Compilatuer mini-C

Amine CHAABOUNI & Valentin OGIER

March 13, 2021

1 Implémentation du compilateur

Le projet consistait à passer d'un code parsé à un code assembleur en construisant les étapes de typage, de conversion RTL, ERTL et LTL, puis de linéariser le code. Quelques difficultés ont été rencontrées mais le débuggage s'est fait assez naturellement grâce aux différents messages d'erreurs qui nous guidaient.

On va ici préciser les choix techniques adoptés et les difficultés rencontrées au cours de ce projet:

1.1 Ocaml

L'apprentissage du langage Ocaml a constitué un premier obstacle pour Amine qui n'a jamais écrit ni même lu du Ocaml. Néanmoins, on a su dépasser cette difficulté grâce aux connaissances de Valentin et à la bonne documentation du langage.

1.2 La déclaration des variables

Au niveau du typage et de la conversion de Ttree vers Rtltree, on est amené à travailler avec des blocks, sous forme de liste de déclarations et de liste de "statements". Un statement peut être un block d'où une imbrication de block. Afin de garder en mémoire toutes les variables déclarées et leurs types (resp. registres associés), on peut créer une table de hachage qui associe à chaque variable son type (resp. registre associé). En Ocaml, il est facile de garder plusieurs bindings par clés en utilisant une Hashtbl. De plus, en terminant avec la variable (en quittant le block dans lequel elle se trouve), il suffit d'enlever le premier binding pour récupérer l'ancien (Hashtblt.remove table key).

Néanmoins, on ne garde pas en mémoire la portée (le scope) de la variable, c'est-à-dire, à quel block elle appartient. La solution choisie est de créer une pile de Hashtbl dans typing.ml (resp. rtl.ml). À chaque nouveau block, on ajoute au sommet de la pile une nouvelle table de hachage qui gardera les différentes variables déclarées. À la fin du traitement du block, il suffit de retirer le premier élément de cette pile pour se retrouver dans le block englobant et poursuivre le traitement de ce dernier.

1.3 Indexation et Spilling

Afin de rendre le code plus léger et lisible, on a d'abord rajouté des champs aux types "structure" et "field" dans le fichier "ttree.ml". Ces champs contiennent respectivement la taille de la structure et la position d'un champ dans la structure. On a décidé de stocker ces informations sous la forme d'un entier multiple de "word_size" pour faciliter leurs utilisations ensuite. Dans l'exemple Listing 1, a est le premier champ de la structure S, de position est $1 \times word\_size = 8$. Et la taille de la structure est $3 \times word\_size = 24$.

```
1    struct S{
2        int a;
3        int b;
4        int c;
5    };
```

Listing 1: strucuture

La même décision a été prise au niveau du spilling (lors de la colorisation) afin de rendre le code dans "ltl.ml" plus lisible.

1.4 Registres Caller-saved/Callee-saved

On a fait le choix de sauvegarder tous les registres Callee-saved au niveau du code ERTL. Une solution aurait été de garder en mémoire les registres utilisées par la fonction afin de ne sauvegarder que ceux-là.

Par ailleurs, on a fait le choix de ne pas garder l'implémentation de la sauvegarde des registres caller-saved. La raison principales étant que cette étape n'était pas mentionnée dans le sujet du TD. Par ailleurs, son implémentation conduit à un code assembleur avec beaucoup de "spilling". Ceci est peut-être dû à l'algorithme simplifié de coloration qu'on a choisi d'adopter.

1.5 Jump dans le code assembleurs

Le code produit présente des instructions "jmp" parfois inutiles, inaccessibles. On a préféré ne pas s'attarder sur cette partie afin de consacrer plus de temps sur les extensions.

2 Tests

Un nouveau test a été rajouté afin de valider la correction du compilateur. Plus précisément, ce test regarde la bonne conversion des expressions "Ttree.Ebinop" pour les opérations "Blt/Ble/Bgt/Bge". Le test est question est appelée "comp1.c". On a traiter les expressions Ebinop dans le but d'optimiser l'utilisation des registres et de profiter des instructions immédiates présentes dans notre boite à outils. Le code test "comp1.c" regarde les différentes manières possible de faire des comparaisons. Il y a 16 cas :

- comparaisons entre une variable et une valeur constante
- comparaisons entre une valeur constante et une variable
- comparaisons entre deux valeurs constantes
- comparaisons entre deux variables

Chacun des 4 cas précédents présente 4 sous-cas pour les différentes comparaisons : $<$, $<=$, $>$, $>=$.

3 Extensions

Dans ce projet, deux extensions ont été implémentées:

3.1 Optimisation de l'appel terminal des fonctions

On s'est restreint ici aux fonctions récursives pour ce qu'elles présentent comme avantage: le tableau d'activation étant exactement le même, il n'est pas nécessaire d'ajouter des instructions supplémentaires du type "delete frame". Supposons qu'on a la fonction présente dans Listing 2:

```
1    int f(int a, int b, int c, int d){
2        if(a<1) return 0;
3        return f(b, c, d, a-1);
4    }
```

Listing 2: Exemple d'appel terminal dans une fonction recursive

On peut prouver que cette fonction termine, mais on va plutôt s'attarder sur le choix technique adopté.

