Chapitre 9
Génération de code

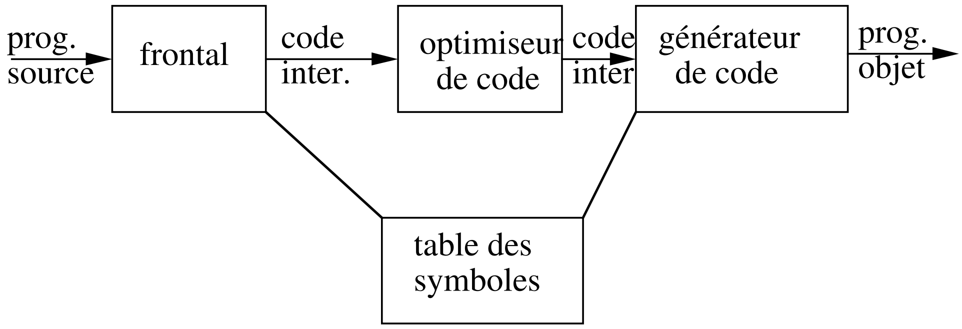

Figure 1: Position du générateur de code

Objectifs:

1. Engendrer du code correct et efficace
2. Utiliser au mieux les ressources de la machine cible.
3. Le générateur lui-même doit être efficace.

Le problème d'engendrer le code optimal est théoriquement indécidable.

Généralités

1. *input*: représentation intermédiaire: expressions postfixes ou code à trois adresses ou code de machine virtuelle ou arbres syntaxiques ...

   La partie frontale a déja effectué: analyse lexicale et syntaxique, traduction dans un langage intermédiaire, contrôle de types.

2. *output*: programme objet: code machine absolu ou relogeable ou assembleur (choix fait ici) ... Si on produit des *modules objet* qui sont du code machine relogeable, on peut compiler séparément. On utilise ! un éditeur de liens pour obtenir un *module chargeable*.

   Chaque module objet a une structure de la forme suivante:

identification
Table des points d'entrée Table des références externes
Instructions machine et constantes
Fin du module

3. *choix des instructions*: rend le code plus ou moins efficace. La qualité du résultat dépend

du jeu d'instructions de la machine cible et de la méthode de traduction (interaction architecture/compilation).

4. *choix des registres*: on doit choisir à la fois les variables rangées dans un registre et les registres utilisés pour une variable donnée.

   Le temps d'accés à un registre peut être 10 fois plus faible que l'accès à la mémoire centrale (par ex. 10ns/100ns)

La machine cible

Mémoire adressable par octet avec des mots de 4 oct! ets et n registres $R_0, R_1, \ldots, R_{n-1}$. Les instructions portent sur 2 adresses:

$$op\ \ source, destination$$

Les opérations comprennent:

`MOV` (déplacer *source* dans *destination*)

`ADD` (ajouter *source* à *destination*)

`SUB` (soustraire *source* de *destination*)

Les instructions peuvent être rangées sur plusieurs mots consécutifs. Modes d'adressage:

MODE	FORME	ADRESSE
absolu	M	M
registre	R	R
indexé	$c(R)$	$c + contenu(R)$
registre indirect	$*R$	$contenu(R)$
indirect indexé	$*c(R)$	$contenu(c + contenu(R))$

Ainsi:

$$\text{MOV } R_0, \text{ M}$$

garde le contenu du registre R_0 à l'adresse mémoire M .

$$\text{MOV } 4(R_0),\text{M}$$

enregistre la valeur $contenu(4 + contenu(R_0))$ à l'adresse M.

L'$*$ indique un mode d'adressage indirect. Ainsi:

$$\text{MOV } {*4}(R_0),\text{M}$$

range $contenu(contenu(4+contenu(R)))$ à l'adresse M .

Enfin

$$\text{MOV } \#1,R_0$$

charge la constante 1 dans le registre R_0.

Coût des instructions

On prend comme coût d'une instruction (en espace comme en temps) le nombre de mots qu'elle occupe. Ainsi

$$\text{MOV } R_0, R_1$$

recopie le registre R_0 dans R_1. Elle a coût 1.

$$\text{MOV } R_4, \text{ M}$$

copie R_4 dans `M`. Le coût est 2 car l'adresse `M` occupe le mot suivant l'instruction.

L'instruction

$$\text{ADD } \#1,\ R_3$$

ajoute la constante 1 au contenu de R_3. Le coût est 2 car la constante figure dans le mot suivant l'instruction.

