Kind Of Objective C

KOOC 2016

Membres de l'équipe

victorien.lecouviour-tuffet@epitech.eu
gabriel.cadet@epitech.eu
hubert.dufourt@epitech.eu
quentin.choupin@epitech.eu
paul.montague@epitech.eu
yannick.beluche@epitech.eu

Table des matières

Table des matières

1.Introduction

1.1 Sujet

Le projet KOOC a pour objectif de fournir une surcouche au C visant à lui donner des fonctionnalités de la programmation objet sous forme d'un exécutable Python capable de pré-processer le projet éponyme.

Les librairies Python utilisées pour faire ce projet sont :

- La librairie Pyrser
- La librairie CNORM

2. Exigences du sujet

2.1 Exigences techniques

Le programme doit pouvoir faire remonter les erreurs, de manière explicite, concernant la surcouche KOOC empêchant la traduction en C.

Le fichier de sortie ne doit pas avoir besoin d'être remanié pour être compilable, et doit être compréhensible par un humain.

2.2 Exigences du langage

Une syntaxe objet univoque, on ne doit pas confondre un appel KOOC et un appel C.

Le code C doit être directement intégrable dans le langage KOOC.

3. Description des fonctionnalités

La surcouche a pour objectif de rajouter certaines fonctionnalités au langage C :

3.1 Modularité du langage

Le langage permet de regrouper facilement en blocs autosuffisants des morceaux de code.

3.2 Abstraction des données

- Utiliser son type utilisateur comme type natif/primitif
- Permet de regrouper un certain nombre de classes selon des caractéristiques communes
- Traitement commun applicable à des entités/concepts varié(e)s
- Réduire la duplication de l'information dans un programme et fournir une interface plus ergonomique pour l'utilisateur.

3.3 Système de scopes

- Un scope définit « l'étendue au sein de laquelle l'identifiant est lié. » Wikipedia, C'est-à-dire permettre de définir un bloc ayant une unité logique dans son fonctionnement

3.3.1 Namespaces

- Possibilité de créer un scope nommé

3.3.2 Système de classes

- Possibilité de créer un scope nommé permettant de définir un type.

3.3.3 Système d'héritage

  - Permet de préciser la définition d'un type en sous-types ayant des caractéristiques communes, sans duplication de code, tout en permettant d'abstraire son utilisation.

3.4 Autogestion de l'inclusion multiple

- Assure l'unicité des symboles en cas de multiples inclusions d'un même fichier.

3.5 Polymorphisme

- Permet d'utiliser un même nom pour définir des éléments différents au sein d'un même scope.

4. Elements du programme

cf : annexe A « Diagramme FAST »

4.1 Production de l'AST

La production de l'AST se fait pendant la phase de parsing. Elle inclue la décoration des symboles et de l'arbre pour une navigation ultérieure rapide.

4.2 Typage

Le typage consiste à résoudre un type et donc le symbole correspondant à partir d'une expression '[ ]'. On applique ensuite la passe de transformation de la classe 'Bracket'.

4.3 Transformation de l'AST en version CNORM

Cela consiste en l'application des passes de transformation sur chacun des nœuds KOOC, sachant qu'un nœud KOOC correspond à une classe possédant une méthode représentant la passe.

4.4 Transformation de l'AST en code C

Cela correspond à l'application de la méthode 'to_c' de la classe 'Node' du CNORM sur la racine de l'AST.

5. Description des unités logicielles

5.1 Parsing du fichier KOOC – Parser.py

- Fonction : Module servant à produire l'AST. Toutes ses fonctions seront des «@meta.hook». Il va se charger de créer tous les nœuds (KOOC et CNorm).
- Entrée : Le fichier *.kc/*.kh
- Sortie : AST décoré

Possède aussi le rôle de création des références vers les tous éléments KOOC, sauf pour les nœuds 'Bracket', pour lesquels on référence le nœud parent englobant toutes les expressions '[ ]' de l'instruction.

5.2 Import des fichiers externes (*.kh) – Preproc.py

- Fonction : Parse le fichier spécifié en entrée s'il n'a pas déjà été importé. Sauvegarde des types (KOOC et C) définis. Crée un nouveau type de nœud KOOC servant à représenter l'import (classe 'Import').
- Entrée : Nom du fichier.
- Sortie : Sous arbre de l'AST produit par parser.py dont la racine est le nœud 'Import'.

