Kooc

DOCUMENTATION

* Développeurs :
* Caner Candan
* Florent Hochwelker
* Majdi Toumi
* Morgan Armand

Sommaire

* Introduction
* Concept
* Critères d’évaluation du langage
  1. Inclusion
     + **Mot clé : @import**
  2. Scope de déclaration de symbole
     + **Mots clés : @module, @implementation et [ … ]**
  3. Proclamation des initialisations globales
  4. Compatibilité avec symbole C
  5. Préservation sémantique du C
  6. Règle de décoration des symboles
  7. Paquetage
     + **Mot clé : @package**
  8. Type abstrait
     + **Mots clés : @class et @member**
  9. Héritage et polymorphisme
     + **Mot clé : @class name : parent, @virtual**
  10. Gestion d’erreurs
* Problèmes rencontrés
* Conclusion

Introduction

Ceci est notre documentation de conception officielle de l’ensemble du projet Kooc.

Nous allons parler de notions gérées par le langage et expliquer les grandes lignes du fonctionnement de celles-ci.

Les différents exemples sont illustres par du code source ainsi que des portions d’arbre expliquant ces derniers. Il existe aussi une documentation technique dans le répertoire “/html” a la racine du projet Kooc ; Cette dernière référence toutes les fonctions et règles utilisées par le compilateur.

Concept

1). Un arbre unique

Nous avons choisi de générer un seul arbre avec tout le code figurant dans le fichier Kooc pour que les dépendances de la CNORM comme les références vers les types ne soient pas corrompu.

Nous conservons en parallèle un sous-arbre nomme « kooc » qui va stocker les références vers les membres de nos modules, implémentations, packages et classes afin que les appels Kooc puissent faire appel au bon élément et tester si les éléments existent réellement lors de la compilation.

Exemple :

* project
  + block
    - [0], [1], [2]
  + Kooc
    - modules
      * [“myModule1”], [“myModule2”]
    - implementations
      * [“myModule1”], [“myModule2”]
    - Packages
      * [“pkg1”]
    - Classes
      * [“myClasse1”]

2). Découpage

Chaque déclaration de l’arbre CNORM contient un champ appelé \_\_FILE\_\_ qui référence le fichier de sortie dans lequel la déclaration doit être écrite, ainsi il nous est possible de placer tous les fichiers dans un même arbre sans les séparer.

3). Surcharge de la CNORM

Pour implémenter quelques notions du Kooc, il était nécessaire de surcharger quelques règles de la CNORM (entre autres : translation\_unit, declarator et primary\_expression).

3). Compilation

Pour utiliser notre préprocesseur Kooc, il faut avant tout exporter des variables d’environnement. Les variables PATH, KOOC\_PATH et CNORM\_PATH doivent être paramétrés.

> export PATH $PATH ":"`pwd`

> export KOOC\_PATH `pwd`

> export CNORM\_PATH `pwd`/cnorm\_x\_x

Ensuite, il suffit d’utiliser le binaire kooc ou kooc.bat selon l’environemment.

Critères d’évaluation du langage

1). Inclusion :

L’inclusion est faite par le mot clé **@import.**

Implementation de l’inclusion :

@import “fichier\_include.kh“

Contrairement a une inclusion simple type C, le mot clé @import génère un fichier avec pour extension un .h et une protection contre l’inclusion multiple. Elle apporte une meilleure protection et gestion du code.

Voici son fonctionnement : lorsque le kooc rencontre cette écriture, il va analyser le fichier inclut « .kh » et générer un autre fichier « .h ».

Le fichier étant générer par le prettyPrinter, il est nécessaire de remplir l’arbre de la CNORM en prenant soin de remplir le champ \_\_FILE\_\_ de chaque déclaration.

