ENSEIRB-MATMECA

Compilation - rapport final

C + map reduce

Auteurs :
Alexis Chan
Antoine Durand

Responsables :
M. Denis Barthou
M. Marc Sergent

Introduction

Ce rapport sert de conclusion au projet de compilation. Il contient des détails sur les différentes fonctions implémentées durant ce projet. Nous ferons un état des lieux entre ce qui a été décrit dans le premier rapport et ce qui a été effectivement réalisé.

La description de nos choix d'implémentation sera faite ainsi que la gestion des différents éléments du compilateur.

Un retour sur les tests sera également fait, ainsi que le passage ou non de ceux-ci par le compilateur écrit. Enfin, nous terminerons par ce qui aurait pu être fait si nous avions eu plus de temps.

Table des matières

1 Résumé des fonctionnalités du compilateur **2**

1.1 Implémentation et gestion des éléments du compilateur 2

1.1.1 Types 2

1.1.2 Déclarations 3

1.1.3 Structures conditionnelles et boucles 4

1.1.4 Tableaux à n dimensions 4

1.1.5 Fonctions 4

1.1.6 Map/Reduce 5

1.1.7 Affectations 5

1.2 Améliorations du compilateur 5

2 Tests réalisés sur le compilateur **6**

3 Conclusion **7**

1 Résumé des fonctionnalités du compilateur

Compte tenu des contraintes qui nous ont été imposées, nous avons tenu à remplir au maximum les objectifs que nous nous sommes fixés dans le premier rendu. En prenant en compte les différentes échéances qui nous ont été imposées, nous avons dû freeze les fonctionnalités afin de nous restreindre aux phases critiques du projet.

- Ce qui marche
  - Tout le code va être placé dans les structures adaptées
  - Appel de fonctions
  - Casts simple
  - assignations de variables
  - Tableaux à 1-dimension
  - Définition de fonctions
  - Structures conditionnelles
  - Boucles
  - Utilisation de variable et de constantes
  - Déclarations dans les limites de la fonction LLVM *alloca* : Si une seule invocation de *alloca* suffit, alors la déclaration est valide, sinon ca ne fonctionne pas.
  - Détection des erreurs de syntaxe et ses coordonnées ligne/caractère
- Ce qui ne marche pas
  - Génération du code LLVM incomplet
  - Assignation sur des expressions
  - Gestion des erreurs incomplète,
  - Map et Reduce
  - Pas de parallélisme particulier

1.1 Implémentation et gestion des éléments du compilateur

1.1.1 Types

Nous gérons les mêmes types cités dans le premier rapport. Néanmoins, l'union présente pour *type_s* a été remplacé par une structure pour savoir quel champ est effectivement utilisé, ce qui n'était pas le cas avec une union. Seuls deux noms de variable ont été modifiés pour apporter une clarté supplémentaire et diminuer la confusion.

```
//in types.c
//
//here, conceptually we would need an union, because a type can be any of thoses three.
//  we set the non used fields to 0 or NULL
struct type_s {
  type_p  prim;
  type_t* tab; //NULL if type is not tab
  type_f* func; //NULL if type is not func
};
```

```
struct type_t { //struct that holds tab type
  type_s* elem;
  int size;
};

//we make no distinction between function pointers and functions. functions are variable
struct type_f { //struct that holds function type
  type_s* ret;
  type_s** params; //always NULL when no param
  int nb_param;
};
```

Du fait que le code LLVM est fortement typé, il a été plus pratique, voir nécéssaire, de créer une fonction qui effectue les conversions de types demandés par LLVM. Ceçi revient en pratique à insérer du code la ou un cast implicite est présent. De fait, même si cette fonctionnalité était au départ très basse sur la liste des priorité, les casts explicites on étés ajoutés, car seul le changement dans la gramaire était alors nécessaire Ces casts ne fonctionnent cependant que sur des types primaires (char, int, float).

