Kooc: Conception

Colliot Vincent, Diego Moran, Frederic Lavrut

Contents

Mangling

What

Mangling/Name Mangling is a technique used to avoid many problem on unique names resolution.
Mangling will modify the name following some pattern in order to avoid name collisions.

Why

We needed to solve unique name resolution to be able to create a C oriented object.
When you create a Class in cpp you'r going to set variable and methode in it, if we want to simulate this we need to tell in c that a function is only callable by the struct.
The solution for this is to change the function name to className_functionName.
The function can in fact still be called by the user. This is why we obfuscate the name.

Code Equivalence

Listing 1: various examples in Kooc

```
int NAME;
char NAME;
bool NAME(int, int);

@namespace NAME {
  @namespace EMAN {
    int NAME;
  }
}
```

Listing 2: various examples in C

```
int ___8variable__3int__4NAME__;
char ___8variable__4char__4NAME__;
bool ___8function__4NAME__3int__3int__4bool__;

int ___4NAME__4EMAN__8variable__3int__4NAME__;
```

Implementation

___[{nb}NAMESPACE__]*[{nb}CLASS__]?{nb}SYMTYPE__{nb}SYMNAME[__[{nb}TYPARG__]*{nb}RETTYPE]?__
{nb} = len(word)
When we mangle we add ___ as separator at the begining and __ between informations, before all informations we tell the lenght of the information.
We mangle with all the namespaces, the class if present. Symtype makes the difference between a function or a variable. Symname is the name given to this symbol.
Then, if it's a function, we also mangle with every argument's type, and the return type.
This is implemented by a Class who have for purpose to change the variable name like the previous declaration.

Import

What

To support the use of import in our kooc we use an @Import Statement for safe inclusion.

Why

We needed to be able to include header in our languague so we used @Import as statement to simulate the #include.

Code Equivalence

Listing 3: Import demo in Kooc code

```
@import(include)
```

Listing 4: Import translation in C code

```
#ifndef ___include
# define ___include
# include "include.h"
#endif
```

Implementation

To implement this we created a custom node (Imp).
We pars the statement to change it to a correct C #include.
This node is not a part of our work tree and only implement a new meta to C methods.

Namespace

What

The purpose of a namespace is to define sub context for declaration of variables and functions.

Why

To be able to use our languague as an object oriented languague we needed to add a Namespace declaration.

Code Equivalence

Listing 5: Namespace demo in Kooc header

```
@namespace(NAMESPACE) {
  int VARIABLE = 5;
  bool FUNCTION();
}
```

Listing 6: Namespace translation in C header

```
extern int MANGLED_VARIABLE;
bool MANGLED_FUNCTION();
```

Listing 7: Namespace demo in Kooc code

```
@definition(NAMESPACE) {
  bool FUNCTION() {
    return true;
  }
}

int main() {
  return NAMESPACE@VARIABLE;
}
```

Listing 8: Namespace translation in C code

```
int MANGLED_VARIABLE = 5;
bool MANGLED_FUNCTION() {
  return true;
}

int main() {
  return MANGLED_VARIABLE;
}
```

Implementation

The definition of a namespace (called 'module' in the subject) is @namespace(NAME){}, and its implementation is @definition(NAME){}
We implemented this by detecting Namespace declaration (@definition) and added a node to do all mangling and operation of transformation for a Namespace declaration.

Class

What

A class is an object containing variables and methods. Inheritance and polymorphism is a possibility of classes.

Why

To be able to use our languague as an object oriented languague we needed to add a Class declaration.

Code Equivalence

Listing 9: Class demo in Kooc header

```
@namespace(n) {
    @class a() {
        @constructor()       @callable();
        @destructor()       @callable(virtual);

        int a               @property();
    };
    @class ap() {
        @constructor()       @callable();
        @destructor()       @callable(virtual);

        int a               @property();
    };
    @class b(n@a, n@ap) {
        @constructor()       @callable();
        @destructor()       @callable(override);

        void    methode()   @callable(virtual);

        int b               @property();
    };
    @class c(n@b) {
        @constructor()       @callable();
        @destructor()       @callable(override);

        void    methode()   @callable(override);
    };
};
```

Listing 10: Class demo in Kooc code

```
#include <stdio.h>
@import(class)

@definition(n) {
    @implementation(a) {
        @destructor() {
        }
        @constructor() {

        }
    };
    @implementation(ap) {
        @destructor() {

        }
        @constructor() {

        }
    };
    @implementation(b) {
        @constructor() {

        }

        @destructor() {

        }

        void    methode() {
            printf("i’m a b\n");
        }
    };
    @implementation(c) {
        @constructor() {

        }

        @destructor() {

        }

        void    methode() {
            printf("i’m a c\n");
        }
    };
}

int main() {
    n@c c_object = @new(n@c)();
    n@a a_object =(n@a)c_object;

    @get(n@a@a)(c_object);
    @call(n@b@methode)(c_object);
}
```

Implementation

The definition of a class is @class(NAME){}, and its implementation is @implementation(NAME){}
Methodes can be private or public and can be set to be virtual or override.
Attribute can be set private or public.
Like the previous implementation of Namespace we created a node to do the specific mangling and transformation of the code.