Cours de Compilation-Exercices

Master d'Informatique M1 2011–2012

24 octobre - 2 novembre 2011

5 Génération de code élémentaire

5.1 Passage de paramètres

Soit le programme:

```
program param;
var x : int;
procedure proc(y:int);
   begin x:=x+y; y:=y+1 end;
begin x:=2; proc(x); print(x) end;
```

Quel est le résultat de ce programme lorsque le paramètre `y` de `proc` est passé :

1. par valeur,
2. par référence,
3. par copie/restauration,
4. par nom.

5.2 Tableaux d'activation et récursion terminale

On reprend le langage étudié lors de l'analyse sémantique. Ce langage manipule des entiers et des booléens et permet de définir des variables et des fonctions.

Lors de la phase de génération de code, les localisations ne sont plus nécessaires dans l'arbre de syntaxe abstraite ni les types. On supposera que la phase d'analyse sémantique a reconstruit un programme dans le type suivant (cf fichier `tree.mli`) en attribuant des noms uniques à chaque variable. Chaque utilisation de variable est de plus annotée afin de distinguer les variables locales des variables globales.

```
type local = Loc   | Glob
type cte = Int of int | Bool of bool
type op = Plus | Mult | Div | Minus | Leq | Geq | Lt | Gt | Eq
type expr =
  | Cst of cte
  | Var of local * string
  | Binop of op * expr * expr
  | Letin of string * expr * expr
  | If of expr * expr * expr
  | Call of string * expr list
type decl =
  | Vdecl of string * expr
  | Fdecl of string * string list * expr
type program = decl list
```

Le résultat de l'évaluation d'un programme formé d'une suite de déclarations consiste à imprimer les valeurs de chaque variable globale introduite.

Ce langage est compilé vers du code MIPS, néanmoins on choisit un schéma de compilation naïf dans lequel les variables locales sont stockées sur la pile. On pourra par contre choisir de stocker les calculs intermédiaires soit uniquement dans la pile soit en exploitant quelques registres.
Dans la suite on considère le programme P suivant:

```
let a = 4;
let square(z:int):int = z*z;
let f(x:int,y:int):int = square(x)+square(y);
let result = let b = 2 in f(a,b)+a
```

1. Donner pour le programme P une manière de stocker les variables globales et les paramètres des fonctions dans la pile. Indiquer pour chaque variable les codes d'accès en lecture et en écriture.
2. Donner le code compilé correspondant au programme P. Pour cela on complètera le fichier `exemple.asm` qui contient une pré-routine pour l'impression ainsi que la structure globale du programme assembleur.
3. On se propose de traduire chaque programme de notre langage vers le code de la machine à pile.
   (a) Construire une fonction qui analyse le programme et attribue à chaque variable locale son décalage par rapport au pointeur $\$fp$ et à chaque variables globale, la taille de l'emplacement mémoire nécessaire pour stocker les variables locales introduites par un **let in** dans la définition. Pour cela on complètera les fonctions `locals_expr` et `locals_program` du fichier `code.ml`.
   (b) Compléter la fonction `code_expr` du même fichier qui produit le code pour le calcul d'une expression e dans un registre r. Pour cela on utilisera les constructions pour l'assembleur MIPS du fichier `mips.ml`.
   (c) En déduire une fonction `code_program` qui engendre le code assembleur.
4. En supposant qu'il n'y a pas de fonctions récursives, calculer pour chaque programme la taille maximale de la pile utilisée lors de l'exécution de ce programme.
5. On suppose maintenant que les fonctions peuvent être récursives. Que peut-on dire sur la taille de la pile ?
6. La récursion terminale, est un cas particulier de récursion où on peut limiter la taille de la pile.
   On dira qu'un identificateur f est *terminal* dans une expression e si l'une des conditions suivantes est vérifiée :
   – f n'apparaît pas dans e,
   – $e = f(e_1, \ldots, e_n)$ et f n'apparaît dans aucun des e_i,
   – $e =$ `if` e_0 `then` e_1 `else` e_2 et f n'apparaît pas dans e_0 et est terminal dans e_1 et e_2;
   – $e =$ `let` $x = e_1$ `in` e_2 et f n'apparait pas dans e_1 et est terminal dans e_2.
   Une déclaration : `let` $f(x_1 : \tau_1, \ldots x_n : \tau_n) : \tau = e$ est récursive terminale si f est terminale dans e.
   On peut par exemple écrire une version récursive terminale de la factorielle :

