en paramètre du deuxième appel de fonction dans le cas de l'exemple) puis de remonter jusqu'au bout de l'expression même (l'assignation de variable “nb” dans l'exemple).

Lors de cas complexes, il se peut qu'une ambiguïté se créé et qu'il soit impossible de déterminer un type exacte à retourner. C'est le cas de la variable “nb” passé en paramètre du deuxième appel de la fonction dans l'exemple. Il est impossible de choisir entre sa version “int” ou “float” étant donné les deux versions de “func”. Cependant, nous savons que si “func” prend en premier paramètre un “int” il ne peut que prendre un “float” en deuxième paramètre. Nous pouvons donc en déduire par intersection que le deuxième appel à “func” en paramètre du premier appel ne peux retourner qu'un “float” et nous savons donc quelle version choisir ainsi que le type de “nb” en paramètre.

Pour résumer, en premier lieu, la résolution du paramètre “nb” retournera un bout de l'abre de données (dictionnaire) contenant toutes les versions possibles de “nb” pour cette résolution, puis lors de l'intersection de “func” passé en paramètre, les possibilités de “nb” dans le bout de dictionnaire seront ré-évaluée pour n'en choisir qu'une.

Si une fois arrivé à la fin de l'expression nous ne pouvons toujours pas déterminer le type et/ou n'avont qu'un bout de dictionnaire contenant les différentes possibilités, une erreur est alors lévée par le compilateur.

Voici un exemple absrait de ce que donnerait la résolution de cet exemple étape par étape :

nb = func(42, func(nb, 4.2)) ;
int || float = func(42, func(int || float, 4.2)) ;

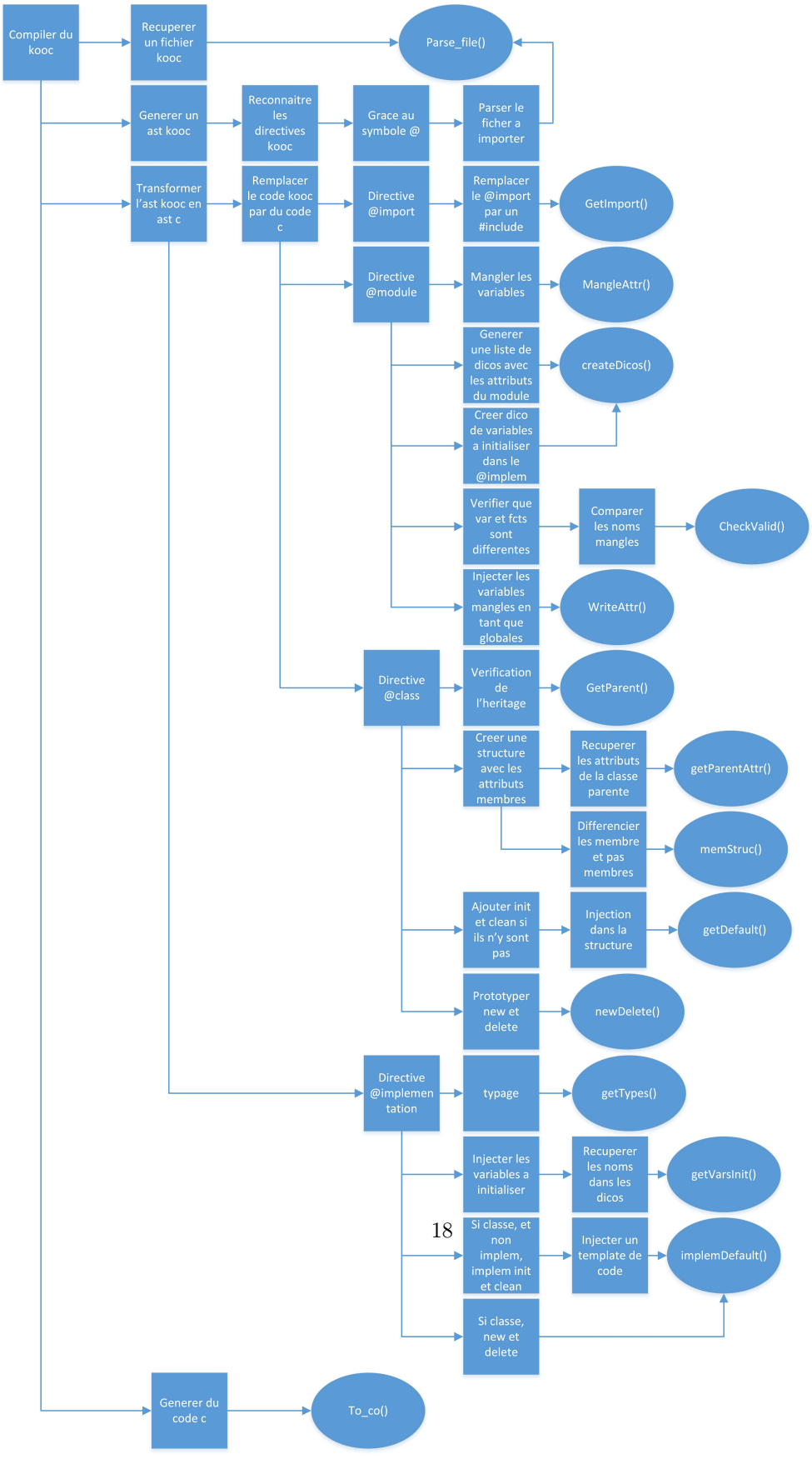
int || float = func(42, int(int || float, 4.2) || float(int || float), 4.2)) ;
int || float = func(42, float(int || float, 4.2)) ;
int || float = func(42, float(float, 4.2)) ;
int || float = int(42, float(float, 4.2)) ;
int = int(42, float(float, 4.2)) ;