5.3 Décorateur de symboles – Mangler.py

Cf. Annexe B « Règles de décoration des symboles »

- Fonction : Décoration des symboles KOOC contenus dans les modules et classes définis par @module et @class
- Entrée : Nœud généré par le CNORM représentant une variable/fonction
- Sortie : Symbole décoré

5.4 Typage des expressions – Typer.py

- Fonction : Typage des expressions KOOC '[ ]'
- Entrée : Nœud parent aux expressions '[ ]' de l'instruction, les englobant toutes
- Sortie : Même nœud parent qu'en entrée, sauf qu'a la place des nœuds KOOC, on aura applique une passe de transformation sur chacun des nœuds 'Bracket', une fois typés.

5.4.1 Description de l'algorithme de typage

L'algorithme est contenu dans le module et ne correspond pas à la fonction appelée de l'extérieur. C'est cette fonction qui en interne appel l'algorithme sur chacun des nœuds. Fonction correspondant à celle décrit ci-dessus.

L'algorithme de typage prend en paramètre un nœud 'Bracket' représentant une expression '[ ]', ainsi qu'une liste de types **possibles**.

Pour chaque nœud, il tente de le typer grâce à la liste de types passée en paramètre. On descend récursivement dans les branches en passant en paramètre la liste de types possibles pour l'élément en cours. En cas de changement dans cette liste : soit tous les types ont été éliminés et le nœud est transformé en nœud CNorm, on retourne alors un tuple contenant ce nœud et un booléen à 'False', soit le nœud est laissé tel quel et on retourne un tuple contenant le nœud en cours et un booléen a 'True'.

Lors de la remontée, en cas de changement dans la liste de types, il faut mettre à jour celle du nœud parent, et recommencer sur le nœud en cours (récursion terminale).

En cas de conflit : On cherche d'abord à inférer le type grâce à une fonction prenant deux listes de types. Si le typage est considéré comme impossible, une erreur remonte.

5.5 Traduction de la surcouche objet

5.5.1 Représentation des modules – Module.py

- Fonction : Sert à définir un nouveau type de nœud KOOC représentant le contenu d'un module KOOC dans l'AST décoré.
- Entrée : Nom du module et l'AST représentant son contenu
- Sortie : L'AST CNORM correspondant

5.5.2 Représentation des classes – Class.py

Hérite de la classe module

- Fonction : Sert à définir un nouveau type de nœud KOOC représentant le contenu d'une classe KOOC dans l'AST décoré.
- Entrée : Nom de la classe et l'AST représentant son contenu
- Sortie : L'AST CNORM correspondant

5.5.3 Représentation de la syntaxe crochets – Bracket.py

- Fonction : Sert à définir un nouveau type de nœud KOOC représentant les crochets, dans l'AST décoré
- Entrée : Sous arbre représentant le contenu des crochets
- Sortie : L'AST CNORM correspondant

5.6 Transformation de l'AST – ASTTransform.py

- Fonction : Appeler les passes de transformation de chacun des nœuds KOOC.
- Entrée : Nœud contenu dans la racine de l'AST contenant les références stockées vers les nœuds KOOC ('Import', 'Module', 'Class'), et nœud 'Bracket' pour lequel le passe de transformation sera appelé juste après son typage.
- Sortie : AST CNORM

6. Annexes

A) Diagramme FAST

B) Règles de décoration des symboles

séparateur: '_'
commence par deux séparateurs : "__"
suivi du type du symbole: "kvar", "kfct", "kmeth"
suivi du nom du module / class
suivi du nom de la variable / fonction / methode
suivi d'une liste de types

Représentation d'un type:

Identifiants:

v : void
i : entier
f : flottant
F_[...]_E : fonction (uniquement pour les ptr sur fct, donc forcément préfixé du symbole pointeur)
U_[...]_E : type utilisateur

signedness:

s : signed
u : unsigned

Pour les types scalaires on donne la size et la signedness, exemples:

unsigned int => i4u
double => f8s

Pour les pointeurs on préfixe le type par autant de 'p' que de niveaux, exemple:

char *var[0] => ppi1s

Pour les ptr sur fct:

pF_retType[_argType]+_E

Pour les user types:

U[_attrType]+_E

Représentation d'une fonction:

typeDeRetour[_typeD1Arg]+

Représentation d'une méthode:

typeDeRetour[*_typeD1Arg]**
même chose que pour une fonction sauf qu'on ne précise pas le type du self dans la liste d'arg

/!\ si le self est le seul arg on ne met pas 'v': et oui, ce n'est pas parce qu'on ne précise pas le self qu'on n'a pas d'argument, donc on ne met rien -> fin du symbole
typiquement "__kmeth_Alek_leVentSouffle_v_v" correspondrait à :

```
void __kmeth_Alek_leVentSouffle_v_v(Alek *self, void);
// gcc output => error: 'void' must be the only parameter
```

Ce qui ne compile pas.