   ```
   let fact_it(n:int,m:int)= if n<=0 them m else fact_it(n-1,n*m);
   let fact(n:int):int = fact_it(n,1)
   ```

   Décrire une manière de calculer si une déclaration de fonction est récursive terminale.
7. Soit f un identificateur terminal dans une expression e. Montrer que si le calcul de e fait un appel à $f(u_1, \ldots, u_n)$ alors c'est que la valeur de e est la même que la valeur de $f(u_1, \ldots, u_n)$.

8. Si le corps de la fonction récursive terminale $f(x_1, \ldots, x_n)$ fait un appel à $f(u_1, \ldots, u_n)$, on décide d'empiler les valeurs de $u_1, \ldots, u_n$ à la place des arguments $x_1, \ldots, x_n$ et de mettre la valeur de retour de $f(u_1, \ldots, u_n)$ à la place de la valeur de retour $f(x_1, \ldots, x_n)$.
   Comment faut-il modifier la génération de code correspondant à un appel $id(u_1, \ldots, u_n)$ lorsque l'on compile le corps de la fonction récursive terminale f et que $id{=}f$.
9. Étendre le calcul de la taille maximale de la pile au cas de déclarations de fonctions récursives terminales compilées de la manière précédente.
10. En utilisant cette stratégie, calculer la taille de la pile nécessaire à l'évaluation de `fact(10)`.

Rappels sur les instructions MIPS

– initialisation

`li`	$r0, C	$r0 ←C
`lui`	$r0, C	$r0 ←$2^{16}$ C
`move`	$r0, $r1	$r0 ←$r1

– arithmétique entre registres:

`add`	$r0, $r1, $r2	$r0 ←$r1+$r2
`addi`	$r0, $r1, C	$r0 ←$r1+C
`sub`	$r0, $r1, $r2	$r0 ←$r1-$r2
`div`	$r0, $r1,$r2	$ro ←$r1/$r2
`div`	$r1, $r2	$lo ←$r1/$r2; $hi ←$r1 **mod** $r2
`mul`	$r0, $r1, $r2	$r0 ←$r1 × $r2 (pas d'overflow)
`neg`	$r0, $r1	$r0 ←-$r1

– Test égalité et inégalité

`slt`	$r0, $r1, $r2	$r0 ←1 si $r1<$r2 et $r0 ←0 sinon
`slti`	$r0, $r1, C	$r0 ←1 si $r1< C et $r0 ←0 sinon
`sle`	$r0, $r1, $r2	$r0 ←1 si $r1≤$r2 et $r0 ←0 sinon
`seq`	$r0, $r1, $r2	$r0 ←1 si $r1=$r2 et $r0 ←0 sinon
`sne`	$r0, $r1, $r2	$r0 ←1 si $r1≠$r2 et $r0 ←0 sinon

– Stocker une adresse :

`la`	$r0, adr	$r0 ←adr

– Lire en mémoire :

`lw`	$r0, adr	$r0 ←mem[adr], lit un mot

– Stocker en mémoire

`sw`	$r0, adr	mem[adr] ←$r0, stocke un mot

– Instructions de saut

`beq`	$r0, $r1, label	branchement conditionnel si $r0=$r1
`beqz`	$r0, label	branchement conditionnel si $r0=0
`bgt`	$r0, $r1, label	branchement conditionnel si $r0¿$r1
`bgtz`	$r0, label	branchement conditionnel si $r0¿0

Les versions `beqzal`, `bgtzal`...assurent de plus la sauvegarde de l'instruction suivante dans le registre $\$ra$.

– Saut inconditionnel dont la destination est stockée sur 26 bits

`j`	label	branchement inconditionnel
`jal`	label	branchement inconditionnel avec sauvegarde dans $ra de l'instruction suivante
`jr`	$r0	branchement inconditionnel à l'adresse dans $r0

aussi `jalr` ...