Cette mécanique conclu donc la résolution de type pour le projet Kooc.

Partie No. 7

Annexe

7.1 Diagrame Fast



7.2 Traduction partie 1

7.2.1 Module

Un exemple de module :

```
@module Test
{
        void f(void) ;
        bool f(int nb1) ;
        int f(float nb1, float nb2) ;
        char *f(int nb) ;
        size_t f(const char *str) { return (strlen(str)) ; }

        typedef char un_test ;

        int a ;
        float a = 4.2 ;
        void *a = NULL ;
        char *const *volatile *a[] ;
}
```

Et sa traduction en C :

```
void _4Test_v1f_v(void) ;
bool _4Test_4bool1f_i(int nb1) ;
int _4Test_i1f_ff(float nb1, float nb2) ;
char *_4Test_Pc1f_i(int nb) ;
inline size_t _4Test_6size_t1f_KPc(const char *str) { return (strlen(str)) ; }
typedef char un_test ;
extern int _4Test_i1a ;
extern float _4Test_f1a ;
extern void *_4Test_Pv1a ;
extern char *const *volatile *_4Test_PPPPc1a[] ;
```

7.2.2 Tableau des symboles de mangling

Variable initiale	**Equivalence manglée**
void	v
char	c
unsigned char	C
unsigned int	I
unsigned short int	S
short int	s
int	i
long int	l
unsigned long int	L
long long	j
unsigned long long	J
float	f
double	d
long double	e
varargs	z
<expr>* OU <expr>* const	P<expr>
volatile <expr>*	PV<expr>
const	K
volatile	V
const volatile	KV
func pointer	P6 ReturnType [ParamType]*
struct	T [attributType]*
enum	E
union	U