L'instruction

$$\text{SUB } 4(R_0), *12(R_1)$$

range la valeur

$$\begin{aligned} contenu(contenu(12 &+ contenu(R_1))) \\ &- contenu(contenu(4 + R_0)) \end{aligned}$$

à l'adresse $*12(R_1!)$. Le coût est 3.

Voici par exemple différentes traductions de l'instruction de code à trois adresses `a := b+c`:

$$\text{MOV b, } R_0$$
$$\text{ADD c, } R_0$$
$$\text{MOV } R_0\text{, a}$$

a un coût 6 ainsi que l'autre traduction

$$\text{MOV b, a}$$
$$\text{ADD c, a}$$

Par contre, si les adresses de a, b, c sont déja dans les registres R_0, R_1, R_2, la traduction

$$\text{MOV } *R_1, *R_0$$
$$\text{ADD } *R_2, *R_0$$

a coût 2 seulement (idem avec les valeurs).

Gestion de la mémoire

Instructions du code à trois adresses:

`call`

`return`

`halt`

`action` (= corps de procedure)

Exemple: Input du générateur de code:

Code à trois adresses
`/* code pour c*/`
action_1
`call p`
$\texttt{action}_2$
`halt`
`/*code pour p*/`
$\texttt{action}_3$
`return`

Enreg. d'act. de c (64 o)

0:	`adresse de retour`
8:	`arr`
56:	`i`
60:	`j`

Enreg. d'act. de p (88 o)

0:	`adresse de retour`
4:	`buf`
84:	`n`

Implémentation

On utilise l'allocation dynamique. Les adresses relatives sont calculées par rapport à une base contenue dans un registre `SP` contenant un pointeur sur le début de l'enregistrement d'activation.

Code pour la procédure principale:

`MOV` $\#stackstart$ `SP`	/*initialise la pile*/
`ACTION`	/*code pour la procedure */
`HALT`	/*fin d'execution*/

traduction de **`call`**

`ADD` $\#caller.recordsize$,`SP`
`MOV` ! $\#here + 16$,$*$`SP` /*sauve l'adr. de retour*/
..
`GOTO` $callee.codearea$

traduction de **`return`**

dans l'appelé :
`GOTO` $*0($`SP`$)$ /*branchement à l'adr. de retour*/
dans l'appelant :
`SUB` $\#caller.recordsize$, `SP`

Exemple

On considère le code à trois adresses suivant (cf. quicksort):

CODE A TROIS ADRESSES

/*code pour `s`*/
`action`$_1$
`call q`
`action`$_2$
`halt`

/*code pour p*/ `action`$_3$ `return`

/*code pour q*/
`action`$_4$
`call p`
`action`$_5$
`call q`
`action`$_6$
`call q`
`return`

On suppose que le code pour `s,p` et `q` commence aux adresses 100, 200 et 300 et que la pile commence à 600.

```
100: MOV #600, SP /*initialise la pile*/
108: ACTION1
128: ADD #ssize, SP /*debut de la séquence d'appel*/
136: MOV 152, *SP /*empile l'adresse de retour*/
144: GOTO 300 /*call q*/
152: SUB #ssize, SP /*rétablit SP*/
160: ACTION2
180: HALT
     ...
     /*code pour p*/
200: ACTION3
220: GOTO *(SP) /*return*/
     ...
     /*code pour q*/
300: ACTION4
320: ADD #qsize, SP
328: MOV 344, *SP /*empile l'adresse de retour*/
336: GOTO 200 /*call p*/
344: SUB #qsize, SP
352: ACTION5
...
```

Calcul des adresses

Deux possibilités pour le code intermédiaire:

1. Les noms sont des pointeurs vers la table des symboles
2. Les suites d'accès sont calculées dans le code intermédi! aire.

Dans les deux cas il faut remplacer les noms par des adresses mémoire.

Exemple de traduction de l'instruction
`x := 0`
adresse relative de `x`=12.

1. Allocation statique: `x` est placé dans une zone commencant à l'adresse *static*. Le code à trois adresse pour `x := 0` peut être

   `static[12] := 0`

   Si *static* prend la valeur 100 le code engendré sera

   `MOV` #0, 112

2. Allocation dynamique. On peut utiliser un *display* qui est un pointeur sur la zone mémoire de `x` conservé dans un registre `R3`. La traduction en code à trois adresses peut être faite en

utilisant une variable pour contenir l'adresse de `x`:

$t_1 := 12 + \texttt{R3}$
$*t_1 := 0$

traduction en code:

`MOV` $\#0$, $12(\texttt{R3})$

Graphe du flot de données

On représente le code à trois adresses par un graphe (*flow graph*) utile pour calculer le code eng! endré. Les noeuds sont les blocs de base et les flèches représentent le flot d'exécution.

