ensiie
école nationale supérieure d'informatique pour l'industrie et l'entreprise

Compilation Avancée

(2018-2019)

TP2

Ajout d'une passe dans GCC

antoine.capra@atos.net
julien.jaeger@cea.fr
patrick.carribault@cea.fr

I Ajout d'une passe dans GCC

Le but de cette partie est d'implémenter une passe dans gcc à partir d'un plugin. Pour se faire vous vous baserez sur les éléments suivants : (i) la documentation des internals de gcc 9.1.0 disponible sur internet, (ii) les supports de cours et (iii) le code source des fichiers *header* fournis avec le système de plugin.

Lorsque GCC 9.1.0 a été compilé, nous avons utilisé le flag *–enable-plugin* , ce qui nous permet de creer nos propres plugins. Si vous voulez le faire sur un gcc antérieur installer à partir des package de LINUX, il vous faudra aussi installer le package `gcc-plugin-dev` correspondant à la version de gcc que vous voulez utiliser. **ATTENTION** : GCC évoluant entre les différentes versions, un plugin fonctionnant sur une version particulière de GCC peut ne pas être fonctionnel sur une autre version (antérieure ou postérieure).

Dans votre répertoire d'installation de gcc, trouvez le fichier *gcc-plugin.h* et parcourez-le. Il définit l'interface de base d'un plugin et devra être inclus dans votre plugin. Les autres fichiers importants sont *gimple.h*, *gimple.def*, *tree.h*, *tree.def*, *basic-block.h*, et *tree-pass.h*.

Q.1: En vous aidant de la documentation, écrivez un makefile et un premier plugin "vide" **plugin.cpp** qui s'activera lors du démarrage de GCC (pensez à mettre un affichage pour vérifier le bon fonctionnement du plugin). Vérifiez que ce plugin se lance bien avec gcc : **gcc -fplugin=mon_plugin.so ./test.c**.

Q.2: En vous aidant de la documentation et des exemples de passes vus dans le TD précédent, ajoutez une nouvelle passe :

1. Définissez une structure **struct register_pass_info new_pass**.
2. Remplissez-la afin de placer une passe après la passe *"cfg"*.
3. Initialisez une nouvelle passe :
   (a) Créez votre classe **my_pass** héritant de la classe **opt_pass**.
   (b) Implémentez les fonctions **gate** et **execute**.
4. Enregistrez-la avec un appel à **register_callback** (voir le code source de gcc *plugin.c* pour le code associé).

.
Vérifier que votre passe est fonctionnelle en mettant un **printf** dans les fonctions **gate** et **execute**. Pourquoi est-elle exécutée plusieurs fois dans le programme test fourni ?

Q.3: Affichez le nom de la fonction courante en utilisant `fndecl_name(cfun->decl)`. Allez voir cette fonction et l'objet **cfun** dans *function.h*.

Q.4: Quelles sont les autres fonctions permettant de récupérer le nom de la fonction courante ? Testez-les.

Q.5: Parcourez l'ensemble des blocs de base en affichant leur numéro d'index (voir l'objet `basic_block` dans le fichier *basic-block.h*) :

```
basic_block bb;
FOR_ALL_BB_FN(bb, cfun)
{
  ...
}
```

Q.6: Affichez, pour chaque basic block, leur numéro de ligne correspondant dans le code source avec la fonction `gimple_lineno` (voir le fichier *gimple.h*). Pourquoi cela génère-t-il un *segmentation fault* avec la macro `FOR_ALL_BB_FN` ? Quelle autre macro permet de parcourir les blocs de base ? Utilisez-le pour ne plus avoir de *segmentation fault*.

Q.7: Afin de mieux visualiser le graphe de flot de contrôle (*cfg*) des fonctions d'un programme, le fichier **plugin_TP2_7.cpp** vous propose d'écrire dans un fichier une sortie au format *graphviz*. Le logiciel *graphviz* permet de construire une image de graphe à partir d'une définition textuel des nœuds et des arcs. A vous de remplir le corps de la fonction **cfgviz_internal_dump** pour connstruire le cfg. Voici un exemple graphviz construisant deux nœuds et un arc entre ces deux nœuds.

```
Digraph G{
N0 [label="Node 0" shape=ellipse]
N1 [label="Node 1" shape=ellipse]
N0 -> N1 [color=red label=""]
}
```

Q.8: Maintenant, pour chaque bloc de base, parcourez l'ensemble des instructions (statements). Pour chaque statement correspondant à un appel de fonction, afficher le nom de la fonction appelée. Voici une liste de fonctions utiles dans ce contexte :

- la fonction *gsi_start_bb* permet de récupérer le premier statement GIMPLE d'un bloc ;
- la fonction *gsi_end_p* vérifie si le statement GIMPLE courant est le dernier du bloc ;
- la fonction *gsi_next* passe au prochain statement GIMPLE ;
- la fonction *gsi_stmt* récupère un statement à partir d'un statement GIMPLE ;
- la fonction *is_gimple_call* permet de savoir si le statement est un appel de fonction ;
- la fonction *gimple_call_fndecl* récupère l'arbre d'opérande (tree) d'un statement de type function call.

Voir aussi l'utilisation des macros `IDENTIFIER_POINTER` et `DECL_NAME`.