7.2.3 Implementation

Un exemple d'implementation :

```
@implementation Test
{
        void f(void)
        {
                printf("coucou la moulinette");
        }

        bool f(int nb)
        {
                if (printf("NB = %d", nb))
                        return (false);
                return (true);
        }
```

```
        int f(float nb1, float nb2)
        {
                return ((int)(nb1 + nb2));
        }

        char *f(int nb)
        {
                return (malloc(sizeof(char)) * ([Test f :4.2 :4.3] * (int)@!(float)[Test a]));
        }
}
```

Et sa traduction en C :

```
int         _4Test_i1a;
float       _4Test_f1a = 4.2;
void        *_4Test_Pv1a = NULL;
char        *const *volatile *_4Test_PPPPc1a[];

void _4Test_v1f_v(void)
{
        printf("coucou la moulinette");
}

bool _4Test_4bool1f_i(int nb)
{
        if (printf("NB = %d", nb))
                return (false);
        return (true);
}
int _4Test_i1f_ff(float nb1, float nb2)
{
        return ((int)(nb1 + nb2));
}

char *_4Test_Pc1f_i(int nb)
{
        return (malloc(sizeof(char) * (_4Test_i1f_ff(4.2, 4.3)) * (int)_4Test_f1a));
}
```

7.3 Traduction partie 2

Un exemple de déclaration de classe :

```
@class StackInt
{
        @member
```

```
        {
                int         size;
                int         nbitem;

                void          init(int);
                void          init();
                void          clean();
                int         nbitem();
        }

    @member int         *data;
    @member void          push(int);
    int         pop(StackInt *);

    int         nbStack;
};
```

Et sa traduction en C :

```
typedef struct          _8StackInt_         StackInt;

void        _8StackInt_v4init_P8StackInti(StackInt *, int);
void        _8StackInt_v4init_P8StackInt(StackInt *);
void        _8StackInt_v5clean_P8StackInt(StackInt *);
int         _8StackInt_i6nbitem_P8StackInt(StackInt *);
StackInt        *_8StackInt_P8StackInt5alloc_v(void);
void        _8StackInt_v4push_P8StackInti(StackInt *, int);
void        _8StackInt_v6delete_P8StackInt(StackInt *);
int         _8StackInt_i3pop_P8StackInt(StackInt *);
StackInt        *_8StackInt_new_v4init_P8StackInti(int);
StackInt        *_8StackInt_new_v4init_P8StackInt();

struct          _8StackInt_
{
    int         _8StackInt_i4size_;
    int         _8StackInt_i6nbitem_;
    int         *_8StackInt_Pi4data_;
};

extern int          _8StackInt_i6nbStack_;
```

Un exemple d'implementation de classe :

```
@implementation StackInt
{
    @member int init(int size)
    {
        int *buf;
```

```
        [self.nbitem] = 0;
        [self.size] = size;
        buf = (int *)calloc(size, sizeof(int));
        [self.data] = buff;
}

@member int init()
{
        int *buf;

        [self.nbitem] = 0;
        [self.size] = 2048;
        buf = (int *)calloc(2048, sizeof(int));
        [self.data] = buff;
}

@member
{
        int nbitem()
        {
                int n;

                n = [self.nbitem];
                return (n);
        }

        void push(int i)
        {
                int pos;
                int *buf;

                pos = [self.nbitem];
                buf = [self.data];
                buf[pos++] = i;
                [self.nbitem] = pos;
        }
}

int pop(StackInt *self)
{
        int *buf;
        int pos;
        int r;

        pos = [self.nbitem];
```

```
                buf = [self.data];
                r = buff[--pos];
                [self.nbitem] = pos;
                return (r);
        }

        void clean(StackInt *this)
        {
                int *buf;

                buf = [this.data];
                free(buf);
        }
}
```

