1 Généralités

Fichiers : `main.c, printer.c, latex.c, u32stack.c`

Pour compiler le projet, un appel à `make` dans le répertoire principal suffit. Pour lancer quelques tests, `make test` ou `make ltest` (passage par `less`).

L'exécutable attend le code à analyser sur son entrée standard. Lors de son appel, les options suivantes peuvent être passées en paramètres :

- `--ast` pour afficher l'arbre de syntaxe abstraite ;
- `--latex` pour une sortie LaTeX de l'arbre de syntaxe abstraite ;
- `--asm` pour afficher le code intermédiaire.

La `u32stack` est une structure qui peut être manipulée comme une pile ou une liste d'entiers. Elle est utilisées dans plusieurs sections du code.

2 Syntaxe

Fichiers : `lexer.lex, parser.y, error.c, ast.c`

La syntaxe reconnu comprend :

- les types void, char et int, ainsi que les pointeurs ;
- la déclaration et l'appel de fonctions ;
- les opérateurs logiques unaire (`!`) et binaires (`&,|,^,&&,||,==,!=,<=,>,>=,>`) ;
- les opérateurs arithmétiques unaire (`-`), binaires (`+,-,*,/,%`) et trinaire (`_ ? : _`) ;
- les opérateurs sur les pointeurs unaires (`&,$,*`) et binaires(`+,-`) ;
- les structures conditionnelles (if, if ...else) ;
- les structures de boucle (while, do ...while, for) ;
- les abbréviations unaires (`++,--`) et binaires (`+=,-=,*=,/=,%=,&=,|=,^=`) ;
- les tableaux ($t[i] \equiv *(t + 4 * i)$).

Pour ne pas alourdir inutilement l'analyse syntaxique, nous avons différencié les opérateur de déréférencement à gauche et à droite. Autrement dit, `$` pour écrire (`$x = 5;`) et `*` pour lire (`x = *y;`).

3 Analyse statique

Fichiers : `hash.c, context.c, static.c`

L'analyse statique se fait en un parcours de l'arbre de syntaxe abstraite. Lors du parcours d'une expression, son type est retourné, pour les vérifications de typage. Au fur et à mesure de la rencontre de la déclaration de symboles (variables ou fonctions), une table des symboles est remplie, avec l'aide d'une fonction de hachage. Les différents identifiants associés à un symbole peuvent prêter à confusion :

- l'identifiant local (`localId`) est la valeur retournée par la table de hachage pour un nom donné, et peut en fait correspondre à plusieurs symboles différents dans le programme (avec le même nom) ;
- l'identifiant global (`id`) est un identifiant qui permet d'accéder au symbole sans tenir compte des scopes ;

- le registre virtuel représente un espace mémoire où la valeur du registre (ou le retour de la fonction) sera stockée ;
- le registre réel correspond au registre machine qui sera utilisé pour manipuler effectivement la variable (plusieurs registres virtuels peuvent correspondre à un même registre réel).

Lors de l'analyse statique, on marque également les variables qui apparaissent dans une expression de la forme `&x` comme devant être spillés, afin de pouvoir effectivement récupérer une adresse mémoire.

4 Code intermédiaire

Fichiers : `asm.c`

Le code intermédiaire est formé d'instructions prenant trois registres comme paramètres (sauf exceptions). Les instructions sont :