Conception de l’arbre Kooc :

Voici comment est généré notre arbre pour ce qui est de l’inclusion de fichier :

<project>

…

<kooc>

<modules>

<\_\_ARRAY\_ENTRY \_\_KEY="test">

<\_\_ARRAY\_ENTRY \_\_KEY="0" \_\_REFERENCE="project.block[&quot;0&quot;]" />

<\_\_ARRAY\_ENTRY \_\_KEY="1" \_\_REFERENCE="project.block[&quot;1&quot;]" />

<nameH \_\_VALUE="km\_test.h" />

<nameC \_\_VALUE="km\_test.c" />

</\_\_ARRAY\_ENTRY>

<\_\_ARRAY\_ENTRY \_\_KEY="VVV">

<\_\_ARRAY\_ENTRY \_\_KEY="0" \_\_REFERENCE="project.block[&quot;2&quot;]" />

<\_\_ARRAY\_ENTRY \_\_KEY="1" \_\_REFERENCE="project.block[&quot;3&quot;]" />

<nameH \_\_VALUE="km\_vvv.h" />

<nameC \_\_VALUE="km\_vvv.c" />

</\_\_ARRAY\_ENTRY>

</modules>

…

</kooc>

</project>

Dans notre nœud **this.kooc.module**, pour chaque module, nous avons un champ **nameH** qui contient le nom du fichier header généré à partir d’un module qui sera donc inclus lors de l’utilisation du mot clé **@import**.

2). Scope de déclaration de symbole :

Le scope de déclaration de symbole est fait par les mots clés **@module** et la définition par **@implementation.**

La syntaxe **[Module function :f1 :f2 :f2]** nous permet d’accéder aux éléments de ces modules.

Implémentation du scope de déclaration de symbole :

@module myModule

{

int a;

char \*b;

float c;

void func(int nbr);

float func();

}

@implementation myModule

{

void func(int nbr)

{

[myModule.a] = nbr;

}

float func()

{

return ([myModule.c]);

}

}

int main(int argc, char \*\*argv)

{

[A func] ;

[A func :42] ;

}

Conception de l’arbre Kooc :

Voici comment est généré notre arbre pour ce qui est du scope de déclaration de symbole :

<project>

…

<kooc>

<modules>

<\_\_ARRAY\_ENTRY \_\_KEY="myTest">

<\_\_ARRAY\_ENTRY \_\_KEY="0" \_\_REFERENCE="project.block[&quot;0&quot;]" />

…

<nameH \_\_VALUE="km\_mytest.h" />

<nameC \_\_VALUE="km\_mytest.c" />

</\_\_ARRAY\_ENTRY>

</modules>

…

<implementations>

<\_\_ARRAY\_ENTRY \_\_KEY="myTest">

<\_\_ARRAY\_ENTRY \_\_KEY="0" \_\_REFERENCE="project.block[&quot;6&quot;]" />

…

</\_\_ARRAY\_ENTRY>

</implementations>

</kooc>

</project>

Les différents scope de déclarations sont définies dans les nœuds **this.kooc.modules** et **this.kooc.implementation.** Pour ce qui est des modules et implémentations, nous avons choisi de généré un couple .c et .h. Grace a cela, nous avons donc une vrai modularité et une clarté du code généré.

4). Proclamation des initialisations globales :

Le code de notre **Kooc** dispose d’un moyen unique de déclaration de variable globale au sein d’un module logiciel et contrôle l’accès interne ou externe de cette ressource.

5). Compatibilité avec symbole C :

Le code de notre **Kooc** a une compatibilité totale avec le C, c'est-à-dire que tous ce qui n’est pas dans les nouveaux mots clés ne sont pas déclarés.

6). Préservation sémantique du C

Le code de notre **Kooc**  permet de différencier simplement du code C et du code Kooc. Ainsi, notre **Kooc**  est vraiment un préprocesseur intelligent.

7). Règle de décoration des symboles :

Le code de notre **Kooc** utilise une technique d’encodage qui ne perturbe pas le compilateur et tous les identifiants générés sont valables.

Implémentation de la décoration des symboles :

Nous avons implémenté les règles de décorations de symboles utilisés en C++.

Conception de l’arbre Kooc :

Voici comment est généré notre arbre pour ce qui est du mangling :

<project>

<block>

<\_\_ARRAY\_ENTRY \_\_KEY="0">

<\_\_LINE\_\_ \_\_VALUE="13" />

…

<name \_\_VALUE="\_ZN6myTest1aEi" />

<oname \_\_VALUE="a" />

…

</\_\_ARRAY\_ENTRY>

…

</block>

…

</project>

Dans ce cas précis, nous avons une variable nomme **a** déclarer dans le module **myTest**. Le nœud name contient le nouveau nom après décoration de symbole. Le nœud oname (old name) contient l’ancien nom des variables, prototypes, …

8). Paquetage :

Le paquetage des données est faite par le mot clé **@package**, il permet de regrouper un certain nombre de module, ainsi il est possible de d’organiser nos modules de manières efficace.