6.3 Exemples de traduction

Trad.kh :

```
@class Trad
{
  int lol = 2;

  @member
    {
      int var;
      double var;

      void print();
      void clean();
    }

  void init(Trad *self, int, double);
  @member void init(double);

  int truc();
}
```

Trad.h :

```
#ifndef TRAD_H_
# define TRAD_H_

extern int  __kvar_Trad_lol_i4s;

typedef struct  __kclass_Trad Trad;
struct      __kclass_Trad
{
  int       __kvar_Trad_var_i4s;
  double    __kvar_Trad_var_f8s;
};

Trad *      __kfct_Trad_alloc_pUTrad_E_v(void);
void        __kmeth_Trad_init_v_i4s_f8s(Trad *self, int, double);
void        __kmeth_Trad_init_v_f8s(Trad *self, double);
Trad *      __kfct_Trad_new_pUTrad_E_i4s_f8s(int, double);
Trad *      __kfct_Trad_new_pUTrad_E_f8s(double);
void        __kmeth_Trad_print_v(Trad *self);
void        __kmeth_Trad_clean_v(Trad *self);
void        __kmeth_Trad_delete_v(Trad *self);
int     __kfct_Trad_truc_i4s_v(void);

#endif /* !TRAD_H_ */
```

Trad.kc :

```
@import "Trad.kh"

#include <stdio.h>

@implementation Trad
{
  @member
    {
      void init(int i, double f)
      {
    [self.var] = i;
    [self.var] = f;
      }

      void init(double f)
      {
    [self.var] = f;
      }

      void print()
      {
    printf("int = %d | double = %f\n", @!(int)[self.var], @!(double)[self.var]);
      }

      void clean()
      {
    [self.var] = 0;
    [self.var] = .0f;
      }
    }

    int truc() {}
}
```

Trad.c :

```
2
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include "Trad.h"
6
7   int __kvar_Trad_lol_i4s = 2;
8
9   __kclass_Trad * __kfct_Trad_alloc_pU__kclass_Trad_E_v(void)
10  {
11    return (malloc(sizeof(__kclass_Trad)));
12  }
13
14  void    __kmeth_Trad_init_v_i4s_f8s(__kclass_Trad *self, int i, double f)
15  {
16    self->__kvar_Trad_var_i4s = i;
17    self->__kvar_Trad_var_f8s = f;
18  }
19
20  void    __kmeth_Trad_init_v_f8s(__kclass_Trad *self, double f)
21  {
22    self->__kvar_Trad_var_f8s = f;
23  }
24
25  __kclass_Trad * __kfct_Trad_new_pU__kclass_Trad_E_i4s_f8s(int i, double f)
26  {
27    __kclass_Trad *self = __kfct_Trad_alloc_pU__kclass_Trad_E_v();
28
29    if (self)
30      __kmeth_Trad_init_v_i4s_f8s(self, i, f);
31    return (self);
32  }
33
34  __kclass_Trad * __kfct_Trad_new_pU__kclass_Trad_E_f8s(double f)
35  {
36    __kclass_Trad *self = __kfct_Trad_alloc_pU__kclass_Trad_E_v();
37
38    if (self)
39      __kmeth_Trad_init_v_f8s(self, f);
40    return (self);
41  }
42
43  void    __kmeth_Trad_print_v_v(__kclass_Trad *self)
44  {
45    printf("int = %d | double = %f\n", self->__kvar_Trad_var_i4s, self->__kvar_Trad_var_f8s);
46  }
47
48  void        __kmeth_Trad_clean_v(__kclass_Trad *self)
49  {
50    self->__kvar_Trad_var_i4s = 0;
51    self->__kvar_Trad_var_f8s = .0f;
52  }
53
54  void        __kmeth_Trad_delete_v(__kclass_Trad *self)
55  {
56    _kmeth_Trad_clean_v(self);
57    free(self);
58  }
59
60  int     __kfct_Trad_truc_i4s_v(void)
61  {
62  }
```

6.4 Librairies utilisées

Pyrser : http://pythonhosted.org/pyrser/

CNORM : http://pythonhosted.org/cnorm/