Un *bloc de base* est un fragment du code à trois adresses qui est toujours exécuté entierement en séquence.

Par exemple, le programme suivant:

```
begin
        prod := 0;
        i := 1;
        do begin
                prod := prod + a[i] * b[i]
                i := i+1
                end
                while i <= 20
end
```

est traduit par le code à trois adresses: On dit

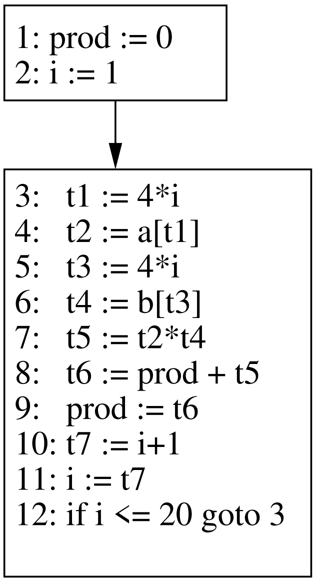

Figure 2: Graphe du flot

que `x := y+z` *définit* `x` et *utilise* `y` et `z`. Un nom dans un bloc de base est dit *vivant* s'il est utilisé plus tard (éventuellement dans un autre bloc).

Transformations sur les blocs de base

On peut appliquer à un bloc des transformations qui améliorent le code produit:

1. *Elimination des sous-expressions communes*: Par exemple, dans le bloc

   a:= b + c

   b := a - d

   c := b + c

   d := a! - d

   la deuxième et la quatrième instruction calculent la même expression b+c-d. On peut donc transformer en:

   a := b + c

   b := a - d

   c := b + c

   d := b

2. *Elimination de code inutile*: Si `x` est mort à l'endroit où apparait l'expression `x := y + z` on peut la supprimer.

3. *Renommage des variables temporaires.* On peut toujours transformer un bloc de façon que

chaque instruction utilisant une variable temporaire utilise une nouvelle variable.

4. *Echange d'instructions.* Si $x, y \neq t_1$ et $b, c \neq t_2$, on peut échanger les instructions:

$$t_1 := b + c$$

$$t_2 := x + y$$

5. *Transformations algébriques.* On peut remplacer

   x := y - y

   par

   x:= 0

Un algorithme de génération de code

On utilise:

1. Un descripteur de registres décrivant le contenu de chaque registre.
2. Un descripteur d'adresses donnant les emplacements où on peut trouver un nom! (peut être conservé dans la table des symboles).

On part d'un bloc de base et pour chacune des instructions x := y *op* z qui le composent on fait:

1. Utiliser la fonction *getreg* pour déterminer l'endroit `L` où doit être fait le calcul.
2. Consulter le descripteur d'adresse de `y` pour déterminer l'un des emplacements `y'` de `y` (de préférence un registre). Si `y` n'est pas déja dans `L`, engendrer `MOV y', L`.
3. Trouver un emplacement `z'` pour `z` et engendrer `OP z', L`.
4. Si `y` ou `z` n'ont plus d'utilisation les enlever des descripteurs de registres.

La fonction *getreg*

1. Si `y` est seul dans un registre et qu'il n'est pas utilisé ensuite retourner comme `L` ce registre.
2. sinon retourner un registre vide s'il y en a un.
3. sinon, si `x` est utilisé ensuite ou si *op* n! écéssite un regitre, choisir un registre `R`. Sauvegarder `R` par `MOV R,M` et retourner `R`.
4. sinon retourner `x`.

Exemple

On considère le bloc:

```
t := a - b
u := a - c
v := t + u
d := v + u
```

traduisant l'affectation `d := (a-b) + (a-c) + (a-c)` avec `d` vivant à la fin. L'algorithme se déroule ainsi:

Instructions	Code engendré	Descripteur de registres	Descripteur d'adresses
		reg. vides	
t := a − b	`MOV a,R0` `SUB b,R0`	`R0` a `t`	`t` dans `R0`
u := a − c	`MOV a,R1` `SUB c,R1`	`R0` a `t` `R1` a `u`	`t` dans `R0` `u` dans `R1`
v := t + u !	`ADD R1,R0`	`R0` a `v` `R1` a `u`	`u` dans `R1` `v` dans `R0`
d := v + u	`ADD R1,R0` `MOV R0,d`	`R0` a `d`	`d` dans `R0` `d` dans `R0` et mémoire