1.1.2 Déclarations

Le type variable est le suivant :

```
struct var_s {
    char* s_id;  //key

    int addr_reg;
    int flags;
    type_s* type;

    UT_hash_handle hh; //for uthash
    UT_hash_handle hh_param; //it's not possible to have the same item in two maps, so u
};
```

Le champ s_id est le nom de la variable. Les flags ne sont pour l'instant qu'utiles pour savor si une variable est externe ou non. les deux champs hh et hh_param servent pour stocker ces variables dans les tables de hachage

Les variables sont stockées dans des tables de hachage implémentées par une bibliothèque externe : *libut*. Elles nous permettent de gérer la portée des variables. Il existe toujours une table de hachage pour chaque bloc de code. Les variables à portée globale et les fonctions sont donc dans la table initiale, celle qui correspond au code du programme entier. Pour chaque bloc d'instructions, une nouvelle table est crée et les variables initialisées sont alors stockées dans cette table-ci. Les tables de hachage sont donc imbriquées les unes dans les autres pour réduire la portée des variables.

Cette imbriquation se réalise simplement grâce à un lien chainé : chaque table possède un pointeur vers sa table parente

```
struct var_lmap { //lmap, as in linked_maps
    var_s* map;

    int depth;

    var_lmap* up;
};
```

Lors de la définition d'une fonction, les paramètres font eux aussi l'objet de la création d'un objet var_s*, cependant le bloc d'instruction n'est pas ouvert à ce moment, et la table de hachage n'existe donc pas, et il n'est donc pas possible d'ajouter ces paramètes à la table qui correspond à la fonction.

Pour pallier à ce problème, une table de hachage additionnelle pending_var est remplie des paramètres successifs, puis vidée dans celle du nouveau bloc lorsque celui-ci est ouvert.

De plus, lors de l'application de la règle de définition d'une fonction, le corp de celle-ci est déjà fini, et donc sa table de hachage et refermée. Pour garder trace des paramètres à ce moment la, une table des paramètres de la fonction actuellement en train d'être définie est crée et utilisée.

1.1.3 Structures conditionnelles et boucles

La gestion des structures conditionnelles et des boucles se fait dans les fichiers *semantics.c* et *semantics.h*. Elles fonctionnent comme on peut s'y attendre en C classique.

```
void selection_semantics(
        char** resultp,  expr_s* cond, char* arg1, char* arg2);
void iteration_semantics(
        char** resultp, expr_s* arg1, expr_s* arg2, expr_s* arg3, char* arg4);
void iteration_do_while_semantics(
        char** resultp, char* arg1, expr_s* arg2);
```

Ces actions sémantiques consistent principalement à écrire les blocs d'instructions entre les bons labels et rajouter les instruction de branchement.

1.1.4 Tableaux à n dimensions

Les tableaux à une seule dimension sont reconnus, la déclaration et la récupération d'éléments sont possible. Les tableaux à plusieurs dimensions sont reconnus mais n'ont pas de traitement particulier de la part du compilateur sur le code llvm produit.

1.1.5 Fonctions

Les fonctions sont considérées par le compilateur comme des variables comme les autres. Il n'y a pas de distinction entre les fonctions et le pointeurs de fonction. Ainsi, La gestion des fonctions a été difficile à mettre en place : les noms de chaque fonction sont stockés dans une table de hachage, l'invocation des fonctions est faisable sans erreurs.

1.1.6 Map/Reduce

Elles n'ont pas été réalisées.

1.1.7 Affectations

La structure actuelle du compilateur fait qu'il n'est pas possible d'obtenir une adresse de variable à partir d'une expression, nous avons uniquement un registre qui contient le résultat de l'expression (une valeur), ce qui fait que l'affectation via une expression ne fonctionne pas. Afin de pouvoir effectuer des affectations de base, la grammaire a été modifiée pour qu'il ne soit possible de faire des affections directement qu'a des variables. Le même problème se posant pour lors de l'utilisation d'index pour les tableaux, la même solution approximative a été utilisée.

1.2 Améliorations du compilateur

La fonctionnalité qui manque le plus a notre compilateur et une gestion plus avancée de l'assignation car elle empêche complètement l'écriture de certains codes. D'autres soucis sont apparus dans les autres fonctionnalités du projet nous empêchant d'implémenter l'assignement.