Et sa traduction en C :

```
int         _8StackInt_i6nbStack_;

void _8StackInt_v4init_P8StackInti(StackInt *__self, int size)
{
        int *buf;

        __self->_8StackInt_i6nbitem_ = 0;
        __self->_8StackInt_i4size_ = size;
        buf = (int *)calloc(size, sizeof(int));
        __self->_8StackInt_Pi4data_ = buf;
}

void _8StackInt_v4init_P8StackInt(StackInt *__self)
{
        int *buf;

        __self->_8StackInt_i6nbitem_ = 0;
        __self->_8StackInt_i4size_ = 2048;
        buf = (int *)calloc(2048, sizeof(int));
        __self->_8StackInt_Pi4data_ = buf;
}

void _8StackInt_v5clean_P8StackInt(StackInt *__self)
{
        int *buf;

        buf = __self->_8StackInt_Pi4data_;
        free(buf);
}
```

```
int _8StackInt_i6nbitem_P8StackInt(StackInt *__self)
{
	int n;

	n = __self->_8StackInt_i6nbitem_;
	return (n);
}

void _8StackInt_v4push_P8StackInti(StackInt *__self, int i)
{
	int pos;
	int *buf;

	pos = __self->_8StackInt_i6nbitem_;
	buf = __self->_8StackInt_Pi4data_;
	buf[pos++] = i;
	__self->_8StackInt_i6nbitem_ = pos;
}

int _8StackInt_i3pop_P8StackInt(StackInt *__self)
	int *buf;
	int pos;
	int r;

	pos = __self->_8StackInt_i6nbitem_;
	buf = __self->_8StackInt_Pi4data_;
	r = buf[--pos];
	__self->_8StackInt_i6nbitem_ = pos;
	return (r);
}

StackInt *_8StackInt_alloc()
{
	return (malloc(sizeof(StackInt)));
}

void _8StackInt_delete(StackInt *__self)
{
	_8StackInt_v5clean_P8StackInt(__self);
	free(__self);
}

StackInt *_8StackInt_new_v4init_P8StackInti(int size)
{
	StackInt *__elem;
```

```
        __elem = _8StackInt_alloc();
        _8StackInt_v4init_P8StackInti(__elem, size);
        return (__elem);
}

StackInt *_8StackInt_new_v4init_P8StackInt()
{
        StackInt *__elem;

        __elem = _8StackInt_alloc();
        _8StackInt_v4init_P8StackInt(__elem);
        return (__elem);
}
```

7.4 Traduction partie 3

Un exemple de déclaration d'une classe héritant d'une autre :

```
@class A
{
        @member         {
                int value;
                void init();
                @virtual void print();
        }
}

    @class B : A
{
        @member int value;
        @member void init();
        @virtual void print();
}
```

Sa traduction en C :

```
/* CLASS A */
typedef struct _1A_         A;

/* A MEMBER FUNCTIONS */
void        _1A_v4init_P1A_(A *);
void        _1A_v5print_P1A_(A *);

/* A DEFAULT FUNCTIONS */
const char        *_1A_Pc17name_of_interface_v_(void);
A        *_1A_P1A5alloc_v_(void);
```

```
A       *_1A_P1A3new_v_(void);
void      _1A_v6delete_P1A_(A *);
void      _1A_v5clean_P1A_(A *);

/* A VTABLE */
struct _1A_vtable_
{
      const char      *(*_Pc17name_of_interface_v_)(void);
      void      (*_v5clean_P1A_)(A *);
      void      (*_v5print_P1A_)(A *);
};

/* A CLASS */
struct      __attribute__((packed)) _1A_
{
      struct _1A_vtable_      *__vtable;
      int      _1A_i5value_;
};
/*!CLASS A */

/* CLASS B */
typedef struct _1B_      B;

/* B MEMBER FUNCTIONS */
void      _1B_v4init_P1B_(B *);
void      _1B_v5print_P1B_(B *);

/* B DEFAULT FUNCTIONS */
const char      *_1B_Pc17name_of_interface_v_(void);
B       *_1B_P1B5alloc_v_(void);
B       *_1B_P1B3new_v_(void);
void      _1B_v6delete_P1B_(B *);
void      _1B_v5clean_P1B_(B *);

/* B VTABLE */
struct      _1B_vtable_
{
const char      *(*_Pc17name_of_interface_v_)(void);
void      (*_v5clean_P1B_)(B *);
void      (*_v5print_P1B_)(B *);
};

/* B CLASS */
struct      __attribute__((packed)) _1B_
{
      struct _1B_vtable_      *__vtable;
```

```
        int             _1A_i5value_;
        int             _1B_i5value_;
};
/*!CLASS B */
```

