Compiler Construction (level 2)

Étienne Renault

May 15, 2018

Contents

Review Partiel

Difficultes sur regle d'inference. (Revoir les regles d'inference sur les types)

ebreak(`f()`) , si `f() = readint()` alors c'est impossible de binder a la compilation.

Coercission aggrandissante, on stocke la variable dans un champs contenant plus de bits.
Coercission retrecissante, on stocke la variable dans un champs contenant moins de bit, Avec de la perte d'info.

1 Langage intermediaire

1.1 Pourquoi langage intermediaire ?

Reduire la quantite de code a ecrire, et les duplications de code.

Il faut trouver un point de convergence entre differents langage afin de produire un langage intermediaire.
Soit on le rapproche de la partie assembleur, soit on le rapproche des langages de plus haut niveau.

On a 4 etapes pour faire un compilateur :
- FrontEnd
- HIR (High Level Representation)
- LIR (Low Level Representation)
- Backend

Si on veut faire des optimisations, il faut les faire entre HIR et LIR, ainsi, cela profitera a tous les langages et tous les backends.

1.2 Comment savoir quand on est dans la representation intermediaire ?

A partir du moment ou le langage intermediaire est defini, on peut le savoir. Il faut s'assurer que notre traduction est correcte. Dans ce cas la le pretty printer est utile. Il y a differentes manieres de faire la representation intermediaire:
- fonctionelle
- imperative

Cela depend des langages a traduire.

1.3 GCC

GCC traduit differents langages vers GENERIC qui est un premier langage intermediaire.
Apres il simplifie avec gsimplifier.
Apres il ne va avoir que des variables constantes. Donc beaucoup plus de variable mais avec un temps de vie plus petit.

1.4 Stack ou Registre

Le langage intermediaire va s'executer en utilisant soit des registres, soit la stack.

1.5 Langage impur/ langage pur

Dans un langage impur, on va avoir des effets de bord. Donc le retour de la fonction appelee est cast en void. Dans un langage pur, cette fonction ne peut pas exister.

1.6 Pourquoi s'executer dans un registre ?

Pour avoir des performances optimales.
Cependant le nombre de registres est limite.
Dans l'ideal il faudrait utiliser uniquement les registres.
En cas de manque de registre on peut utiliser les caches L1 et L2.
Ou la RAM.
Ou le disque.

Cependant plus l'on descend dans cette liste , plus l'execution va etre lente.

Pour les registres, l'ordre des arguments que l'on va passer dans les registres ne doit pas etre neglige. C'est le fondateur du langage qui decide de cette organisation.

1.7 Actions sur la Pile

Rappel du cours d'assembleur:
On va sauvegarder fp (frame ptr) et sp (stack ptr) dans la stack pour pouvoir les restaurer a la fin des appels de fonction.
Regle a respecter : je n'ai pas le droit d'acceder a de la memoire en dehors de ma frame, je peux cependant l'aggrandir.

2 Frame n recursion

2.1 recursion

On doit garder un static link sur les frames precedentes.

ONE
TWO
THREE

TWO garde un static ptr sur ONE
et THREE garde un static ptr sur TWO.

ONE
TWO
TWO
TWO
TWO
TWO
THREE

chaque TWO garde un static ptr sur ONE
et THREE garde un static ptr sur le TWO le plus profond.

2.2 MIPS

En MIPS le static link est stocke au debut de la frame.

2.3 Assembleur

Pour la partie Hir Level, on va devoir sauvegarder les variables locales par frame.
La connaissance de la taille de la frame sera faite a l'execution.

2.4 Linearisation

On va chercher a applatir l'arbre construit par le compilateur.

2.5 CJUMP

```
_____________
if condition
then
do A
else
do B
_____________
```

$\Longleftrightarrow$

```
_____________
JumpEqual condition labelA
labelB (Optional)
...  (do B)
Jump labelendAB
labelA
...(do A)
labelendAB
_____________
```

3 Assembleur

3.1 Blocks

Il commece par un label.
Il finit par 1 jump ou cjump.
Il ne possede pas plus d'un label.

Pour organiser les blocks :
On recupere les blocks de base, et on les reorganise.
Ainsi la logique du programme est conservee.