Implementation du paquetage :

Le code de notre Kooc permet deux syntaxes pour créer et utiliser des packages en Kooc.

1er syntaxe :

@package A // accède au package A

@package A.B // accède au package B de A

@package A.B.C // accède au package C de B

2eme syntaxe :

@package A

{

@package B

{

@package C

{

}

}

}

La seconde syntaxe a le mérite d’être plus claire quant a la localisation de chaque package dans le code.

Conception de l’arbre Kooc :

Voici comment est généré notre arbre pour ce qui est du paquetage :

<project>

<block>

<\_\_ARRAY\_ENTRY \_\_KEY="10">

<\_\_LINE\_\_ \_\_VALUE="14" />

…

<name \_\_VALUE="\_ZN6level14mod15printEv" />

<oname \_\_VALUE="print" />

<\_\_FILE\_\_ \_\_VALUE="tests/km\_level1\_mod1.h" />

</\_\_ARRAY\_ENTRY>

…

</block>

<kooc>

<modules>

<\_\_ARRAY\_ENTRY \_\_KEY="level1.mod1">

<\_\_ARRAY\_ENTRY \_\_KEY="0" \_\_REFERENCE="project.block[&quot;10&quot;]" />

<nameH \_\_VALUE="km\_level1\_mod1.h" />

<nameC \_\_VALUE="km\_level1\_mod1.c" />

</\_\_ARRAY\_ENTRY>

…

</modules>

</project>

Dans ce cas précis, nous avons un prototype d’une fonction **print** déclarer dans le module **mod1** du package **level**, …

9). Type abstrait :

Le code de notre **Kooc**  permet la notion de « type abstrait » et nous permet donc de donner à un type de donne le comportement d’un type primitif.  C’est le mot clé **@class** qui permet le support de ces types manquants au modules **Kooc** .

Implémentation du typage abstrait :

@class Stack

{

@member int function(int num) ;  
 @member int function(int n1, int n2);

int koala;

}

Conception de l’arbre Kooc :

Voici comment est généré notre arbre pour ce qui est des classes :

<project>

<block>

<\_\_ARRAY\_ENTRY \_\_KEY="0">

<\_\_FILE\_\_ \_\_VALUE="tests/kc\_StackInt.h" />

<name \_\_VALUE="StackInt" />

<type \_\_VALUE="\_\_TYPE\_\_" />

<etype \_\_VALUE="declaration" />

…

</\_\_ARRAY\_ENTRY>

…

</block>

<kooc>

<classes>

<\_\_ARRAY\_ENTRY \_\_KEY="StackInt">

…

<\_\_ARRAY\_ENTRY \_\_KEY="2" \_\_REFERENCE="project.block[&quot;6&quot;]" />

<\_\_ARRAY\_ENTRY \_\_KEY="3" \_\_REFERENCE="project.block[&quot;5&quot;]" />

…

<nameH \_\_VALUE="kc\_StackInt.h" />

<nameC \_\_VALUE="kc\_StackInt.c" />

</\_\_ARRAY\_ENTRY>

</classes>

</kooc>

</project>

Dans ce cas précis, nous avons une classe StackInt, notre nœud \_\_FILE\_\_ contient donc le nom du fichier de sortie de cette classe portant comme préfixe kc (kooc class) puis le nom de la classe (.h et .c). Le fichier source généré contient les fonctions membres mangles ainsi que la structure représentant la classe.

10).Héritage et polymorphisme:

Le code de notre **Kooc**  permet la réutilisation du code ce qui permet faire évoluer notre conception de classe. De plus, il permet d’hériter des fonctions membres a l’aide du mot clé **@virtual** d’un autre type, ainsi ils peuvent réutiliser ou ré implémenter

Implémentation de l’héritage et du polymorphisme:

Pour ce qui est de l’implementation de l’héritage, nous utilisons le support d’agrégation qui permet d’avoir simplement notre structure parent dans la structure qui hérite de cette dernière.

Le polymorphisme est implémenté tel qu’il est présente dans le cours ; Pour chaque classe, nous créons donc deux structures, l’une décrivant l’interface (variables membres), l’autre décrivant la vtable. Pour plus de détails, se reporter à l’exemple de la ferme ou au test\_poly.