Un exemple d'implémentation d'une classe héritant d'une autre :

```
@implementation A
{
        @member void init()
        {
                [super init];
                [self.value] = 42;
        }

        @virtual void print()
        {
                int value;
                char *name;

                name = [self.name_of_interface];
                value = [self.value];
                print("%d %s", value, name);
        }
}

@implementation B
{
        @member void init()
        {
                [super init];
                [self.value] = 124;
        }

        @virtual void print()
        {
                int value;
                char *name;

                name = [self.name_of_interface];
                value = [self.value];
                printf("%d %s", value, name);
        }
}
```

Sa traduction en C :

```
/* A IMPLEMENTATION */
```

```
/* A INIT */
struct _1A_vtable_ _1A_vtable_ = {_1A_Pc17name_of_interface_v_, _1A_v5clean_P1A_,
_1A_v5print_P1A_};

/* A MEMBER FUNCTIONS */
void _1A_v4init_P1A_(A *__self)
{
        __self->_1A_i5value_ = 42;
        __self->__vtable = &_1A_vtable_;
}

void _1A_v5print_P1A_(A *__self)
{
        int value;
        const char *name = __self->__vtable->_Pc17name_of_interface_v_();

        value = __self->_1A_i5value_;
        printf("%d %s", value, name);
}

/* A DEFAULT FUNCTIONS */
const char *_1A_Pc17name_of_interface_v_(void)
{
        return ("A");
}

A *_1A_P1A5alloc_v_(void)
{
        return (malloc(sizeof(A)));
}

A *_1A_P1A3new_v_(void)
{
        A *res;

        res = _1A_P1A5alloc_v_();
        _1A_v4init_P1A_(res);
        return (res);
}

void _1A_v6delete_P1A_(A *__self)
{
        _1A_v5clean_P1A_(__self);
        free(__self);
}
```

```
void _1A_v5clean_P1A_(A *__self)
{
        _1A_v5clean_P1A_((A *)__self);
}
/*!A IMPLEMENTATION */

/* B IMPLEMENTATION */
/* B INIT */
struct _1B_vtable_ _1B_vtable_ = {_1B_Pc17name_of_interface_v_, _1B_v5clean_P1B_,
_1B_v5print_P1B_};

/* B MEMBER FUNCTIONS */
void _1B_v4init_P1B_(B *__self)
{
        _1A_v4init_P1A_((A *)__self);
        __self->_1B_i5value_ = 124;
        __self->__vtable = &_1B_vtable_;
}

void _1B_v5print_P1B_(B *__self)
{
        int value;
        const char *name = __self->__vtable->_Pc17name_of_interface_v_();

        value = __self->_1B_i5value_;
        printf("%d %s", value, name);
}

/* B DEFAULT FUNCTIONS */
const char *_1B_Pc17name_of_interface_v_(void)
{
        return ("B");
}

B *_1B_P1B5alloc_v_(void)
{
        return (malloc(sizeof(B)));
}

B *_1B_P1B3new_v_(void)
{
        B *res;

        res = _1B_P1B5alloc_v_();
        _1B_v4init_P1B_(res);
        return (res);
```

```
}

void _1B_v6delete_P1B_(B *__self)
{
        _1B_v5clean_P1B_(__self);
        free(__self);
}

void _1B_v5clean_P1B_(B *__self)
{
}
/*!B IMPLEMENTATION */
```