Un des problèmes principaux de ce compilateur est qu'il n'y a pas de gestion des erreurs : Il est toujours supposé que le code a été écrit correctement. Si ce n'est pas le cas, l'erreur pourra se manifester à plusieurs endroits :

— au plus tard, lors de l'exécution
— pendant la compilation du code llvm, typiquement si l'erreur n'aurait été qu'un warning qui n'empêche pas le compilateur de fonctionner. Cependant, certaines erreurs peuvent provoquer la génération d'un code LLVM invalide, voir syntaxiquement faux.
— au plus tôt, pendant la compilation. Du fait que les erreurs ne sont pas gérées, un code invalide peux conduire à un segfault. Par exemple, il est toujours supposé qu'une variable utilisée a déjà été déclarée. Si ce n'est pas le cas, l'utilisation d'un pointeur NULL provoque immédiatement un segfault. De manière générale, le code a été fait de façon a provoquer les segfault le plus rapidement possible, afin de simplifier le débuggage.

Il s'agit certainement d'une amélioration capitale pour rendre le compilateur utilisable, même s'il est toujours possible de détecter la position des erreurs avec gdb.

Avec du temps supplémentaire, les parties portant sur map/reduce ainsi qu'une allocation adaptée des tableaux multi-dimensionnels auraient été faisables. Le parallélisme ainsi que la vectorisation des éléments auraient pris plus de temps en comparaison.

2 Tests réalisés sur le compilateur

Comme prévu dans le premier rapport, de nombreux tests ont été réalisés. Nous avons séparés certains tests afin de permettre une détection des erreurs plus simple. Les tests dont le nom débutent par "wrong" ne doivent pas réussir.

```
Valtira@Licorys compilKrokodil$ ./runTests.sh
cooking the executable
gcc -Wall -g -std=gnu11 -D_GNU_SOURCE
-o krokodil grammar.c scanner.c semantics.c types.c data.c
cooking is over
Tests compilation yay
./tst/AffAvance.c
./runTests.sh: line 10: 26402 Segmentation fault: 11  ./src/krokodil
"$file" >> analysis.out
./tst/affBase.c
./tst/affFunc.c
./runTests.sh: line 10: 26404 Segmentation fault: 11  ./src/krokodil
"$file" >> analysis.out
./tst/affValTab.c
./tst/affValTab.c:10:13: syntax error
./tst/comparateurs.c
./tst/comparateurs_fl.c
./tst/funcInit.c
./tst/map.c
./runTests.sh: line 10: 26409 Segmentation fault: 11  ./src/krokodil
"$file" >> analysis.out
./tst/mapreduce.c
./runTests.sh: line 10: 26410 Segmentation fault: 11  ./src/krokodil
"$file" >> analysis.out
./tst/operationsFloat.c
./tst/operationsInt.c
./tst/operationsIntANDFloats.c
./tst/reduce.c
./runTests.sh: line 10: 26414 Segmentation fault: 11  ./src/krokodil "$file" >> analysis
./tst/syntaxMotCles.c
./tst/syntaxMotCles2.c
./tst/syntaxMotCles2.c:52:21: syntax error
./tst/tabInit.c
./tst/wrongFuncBloc.c
./tst/wrongMain.c
./tst/wrongOperations.c
./tst/wrongReturn.c
./tst/wrongTabInt.c
./tst/wrongVoid.c
```

Dûs aux fonctionnalités manquantes, certains de nos tests qui doivent réussir résultent en

un *seg fault*. Notre gestion des erreurs n'est pas complète, certains tests d'échec réussissent en conséquence.

3 Conclusion

Bien qu'il a été très difficile d'arriver à couvrir tout les aspect de ce projet dans les temps voulus, il nous a cependant permis d'aborder et de résoudre des problèmatiques clés dans le fonctionnement d'un langage de programmation, ce qui offre une compréhension plus profonde des concepts qui se cachent derrière les particularités de chaque langage. Notre plus grand regret étant de ne pas avoir pu disposer de plus de temps, car il est bien visible qu'avec un projet a plus long terme il aurait été possible d'arriver à un résultat dépassant une simple expérimentation.