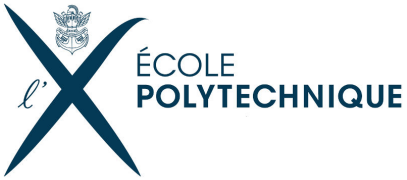
École Polytechnique
Master 1 - Institut Polytechnique de Paris




14 mars 2021

**INF564 - Projet de compilation**

# Compilateur pour un fragment de C

TABET GONZALEZ Salwa

# 1 Typage

Cette partie fut la plus facile pour moi car très méthodique. J'ai choisi de créer un autre type d'erreurs qui prend aussi en compte la localisation. J'ai mis toutes les structures et fonctions dans des Hashtbl pour créer des contextes. Ensuite, il y a deux fonctions qui traduisent les types de `Ptree` en types de `Ttree` et gèrent les compatibilités entre types. Ces fonctions furent utiles pour les grandes fonctions `typing_expr` et `typing_stmt` qui, comme leurs noms l'indiquent, gèrent le typage des expressions et des statements. Pour les blocs d'instructions, une fonction `typing_block` appelle la fonction de typage des statements, qui est elle-même appelée pour typer les fonctions. En effet, tout est fonction en `Ttree`, même les structures. Pour le programme principal, j'ai eu du mal avec les contextes: j'ai enfin compris qu'il fallait ajouter les structures et fonctions une première fois dans la table, puis une deuxième fois pour les mettre à jour elles et leurs dépendances.

# 2 Production de code

## 2.1 RTL

A partir du modèle de `typing.ml`, j'ai écrit des fonctions qui traduisent les opérateurs `Ttree` en opérateurs `Ops`. Ensuite, il y a une grande fonction `expr` qui, sous le modèle de `Typing.typing_expr`, prend en arguments des locals (comme le contexte, auparavant), un registre de destination et une étiquette de destination. Celle-ci traduit l'expression, en lui donnant au registre de destination la valeur de cette expression, et renvoyant l'étiquette où il fut transférer ensuite le contrôle. Cette étiquette est, de plus, ajoutée au graphe de flot de contrôle.

Pour éviter que la fonction soit trop lourde (contrairement aux fonctions dans `Typing`), j'ai décidé de créer des fonctions intermédiaires qui sont mutuellement récursives. Elles gèrent les conditions, les boucles, les opérateurs unaires et binaires, et finalement les appels de fonction. C'est à ce moment-là que j'ai décidé d'ajouter mon extension. Ensuite, on gère les statements de manière similaire. Enfin, dans le programme principal, tout comme dans la partie précédente nous devons ajouter chacune des fonctions deux fois dans la table des fonctions: une première fois pour s'assurer de leur existence, et une deuxième fois pour mettre à jour leurs dépendances, locals, et surtout, leur étiquette d'entrée. Cette partie m'a pris du temps pour le débogage, j'avais malencontreusement défini trois fois les locals, et de plus elles subsistaient en tant que variables globales de la fonction `program`. Cela faisait que même des fonctions très simples avaient 3 locals, et les registres des formals ne correspondaient à rien. J'avais envoyé un message au professeur, qui m'a gentiment aidée à déboguer ceci, ce qui m'a permis de ne pas rendre ce projet trop en retard.

## 2.2 ERTL et analyse de durée de vie

De même ici, on crée un graphe. Le sujet donnait les étapes pour écrire la fonction `handle_ecall` et `convert_fun`. Comme pour les prties précédentes, on commence par traduire les instructions `Rtltree` en instructions `Ertltree`. Pour moi, la partie la plus difficile de cela était la partie avec la pile. Je ne savais pas trop comment gérer la "longueur" de la pile, ni que faire de la position. Je pense ne pas avoir très bien compris cette partie du cours.

Ensuite, pour l'analyse de la durée de vie, je ne savais pas trop que faire avec `val liveness` et `type live_info`, mais on m'a expliqué que c'était en lien avec `Ertltree`. Tout était expliqué dans l'énoncé, encore une fois, et l'algorithme de Kildall était expliqué dans le cours (un peu succintement).