Au niveau de la conversion RTL → ERTL, après avoir mis les arguments dans les pseudo-registres utilisés pour l'appel de la fonction, il suffit d'ajouter des instructions "Embinop" et l'opération "Mmov" afin de déplacer les nouveaux pseudo-registres dans ceux précédemment utilisés par la fonction (Ceux associés aux paramètres de la fonction dans sa définition). Ensuite, il faut changer l'appel à la fonction par une instruction "Egoto" qui nous ramène au label d'entrée de la fonction, c'est-à-dire, après l'allocation du tableau d'activation et de la récupération des différents arguments passés à la fonction (que ce soit en paramètres dans des registres ou sur la stack).

3.2 Implémentation des initialisations des variables

Cette extension est beaucoup plus technique et demande plusieurs modifications à différents niveaux. Notamment, les fichiers suivants ont été modifiés :

- Parser.mly
- Ptree.mli
- Ttree.mli
- Typing.ml
- Rtl.ml

D'abord, au niveau du parser. À cause de conflit de parsing, il n'était pas facile de définir les bonnes règles de précédence afin d'accepter ces deux écritures:

```
int x;
```

```
int x = 0;
```

Ainsi, faute d'idée ingénieuse, le choix s'est tourné vers un changement important du Ptree. Le type block ne contient désormais qu'une liste d'instruction. Toute déclaration sera considérée comme un "statement", "Ptree.Sdecl".

Le désavantage de ce choix est qu'on ne peut plus traiter toutes les déclarations d'un seul coup puis traiter les autres instructions.

Néanmoins, un grand avantage se présente: on peut à présent déclarer des variables à n'importe quel niveau du block et non pas au début.

Ensuite, au niveau du typeur, on remplit la table de hachage correspondante au block avec toutes les déclarations qu'on rencontre afin de garder l'architecture générale de Ttree, puis on renvoie un "Ttree.Sskip".

Au niveau de Rtl, comme on a la liste de déclaration des variables, aucun problème se présente et on garde notre manière de travail.

Discutons maintenant les instructions d'initialisation de variable. Au niveau du parser, on associe un nouveau type de "statement", "Ptree.Sinit", pour les initialisations.

Au niveau du typeur, on rajoute les nouvelles déclarations de variables dans la table de hachage correspondante au block puis on renvoie un "Ttree.Sexpr" dont le champ "expr_node" correspond à une expression d'initialisation (nouvellement créée, appelée "Ttree.Einit_local").

Au niveau de Rtl, comme les variables ont déjà été créées et des pseudo-registres attribués, il est assez naturel de traiter l'expression "Ttree.Einit_local" comme une expression "Ttree.Eassign_local". La seule différence se trouve dans l'exemple ci-dessous:

```
1   int main(){
2       int x;
3       x = 65;
4       {
5           int a = x;
6           int x = 67;
7           putchar(a);
8           putchar(x);
9       }
10      return 0;
11  }
```

Listing 3: Ordre de déclarations

Le résultat du code devrait être 'CA'. Néanmoins, vu qu'en rentrant dans le block, on a déjà défini toutes les variables, dans la ligne 5, on va utiliser le second x, celui qui vaut 67.

Le choix qui a été pris est de supprimer la variable de la table de hachage du block quand on rencontre sa déclaration (une déclaration comprise dans l'expression "Ttree.init_local").

Comme on lit la fonction de la fin vers le début, on va lire la ligne 6 avant la ligne 5. En supprimant la variable x de la table de hachage du block 4-9, au moment de lire la ligne 5, la table ne contiendra plus x et on sera amenée à regarder dans la table de hachage correspondante au block englobant (1-11).

La difficulté reste néanmoins non réglée. Observons le code suivant, dans Listing 4:

```
1   int main(){
2       int a,b,c,d=1;
3   }
```

Listing 4: Limitation de l'extension

Dans ce cas, un vrai code C devrait déclarer les variables a, b, c et d et initialiser uniquement d à 1.

Dans notre implémentation, toutes les variables sont initialisées à 1. La difficulté se trouve au niveau du parser. En effet, si on arrive à définir une parsing adéquat pour

```
int a = 0, b, c, d = 1;
```

par exemple, il est possible de changer les statement Ptree.Sdecl et Ptree.Sinit en un nouveau statement englobant et de faire le tri au niveau du typeur ou de Rtl.

Cette extension est accompagnée de 4 tests qui l'illustrent:

- initialize1.c: vérifie la possibilité d'écrire une initialisation de variable, ainsi que la possibilité de mettre cette instruction au milieu du block et non pas au début, avec toutes les déclarations de variables, comme le mini-C de base.

- initialize2.c: vérifie la bonne portée des variables et met l'accent sur le principe présentée dans le code Listing 5.

```
1  int main(){
2      int x = 65;
3      { int x = x; putchar(x); }
4      return 0;
5  }
```

Listing 5: undefined behaviour

  Ce code n'a pas pour résultat 'A'. C'est une "undefined behaviour". Ainsi, la déclaration de la variable est faite avant même d'évaluer la partie droite de l'instruction. Une implémentation, proposée en commentaire consiste à associer la valeur du premier x lors de la déclaration du second x. Afin d'avoir cette propriété, il suffit de dé-commenter, dans le ficher rtl.ml, la ligne 108 et de commenter les lignes 122 à 125.

- initialize3.c: vérifie la bonne implémentation de l'initialisation sur des pointeurs de structure. Met aussi en avant la difficulté rencontrée concernant l'initialisation de toutes les variables.

- initialize4.c: vérifie la possibilité de commencer le block avec une initialisation de variable. Teste aussi la durée de vie de la variable (cf Listing 3).