Rapport de projet INF564 : Compilation

1 - Typing

C'est probablement la partie qui a demandé le plus de temps. Toutes les règles de la partie "2 - Typage statique" de l'énoncé du projet sont implémentées dans le fichier typing.ml. Si l'une des conditions de la règle n'est pas respectée, une erreur de typage est levée, en suivant la forme donnée dans l'énoncé et en précisant au maximum la raison de l'erreur ainsi que sa localisation précise.
Afin de faciliter la lecture du code, un commentaire indique à chaque fois la règle qui est implémentée (par exemple (* 2.2.6 *) pour la sixième règle de la partie 2.2 de l'énoncé, soit la règle de typage d'une affectation)

2 - Production du code RTL

Pour produire le code RTL à partir d'une fonction de Ttree, on stocke le graphe de flot de contrôle dans la variable globale *graph.* C'est à cette étape-là qu'on gomme la différence entre expression et stmt, en traduisant tout par une ou plusieurs instructions.

Optimisation des opérations

Pendant la production du code RTL, l'optimisation se fait essentiellement au niveau des opérations binaires. Pour optimiser Ebinop(op,e1,e2), je commence par traduire e1 et e2 en des instructions ertl, ce qui crée potentiellement des étiquettes inutiles dans le graphe en construction (qui ne seront jamais appelées). Si ces instructions sont des Econst(n,_,destl), je peux optimiser en faisant le calcul pendant la traduction.

3 - Production du code ERTL

Il s'agit de traduire instructions par instructions le code rtl en ertl, à l'exception du début et de la fin des fonctions, ainsi que des appels de fonctions. On en profite pour optimiser les appels quand ils sont terminaux.

Optimisation des appels terminaux

Afin d'optimiser les appels de fonctions lisiblement, j'ai séparé les étapes à effectuer en plusieurs sous-fonctions :

- Les fonctions *callee_saved* et *callee_restored* sont appelées en début et fin de fonctions pour sauvegarder et restaurer les registres callee-saved.
- La fonction *args_caller* met les arguments dans des registres réels avant l'appel de fonction, et appelle *push_pile* si il y a plus de six arguments.
- La fonction *args_callee* est appelée en début de fonction pour récupérer les arguments, et appelle pull_pile s'il y en a plus de six.
- *begin_fun* effectue le Ealloc_frame, appelle *args_caller* et *callee_saved.*
- *end_fun* conserve le résultat dans le registre approprié, appelle *callee_restored* et effectue le Edelete_frame et le Ereturn.
- Dans le cas d'un appel terminal, on utilise *end_fun_terminal* au lieu de *end_fun* : Cette fonction permet de fermer la fonction qui appelle avant de laisser la place à la fonction appelée : on restaure les registres callee_saved, on effectue Edelete_frame puis on passe la main à la fonction appelée.

4 - Étude de la liveness

On utilise la variable globale *graph_info* pour stocker la table associant à chaque étiquette les informations de l'instruction correspondante. On commence par compléter les champs succ, defs et uses, qui ne sont pas modifiables ensuite. Grâce aux successeurs on peut aussi remplir le champ pred. On initialise ins avec uses. Ensuite on va remplir les champs ins et outs en suivant l'algorithme de kildall.

5 - Construction et coloration du graphe d'interférence

Cette partie utilise l'algorithme simplifié proposé par l'énoncé du TD pour construire puis colorier le graphe d'interférence à partir des données de durée de vie (première moitié du fichier ltl.ml). Certaines parties furent un peu fastidieuses, en raison de difficultées à utiliser proprement des données non mutables (issues des modules Set et Map de OCaml).
De plus, il fut difficile de tester l'entière validité de l'algorithme en raison de sa complexité et des quelques divergences déjà existantes avec les énoncés des TDs précédents.

6 - Production de code LTL

La production de code LTL est implémentée dans la seconde moitié du fichier ltl.ml.

7 - Linéarisation

La linéarisation est implémentée dans le fichier lin.ml. Si celle-ci n'a pas posé de grande difficulté, les fichiers *.s contenant le code assembleur produit étaient au départ compilée en fichier qui n'étaient pas exécutables sans erreur : à leur exécution, l'erreur suivante était levée : "Symbol `putchar' causes overflow in R_X86_64_PC32 relocation". Une solution proposée sur http://stackoverflow.com/ consiste à ajouter l'option "-no-pie" à la compilation par gcc, ce qui a fonctionné pour nous.

8 - Utilisation du code

Le code est compilé avec la commande "make". La commande "./mini-c file.c" peut alors être utilisée pour produire le code assembleur correspondant à "file.c" dans le fichier "file.s". Ce dernier peut ensuite être compilé en un exécutable, par exemple avec la commande "gcc file.s -no-pie -o file".