Cours de Compilation-Exercices

Génération de code

Master d'Informatique M1 2011–2012

21 novembre 2011

7 Production de code optimisé

7.1 Allocation de registres

Soit le programme écrit en langage intermédiaire :

```
y:=1
m:=x
e:=n
loop:
if e=0 goto fin:
t:=y mod 2
if t=0 goto pair:
y:= m*y
pair:
m:=m * m
e:=e div 2
goto loop:
fin:
print y
```

1. Donner en chaque point de programme, l'ensemble des variables vivantes (variables qui seront lues avant d'être affectées).
2. Construire le graphe d'interférence (graphe dont les sommets sont les variables et où il y a une arête entre x et y si elles sont simultanément vivantes).
3. Proposer un coloriage du graphe pour mettre les variables dans 4 registres.
4. Montrer qu'il n'est pas possible d'utiliser uniquement 3 registres.
5. On se propose donc de stocker la variable y en mémoire à l'adresse Y. Proposer une nouvelle version du programme, dans laquelle chaque accès à la variable y passe par une variable intermédiaire et l'accès à la mémoire. On utilisera uniquement des instructions

   `mem[Y]:=n, mem[Y]:=x, x:=mem[Y]`

   pour mettre en mémoire une constante, la valeur d'une variable ou récupérer dans une variable une valeur en mémoire.
6. Donner les variables vivantes pour ce nouveau programme.
7. Proposer une allocation des variables utilisant uniquement 3 registres.

7.2 Allocation de variables locales

(Examen janvier 04)

L'objectif de cet exercice est d'utiliser des techniques d'allocations de registre pour proposer un schéma de compilation qui se sert essentiellement de la pile pour les calculs intermédiaires mais en optimisant l'espace occupé.

On étudie un langage pour des expressions qui contiennent des définitions locales de variables. Ce langage est décrit par la grammaire suivante dans laquelle `cte` est un token correspondant à une valeur entière $(1, 2, \ldots)$, `ident` est un token correspondant à un identificateur qui peut représenter une variable locale ou bien une fonction prédéfinie et finalement `let` et `in` sont des mots clés et `=`, `(`, `,` et `)` sont des symboles. E est un non-terminal permettant de dériver les expressions du langage.

$$E ::= \texttt{cte} \mid \texttt{ident} \mid \texttt{let ident = } E \texttt{ in } E \mid \texttt{ident(}E\texttt{,}E\texttt{)} \mid \texttt{(}E\texttt{)}$$

Un type CAML possible pour les arbres de syntaxe abstraite de ce langage est

```
type ast = Cte of int | Var of string
         | Let of string * ast * ast | Call of string * ast * ast
```

Dans une expression `let` x `=` e_1 `in` e_2, x est une nouvelle variable qui peut être utilisée uniquement dans l'expression e_2. L'évaluation de l'expression complète consiste à évaluer e_1, à stocker la valeur obtenue dans une case mémoire m puis à évaluer e_2 en accédant à la case mémoire m à chaque utilisation de x.

On cherche à produire, pour chaque expression E, du code MIPS dont l'exécution a pour effet de calculer la valeur de l'expression E dans un registre $\$r$. On suppose que toutes les variables locales ont un nom différent, et qu'une variable sera toujours stockée à un emplacement fixé à partir de la base de la pile. On suppose que pour chaque fonction prédéfinie f, il y a un label `label`(f) auquel sont associées les instructions machines correspondant au calcul de f. Ces fonctions prennent en argument deux entiers et renvoient un entier, qui est stocké dans le registre $\$v0$.

1. Proposer une fonction `code` qui produit pour chaque expression E les instruction correspondant à l'évaluation de cette expression. On supposera donnée une fonction `index` qui à chaque variable locale x de E associe un entier n qui représente le décalage de l'adresse où x est stockée, par rapport à la base du tableau d'activation $\$fp$. On supposera aussi que $\$fp$ et $\$sp$ sont déja positionnés avec l'espace réservé pour les variables locales de E.

2. Soit l'expression

   ```
   let x = 1 in
   let t = let u = 2 in plus(u,3) in
   f(let y = h(2,t) in g(y,x), let z = 4 in g(z,z))
   ```

   Proposer un décalage pour les variables de cette expression, qui n'utilise que deux cases mémoire en supposant que dans l'expression $f(e_1, e_2)$, l'expression e_1 est évaluée avant e_2.

3. On se propose de mettre en œuvre une analyse fine pour le calcul de `index` qui prenne en compte l'utilisation effective des variables.

   Pour réaliser cette analyse, on transforme l'expression en une suite d'affectations élémentaires de la forme $v := t$. La variable v peut être soit une variable du programme $(x, y, \ldots)$ soit une variable spéciale (notée r_1, r_2, $\ldots$) qui représente alors un emplacement "anonyme" dans la pile. Le terme t peut être une constante c, une variable de programme x ou bien un appel de fonction $f(r_i, r_j)$.

   On introduit le langage intermédiaire suivant:

   ```
   type v =  Var of string | Stack of string
   ```

```
type sexpr =
    Scte of int
  | Svar of v
  | Scall of string * v * v
```

Pour compiler une expression e on se donne une variable r dans laquelle on souhaite stocker la valeur de e. Si e est une constante ou une variable, on utilise l'affectation correspondante. Si e est de la forme `let` $x = e_1$ `in` e_2 on engendre les affectations élémentaires pour calculer la valeur de l'expression e_1 dans la variable x puis les affectations pour calculer e_2 dans la variable r. Si e est de la forme $f(e_1, e_2)$, on crée des nouvelles variables r_i et r_j, on engendre les affectations pour calculer e_1 dans r_i puis e_2 dans r_j et on ajoute l'affectation $r := f(r_i, r_j)$.

(a) Donner la suite d'affectations correspondant à l'expression de la question 2.

(b) Fournir pour chaque point du programme ainsi obtenu l'ensemble des variables vivantes. On ne prendra pas en compte les variables de pile r_i. Construire le graphe d'interférence et en déduire une manière d'allouer chacune des variables.

(c) Proposer une fonction `transforme` qui étant données une variable r et une expression e, calcule la suite d'affectations équivalente pour le calcul de la valeur de e dans la variable r. La suite d'affectations sera représentée par une liste de paires, en ordre inverse (la dernière affectation en tête de liste). On commencera par écrire une fonction `gen_stack` qui génère à chaque appel une nouvelle variable de la forme `Stack "_r_`i`"`.

4. On se donne une suite d'affectations p, proposer une fonction `vivantes` qui calcule l'ensemble des variables vivantes au début de chaque instruction de p (qui seront donc utilisées avant d'être redéfinies). On pourra utiliser les notations ensemblistes suivante : l'ensemble vide `empty`, le singleton `(singleton x)`, l'union de deux ensembles `(union e1 e2)` et la différence entre deux ensembles `(diff e1 e2)`.

5. Proposer un algorithme pour calculer les décalages optimisés pour les variables locales du programme.