Examen de Compilation

13 Janvier 2010 – Durée : 3h

Documents autorisés : cours, TPs, projets, documentation MIPS

Livres interdits

Le barème ci-dessous est indicatif et est susceptible d'être modifié

La représentation du code intermédiaire est rappelée en annexe.

Exercice 1 [5 points]

Considérons le programme CTigre suivant :

```

function f(x : int) : int =
  var a := x in
    function g(y : int) : int =
      function h(z : int) : int =
        var b := a+z in
          b+y
        in h(y+1)
      in g(x)
    in printint(f(1))

```

1. Indiquez pour chaque fonction la valeur de l'attribut *level* et, pour chaque variable, la valeur des attributs *level* et *offset* (le niveau du programme principal étant 1, et l'offset ou décalage d'une variable étant sa position dans le bloc d'activation).
2. Donnez la structure du cadre de pile pour chaque fonction apparaissant dans ce programme.
3. En suivant les conventions du cours et du projet, indiquez l'évolution de la pile lors de l'exécution de ce programme, en supposant *FP* et *SP* initialisés à l'adresse 1000 (donnez une présentation détaillée de chaque bloc d'activation, avec description de chaque case [lien statique, ancienne valeur de *FP*, variables locales, etc.]).
4. Donnez le code intermédiaire qui serait produit par le compilateur pour accéder (en lecture) aux variables *a*, *b*, et *y* dans le corps de la fonction *h*.
5. Vérifiez, sur l'état de la pile au moment où la fonction *h* est en cours d'exécution, que le résultat de la question précédente est correct (faire un schéma).

Exercice 2 [5 points]

Considérons la séquence d'instructions assembleur abstrait MIPS suivante :

```

start:
  add x u z

```

```

loop:
    beqz x next
    add u x x
    beqz u loop
next:
    li t 32
    add u u u
    b start

```

- Construire le graphe de flot correspondant ;
- Calculer la durée de vie des temporaires t, x, u, z , en utilisant l'une des méthodes vues en cours ;
- Construire le graphe d'interférence associé ;
- Appliquez la méthode vue en cours pour colorier ce graphe avec 3 couleurs. Détailler chaque étape de l'algorithme. Si une réécriture du code est nécessaire, arrêtez-vous juste après la première réécriture.

Exercice 3 [3 points]

Linéariser le code intermédiaire suivant. Commencez par remonter et supprimer les ESEQ, puis réarrangez les blocs de base comme vu en cours.

Attention : ne recopiez pas tout le code... Définissez des blocs, numérotez-les et travaillez avec les numéros.

```

[LABEL L1
  CJUMP(EQ, ESEQ([CJUMP(GT,
                    MEM(BINOP(MINUS, TEMP t30, CONST 4)),
                    CONST 0,
                    L4,
                    L5);
          LABEL L4;
          MOVE(TEMP t102, CONST 1);
          JUMP(L6);
          LABEL L5;
          MOVE(TEMP t102, CONST 0);
          LABEL L6],
        CONST 1,
        L2,
        L3);
  LABEL L2;
  EXP(ESEQ(MOVE(MEM(BINOP(MINUS, TEMP t30, CONST 4)),
                BINOP(MINUS, MEM(BINOP(MINUS, TEMP t30, CONST 4)), CONST 1)),
        CONST 0));
  JUMP(L1);
  LABEL L3;
  EXP (CONST 0)]

```

Problème [7 points]

Le modèle de compilation que nous avons vu en cours n'est pas adapté aux langages fonctionnels comme OCaml, Haskell ou Scheme ; le but de cet exercice est de montrer concrètement où se trouve le problème.

Remarque : Les principales fonctions de saut de l'assembleur MIPS sont rappelées en annexe.

Fonctions en paramètres

On souhaite étendre CTigre pour permettre de passer des fonctions en paramètres aux fonctions. Par exemple

```
function apply(f:int -> int, x:int) : int =  
    f(x)
```

1. Que doit-on empiler comme premier paramètre pour faire un appel à la fonction `apply` ci-dessus sur une fonction `g` (et un entier) ? Quelle est l'occupation mémoire sur la pile de ce paramètre ? Conserve-t-on la représentation uniforme des données ?
2. Écrire le code MIPS correspondant à l'appel `apply(g, 4)`.
3. Écrire le code MIPS correspondant au corps de la fonction `apply` (sans prologue ni épilogue), en suivant les conventions du cours.
4. Quelles modifications (éventuelles) du code intermédiaire proposez-vous ?
5. Donnez la traduction en code intermédiaire de l'appel `apply(g, 4)`.
6. Donnez la traduction en code intermédiaire du corps de la fonction `apply`.

Fonctions qui retournent des fonctions

Considérons maintenant un langage semblable à CTigre mais qui permettrait de retourner des fonctions comme résultat de l'appel d'une fonction. On pourrait alors écrire des fonctions comme `foo` :

```
function foo(x:int) : int -> int =  
    function g(z:int) : int = z+z in  
    g
```

et `bar` :

```
function bar(x:int) : int -> int =  
    function h(z:int) : int = z+x in  
    h
```

1. Quel doit être le résultat de l'exécution de `foo(3)(4)`, `foo(4)(4)`, `bar(3)(4)`, `bar(4)(4)` ?
2. Indiquez les valeurs de *level* et *offset* pour les variables `x` et `z`.
3. Montrez l'évolution de la pile lors de l'exécution de `foo(3)(4)`. Au moment de l'exécution de `g`, y a-t-il un problème ?
4. Montrez l'évolution de la pile lors de l'exécution de `bar(3)(4)`. Au moment de l'exécution de `h`, y a-t-il un problème ?
5. Suggérez une technique permettant de résoudre le problème mis en évidence ci-dessus.

Annexe A – MIPS : quelques instructions de saut

Sauts court (décalage sur 16 bits)

b label	<i>Branch instruction</i> †
Saut (court) inconditionnel à label.	
beq Rsrc1, Src2, label	<i>Branch on Equal</i>
Saut (court) conditionnel à label si le contenu de Rsrc1 est égale à Src2.	
bgt Rsrc1, Src2, label	<i>Branch on Greater Than</i> †
Saute à label si le contenu de Rsrc1 est plus grand que celui de Src2	
blt Rsrc1, Src2, label	<i>Branch on Less Than</i> †
bne Rsrc1, Src2, label	<i>Branch on Not Equal</i>

On dispose aussi de sauts « longs » (décalage sur 26 bits).

j label	<i>Jump</i>
Saut long à label.	
jal label	<i>Jump and Link</i>
jalr Rsrc	<i>Jump and Link Register</i>
Saut long à label ou à l'adresse contenue dans Rsrc. Sauve l'adresse de la prochaine instruction dans le registre 31. Cela sert pour les appels de fonction.	
jr Rsrc	<i>Jump Register</i>
Saut long inconditionnel à l'adresse contenue dans Rsrc.	

Annexe B – code intermédiaire

```
type stm =
  | LABEL of lbl
  | JUMP of lbl
  | CJUMP of relop * exp * exp * lbl * lbl
  | MOVE of exp * exp
  | EXP of exp
and stms = stm list
and exp =
  | BINOP of binop * exp * exp
  | MEM of exp
  | WORDMUL of exp
  | FP
  | TEMP of temp
  | ESEQ of stms * exp
  | CONST of int
  | STR of lbl
  | CALL of lbl * exp list
```