- `STOP` : fin du programme (affichage de la valeur de retour) ;
- `SET` : le deuxième registre est en fait une valeur immédiate (`r0 <- imm1`) ;
- `MOV` : copie un registre (`r0 <- r1`) ;
- `MRD` : lit un mot en mémoire (`r0 <- *r1`) ;
- `MWR` : écrit un mot en mémoire (`*r0 <- r1`) ;
- `RGA` : copie l'adresse mémoire d'un registre virtuel (forcément spillé) (`r0 <- &r1`) ;
- `NOT`, `AND`, `OR`, `XOR`, `LNOT`, `LAND`, `LOR`, `EQ`, `NEQ`, `LE`, `LT`, `GE`, `GT`, `ADD`, `SUB`, `MUL`, `DIV`, `MOD` : les opérateurs correspondants (`r0 <- r1 OP r2`) ;
- `JMP, JZ, JNZ` : branchement (`r0` est en fait un numéro de label, `r1` est la condition éventuelle)
- `CALL` : liste de registres contenant le label de la fonction, le registre dans lequel mettre la valeur de retour et la liste des arguments (met les arguments dans les registres ou la pile, puis la valeur de retur dans le registre approprié) ;
- `RET` : restaure les registres ;
- `LBL` : le premier registre est en fait le numéro de label, le deuxième indique s'il s'agit du début d'une fonction et le troisième, le cas échéant, est l'identifiant du symbole de cette fonction (sauvegarde les registres, prépare le spill, et met les arguments dans les registres des paramètres).

5 Allocation de registres

Fichiers : `set.c, flow.c, intgraph.c, regalloc.c`

L'allocation de registres se fait par une heuristique de coloration de graphe.

Pour chaque fonction (code intermédiaire entre deux labels de fonction), on effectue une analyse de vivacité. On considères quatre familles d'ensembles : def et use en `u32stack`, et in et out en `set` (`set.c`). On commence par remplir def et use avec les registres respectivement remplis et utilisés, pour chaque instruction. On cherche ensuite le point fixe des équations :

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] \backslash \text{def}[n])$$

Et

$$\text{out}[n] = \bigcup_{s \in \text{succ(n)}} \text{in}[s]$$

On construit ensuite le graphe d'interférence composé d'autant de sommets que de registres virtuels. Pour n donné, on place des arrêtes d'interférence entre les éléments de `def[n]` et `out[n]` (dans le cas du `MOV` l'arrête entre l'unique élément de `def[n]` et l'unique élément `use[n]` est remplacée par une arrête de préférence).

L'allocation de registres consiste à colorier le graphe avec les k registres réels. Par ordre de priorité, on :

- retire les sommets de degré inférieur à k (le registre correspondant est ajouté à la liste de ceux qu'on pourra facilement colorier) ;
- contracte une arrête de préférence dont le degré combiné est inférieur à k (on ajoute un des registres dans la liste pour le colorier comme l'autre) ;
- retire une arrête de préférence ;
- spill un sommet (on marque le registre virtuel comme devant être stocké en mémoire et on retire le sommet).

6 Code MIPS

Ficiers : `mips.c`

Lors de la génération de code MIPS, la différence entre les registres spillés et non spillés et réduite autant que possible, par l'intermédiaire de fonctions qui chargent ou sauvegardent ces premiers depuis ou dans des registres temporaires.

7 malloc et free

Fichiers : `malloc/*`

Les fonctions `malloc` et `free` sont enregistrées au début du programme dans le contexte et fonctionnent comme des fonctions classiques (elles ne font pas partie du langage lui-même). Leur code est ajouté au début de la sortie, depuis les sources MIPS du répertoire `malloc`. Ces sources ont été obtenues par cross-compilation du fichier `malloc/malloc.c`.

L'allocation dynamique de mémoire fonctionne en manipulant des blocs (une en-tête contient un pointeur vers le bloc suivant dans la liste de `malloc`, ainsi que sa taille).

`malloc` maintient une liste de blocs libres attribués par le système. Lors de la demande d'un bloc, elle cherche dans la liste un bloc de taille suffisante, le découpe si nécessaire, le retire de la liste et retourne l'espace mémoire de ce bloc (après l'en-tête) à l'utilisateur).

`free` remonte dans l'en-tête et rajoute le bloc dans la liste (en le fusionnant éventuellement avec des blocs contigüs).

Notre implémentation de `malloc` et `free` est très largement inspirée de celle proposée à titre d'exemple dans le K&R.