### 2.3 LTL

Tout d'abord, la construction du graphe d'interférence. Cette fois-ci le nom du fichier à créer n'était pas spécifié donc j'ai eu du mal à savoir ce qui était attendu. Le module est censé construire le graphe d'interférence, en faisant un premier parcours des instructions ERTL pour ajouter une arête de préference pour chaque instruction *mov x y*, puis un second parcours pour les interférences. Comme dans les autres modules, on commence par créer un graphe vide, on y ajoute les `intfs` vides que l'on mettra à jour au fur et à mesure. Ensuite, on ajoute la première étape grâce à la fonction auxiliaire `step_pref` et la deuxième grâce à la fonction auxiliaire `step_infs`.

On suit les instructions du TD pour le reste. J'ai passé beaucoup de temps ici aussi pour le débogage, je pense que mon code n'est pas des plus clairs car il y a beaucoup de branchements *if ... then ... else* et j'avais un problème au niveau d'un *else* qui était considéré comme le *else* d'un autre branchement que celui auquel il appartenait dans mon esprit. C'est la raison pour laquelle il subsiste beaucoup de mots-clés *begin ... end* entourant ces branchements. Cette petite erreur m'a beaucoup coûté en termes de temps, car même si j'avais fini de déboguer le RTL, le programme avait des exceptions venant de `Rtlinterp`. Sachant que ce fichier venait des archives du TD, je me demandais s'il fallait que j'ajoute quelque chose dedans, comme pour le fichier `Typing` où nous avons ajouté des des sizes et des fields. En réalité, le problème venait du LTL.

## 3 Extension

L'extension que j'ai choisi de faire est très simple. Il s'agit d'optimiser la production de code RTL en gérant les multiplications par des constantes différemment. J'ai choisi de seulement traiter les cas des constantes 0 et 1 : dans le premier cas, il suffit de ne pas convertir du tout l'expression autre que la constante 0, et directement générer une étiquette contenant la constante 0. Dans le deuxième cas, on traite directement l'autre expression, sans avoir besoin de créer de nouveaux registres ou labels pour la constante 1.

Le comportement de cette extension est illustré dans les fichiers de test `mult1.c` et `mult2.c`. Je les mets à la suite.

```
int main(){
    putchar(65+(0*2));
    putchar(65+(2*0));
    return 0;
}
```

Listing 1: mult1

```
int main() {
    putchar(65 + (1 * 2));
    putchar(65 + (2 * 1));
    return 0;
}
```

Listing 2: mult2

Leur RTL est comme suit :

```
=== RTL ================================================
#1 main()
  entry : L17
  exit  : L1
  locals:
  L17: mov $65 #12  --> L16
```

```
  L16: mov $0 #13  --> L15
  L15: add #13 #12  --> L14
  L14: mov #12 #11  --> L13
  L13: #1 <- call putchar(#11)  --> L12
  L12: mov $65 #7  --> L11
  L11: mov $0 #8  --> L10
  L10: add #8 #7  --> L9
  L9: mov #7 #6  --> L8
  L8: #1 <- call putchar(#6)  --> L7
  L7: mov $0 #1  --> L1
```

Listing 3: output mult1

```
=== RTL ================================================
#1 main()
  entry : L17
  exit  : L1
  locals:
  L17: mov $65 #12  --> L16
  L16: mov $2 #13  --> L15
  L15: add #13 #12  --> L14
  L14: mov #12 #11  --> L13
  L13: #1 <- call putchar(#11)  --> L12
  L12: mov $65 #7  --> L11
  L11: mov $2 #8  --> L10
  L10: add #8 #7  --> L9
  L9: mov #7 #6  --> L8
  L8: #1 <- call putchar(#6)  --> L7
  L7: mov $0 #1  --> L1
```

Listing 4: output mult1