Conception de l’arbre Kooc :

Voici comment est généré notre arbre pour ce qui est de l’héritage et d polymorphisme:

<project>

…

<\_\_ARRAY\_ENTRY \_\_KEY="11">

<type \_\_VALUE="\_\_VARIABLE\_\_" />

<etype \_\_VALUE="declaration" />

<name \_\_VALUE="\_kc\_vtable\_B" />

<ctype>

<pctx \_\_REFERENCE="project.block" />

<type \_\_VALUE="\_\_COMPOSED\_\_" />

<identifier \_\_VALUE="\_kc\_implementation\_B" />

<specifier \_\_VALUE="struct" />

</ctype>

…

<\_\_FILE\_\_ \_\_VALUE="tests/kc\_B.c" />

</\_\_ARRAY\_ENTRY>

…

<kooc>

<modules />

<classes>

<\_\_ARRAY\_ENTRY \_\_KEY="A">

…

<\_\_ARRAY\_ENTRY \_\_KEY="4" \_\_REFERENCE="project.block[&quot;7&quot;]" />

<\_\_ARRAY\_ENTRY \_\_KEY="5" \_\_REFERENCE="project.block[&quot;8&quot;]" />

<nameH \_\_VALUE="kc\_A.h" />

<nameC \_\_VALUE="kc\_A.c" />

<parent />

</\_\_ARRAY\_ENTRY>

<\_\_ARRAY\_ENTRY \_\_KEY="B">

…

<\_\_ARRAY\_ENTRY \_\_KEY="4" \_\_REFERENCE="project.block[&quot;16&quot;]" />

<\_\_ARRAY\_ENTRY \_\_KEY="5" \_\_REFERENCE="project.block[&quot;17&quot;]" />

<nameH \_\_VALUE="kc\_B.h" />

<nameC \_\_VALUE="kc\_B.c" />

<parent \_\_VALUE="A" />

</\_\_ARRAY\_ENTRY> <block>

…

</classes>

</kooc>

</project>

Dans ce cas précis, nous avons une classe nomme **B** qui hérite de la classe **A**, de plus, une fonction virtuelle est redéfini dans chaque classe.

11).Gestions d’erreurs:

Le code de notre **Kooc**  permet une gestion d’erreur bien précise, par exemple la gestion des types  pour le cast explicite ; C'est-à-dire que lorsqu’un cast est effectue de manière implicite, nous avons une erreur signalant une ambigüité. Nous gérons le mauvais nombre de paramètres, la gestion d’erreur «basiques », les indications de lignes et colonnes, …

Problèmes rencontrés

La CNORM est loin d’être parfaite, du moins, quand on l’utilise pour le kooc, imaginons le code suivant :

(1) typedef int Entier ;

@module A

{

(2) typedef int Entier;

(3) Entier a ;

}

A l’aide de la règle déclaration, la CNORM va dans (1), ajouter une référence dans le nœud « types » de l’arbre avec comme key « Entier ».

Puis arrive à la ligne (2), la CNORM va voir que dans le nœud « types » il existe déjà un élément appelé « Entier », donc ERROR !

Il faut trouver un moyen pour différencier les deux types, pour cela on essaye en manglant les types dans les modules, l’ERROR existe encore, PK ? Car nous pouvons mangle que après que la CNORM est ajoute dans l’arbre le type mais comme il existe déjà…ERROR !

De plus si on regarde la ligne (3), si on arrive a mangle le type de la ligne (2), on a un autre problème a regle, celui de l’appel a ce type, « Entier » lui sera lie au type définie a la ligne (1) et non celui a la ligne (2) .

Solution :

De nouveaux mots-clés peuvent régler ce problème, un petit exemple pour illustrer cela :

(1) typedef int Entier ;

@module A

{

(2) @typedef int Entier;

(3) @typeof Entier a ;

}

Si on se trouve dans des blocks Kooc, on utilise des mots-clés commencent par « @ » pour différencier explicitement la déclaration C du code Kooc et ainsi de mangle avance de stocker l’élément dans l’arbre, puis a la ligne (3) en utilisant « @typeof » on refait appel au type mais cette moi nous faisons appel au type mangle a la ligne (2).

Conclusion

Jamais nous ne remettons en cause ou altérons un comportement d'une structure du langage C.

Nous rajoutons seulement de nouvelles structures pour traiter de nouveaux concepts.

The dreamTeam :p