GENERATION DU (VRAI) CODE

Blocs de base

- une suite (maximale) d'instructions qui est toujours exécutée sans transfert de contrôle

- Important pour la génération de code

- commence après une étiquette ou `goto` ...

- Le programme consiste en plusieurs blocs; les transferts possibles (apparemment) entre eux donnent un graphe.

Les problèmes

- Traduire une instruction (ou plusieurs) du code intermédiaire en une suite fixe d'instructions du code cible

- Décider quels pseudo-registres du code intermédiaire sont des vrais registres (et lesquels) [nombre fini disponibles]

- Impossibilité de savoir à l'avance les chemins vraiment parcourus par un programme sauf à l'intérieur d'un bloc de base

Génération directe de code pour les expressions à l'intérieur d'un bloc de base Première méthode basée sur un arbre

Pour décider où trouver un opérand et où calculer un résultat, il faut savoir
pour chaque identificateur, où est sa valeur actuelle (éventuellement registre/ mémoire
pour chaque registre, quel(s) identificateur(s) il stocke.
On suppose les registres ne contiennent pas de valeurs au début du bloc.

```
ADD $x, $y, $z
```

Choisir location (vrai registre) L pour le résultat

Choisir location Y qui va contenir y; générer `LD** $Y` (si nécessaire) pour y mettre y

Même chose avec Z et z

Générer `ADD* $L $Y $Z`; mettre à jour les informations sur L et x

mettre à jour les informations sur Y, Z, y et z

Choisir une location L (resp. Y, Z)

- Si possible, choisir un registre qui contient déjà x (resp. y ou z)

- si possible, choisir un registre non utilisé

- choisir un registre L qui contient la valeur d'un identificateur m; générer `ST** $L` (si besoin) (et mettre à jour les informations sur L et m

Le prétraitement:

calculer pour chaque identificateur s'il est *vivant* à la fin du bloc (sa valeur peut-elle être utilisée avant affectation?) et pour chaque valeur où est sa dernière utilisation dans le bloc.

Code à générer à la fin du bloc:

Stocker les valeurs des identificateurs vivants qui ne sont pas déjà dans leur adresse mémoire.

Application à de vrais langages

Comment traiter les variables du langage qui sont stockés sur la pile?

Du code intermédiaire on peut savoir que deux accès à une adresse de type `$pile, constante` peuvent être traités comme deux accès au même pseudo-registre (même constante) ou à deux registres (constantes différentes).

Pour incorporer aussi des variables de niveau différents (statiques ou adressées par l'adresseur), il faut s'en servir des informations qui ne sont pas évidentes du code intermédiaire (non-interférence entre blocs distincts).

De la même façon, tout accès à un élément d'un tableau ou accès via un pointeur doit être considéré comme un accès aléatoire faute d'utilisation d'information spécifique au langage source.

Une méthode plus raisonnable Basée sur le dag de dépendances

En général, on peut trouver des sous-expressions communes qui seraient calculées plusieurs fois par le code simple.

On évite ça si on construit le dag (graphe orienté acyclique) de toutes les valeurs calculées ou utilisées dans le bloc avec les dépendances indiquées par les arcs.

En générant le code selon un tri topologique de ce graphe, on ne calcule pas deux fois le résultat du même opérateur sur les mêmes opérands.

CHANGER L'ORDRE D'EXECUTION

Parce que plusieurs tris topologiques sont possibles, plusieurs ordres d'exécution sont possibles.

Il y a une différence dans le nombre de registres utilisés à un moment donné, et donc dans le nombre d'instructions `LD**` et `ST**` générées.

Sauf dans le cas où le graphe est un arbre il est trop difficile de calculer l'ordre optimal.

L'ORDRE OPTIMAL D'EXECUTION POUR UN ARBRE

- (optimal pour un nombre donné de registres disponibles et selon un critère qui peut dépendre de la taille du code et du temps d'exécution)

- Programmation dynamique

- Pour chaque sous-arbre (on commence aux feuilles), calculer la valeur du critère pour chaque nombre n de registres jusqu'au nombre total

- pour un ordre donné de calcul des sous-arbres calculer le temps et taille si on a n

registres disponibles pour le premier, $n-1$ pour le deuxième etc.

- ne marche plus quand un sous-arbre peut utiliser des valeurs calculés avant (et donc forcément stockées...)

ALLOCATION DE REGISTRES PAR COLORIATION DE GRAPHES

Quand un ordre de calcul est fixé, on peut construire un graphe dont les sommets sont les affectations aux identificateurs et les arêtes sont les incompatibilités (2 affectations sont incompatibles si à un moment de l'exécution leurs valeurs existent et seront utilisées plus tard)

On essaie de décider si ce graphe peut être coloré avec k couleurs où k est le nombre de registres disponibles

Un problème NP-complet si $k > 2$!

Mais il existe des heuristiques qui sont utiles

Effacer un sommet de degré inférieur à k

Si impossible, choisir un sommet de degré grand, le supprimer du graphe, résoudre le problème pour le graphe réduit
et ajouter des instructions pour stocker et relire la valeur.