Compilation

TP 6 : Représentations intermédiaires – CFG et DAG

G. Iooss & C. Alias

La production directe de code Digmips est inefficace. On produit donc du code intermédiaire (ou *pseudo code*), qui sera ensuite traduit en code Digmips en plusieurs passes (optimisation, sélection, ordonnancement, allocation). Le but de ce TP est de définir, construire et optimiser ce code intermédiaire.

Exercice 1. *Pseudo-code, graphe de flot de contrôle*

Récupérez et décompactez le fichier `src_tp6.tar.gz`. Par rapport au TP précédent, les nouvelles classes sont les suivantes:

- **Register.h/.cc** remplace la gestion des temporaires. On suppose que l'on possède un nombre illimité de registres (numérotés à partir de 0) et on s'occupera de leur allocation (dans la pile ou sur les 4 registres non réservés) au moment de la génération du code.
- **BitVector.h/.cc** est juste une classe de manipulation des vecteurs de bits.
- **Code.h** contient une classe abstraite. Elle regroupe les méthodes communes aux différentes classes qui implémentent les représentations du programme source (`PseudoCode`, `BasicBlock`, ...).
- **PseudoCode.h** représente une pseudo-instruction. Remarquer l'implémentation des différentes méthodes abstraites de la classe `Code`. Le premier groupe d'instructions (ligne 15) correspondent aux instructions vues en cours. Les groupes suivants en implémentent des spécialisations sur les registres réservés (SP, ARP). Enfin, remarquer les pretty-constructors quasi-identiques à ceux qui générait le code Digmips au TP précédent.
- **Cfg.h** implémente un graphe de flot de contrôle. Chaque nœud de ce graphe contient un objet `Code` (donc, ici, soit un `PseudoCode`, soit un `BasicBlock`). `live-in[i]` contient les temporaires vivants *juste avant* d'exécuter le nœud `i`. `live-out[i]` contient les temporaires vivants *juste après* l'exécution du nœud `i`. Remarquer les pretty-constructeurs, qui ajoutent une instruction de pseudo-code à un CFG global (variable `cfg` définie dans parser.ypp, ligne 53).

Manip.

- Dans `main.cc`, construisez le CFG correspondant au code suivant. les temporaires se créent avec la fonction `new_register()`.

```
    r0 = 1
    r1 = 0
    r2 = 10
loop :
    cjump r1, GE, r2, end_loop
    r1 = r1 + r0
    r3 = 1
    r4 = r3 + r0
end_loop :
    r4 = r4 + r0
```

- **Afficher le CFG**. Pour cela, produire la description `dotty` avec l'instruction `cfg->print_dot(cout)`, puis compilez le résultat [1].
- **Calculer les durées de vie** en utilisant la méthode `do_liveness()` de `Cfg`. Afficher le CFG résultant.
- **Extraire les blocs de base** avec la méthode `extract_basic_blocks()` de `Cfg`. L'extraction se fait après le calcul de durées de vies. Afficher le CFG résultant.

[1] La commande est toujours `dot -Tps cfg.dot > cfg.ps`

Exercice 2. *DAGs*

Dag.h implémente un DAG entre pseudo-instructions. `node_reg[tmp]` donne le nœud qui enracine l'expression calculée dans le temporaire `temp`. `node_def[noeud]` donne la liste des temporaires qui sont supposés contenir le résultat calculé au nœud `nœud`.

Manip.

- Ouvrir `Dag.cc` et **étudiez le constructeur**. Il s'agit d'une variante de l'algorithme d'élimination de redondances vu en cours, mais sans fonction de hachage. Pour chaque instruction `r = r' op r"`, on regarde si `r'` et `r"` sont associés à des nœuds existants, et ces nœuds ont un père commun n qui réalise *op*. Si oui, $node\_reg[r] := n$. Des règles analogues existent pour les autres types d'opérateurs.
- Dans `main.cc`, **construisez le DAG du noeud 2** [2]. **Affichez le** avec `dag->print_dot(cout);`.

Exercice 3. *Production de code intermédiaire*

parser.ypp est modifié pour créer un nouveau CFG pour chaque fonction (ligne 812). Remarquer l'utilisation des pretty-constructors de `Cfg.h` dans les règles de traduction. Après avoir traduit la fonction, on calcule les durées de vies (ligne 827), puis on extrait les blocs de base (ligne 830) pour finalement produire un DAG pour chaque bloc de base (ligne 834 et suivantes).

Manip.

- Dans `main.cc`, commentez vos ajouts et décommentez le code d'appel au parseur. **Testez sur** `test/test.c`.

[2] Avec l'instruction `Dag* dag = new Dag((BasicBlock*)cfg_bb->node[2]);`, où `cfg_bb` est le CFG avec blocs de base construit dans l'exercice 1