MASTER D'INFORMATIQUE 2008–2009

Examen de Compilation

6 Janvier 2009 – 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 :

```
let main() =
  type t = {v1 : int, v2 : int} in
  let r : t := t {v1=44, v2=22} in
  let i := r.v1 in
  and a := 5 in
  let g(x : int, w:int) : int =
      let f(y : int) : int = y * i in
      let h(z : int) : int = (w + f(z)) / a in
      f(h(x)) - r.v2
  in
    g(i,a); 0
in main()
```

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 et du tas lors de l'exécution de ce programme, en supposant *FP* et *SP* initialisés à l'adresse 2000 et le tas libre à partir de l'adresse 1000 sur une architecture 32 bits (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 aux variables `a`, `z` et `w` dans le corps de la fonction `h`, et à `r.v2` dans le corps de `g`.
5. Vérifiez, sur l'état de la pile au moment où la fonction `g` est en cours d'exécution, que le résultat de la question précédente est correct (faire un schéma).
6. Pour chacune des variables du programme et des paramètres de fonctions, expliquez s'il serait possible d'utiliser un registre plutôt que la pile (et pourquoi ?).

Exercice 2 [5 points]

Considérons la séquence d'instructions assembleur abstrait MIPS suivante :

```
start: addi x x 1
       add y x 1
       bne z y start
cont:  addi t z 1
       beq t x cont
```

1. Construisez le graphe de flot correspondant ;
2. Calculez la durée de vie des temporaires x, y, z, t, en utilisant l'une des méthodes vues en cours ;
3. Construisez le graphe d'interférence associé ;
4. 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.
5. Même question avec 2 couleurs.

Exercice 2 [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.

```
MOVE(TEMP 2,
  ESEQ([
    CJUMP(EQ,
      ESEQ([
        CJUMP(EQ,
              MEM(BINOP(MINUS, TEMP 30, CONST (-4))),
              CONST 3,
              "L4__EQ_11",
              "L5__EQ_12");
        LABEL "L4__EQ_11";
        MOVE(TEMP 103,CONST 1);
        JUMP("L6__EQ_donel");
        LABEL "L5__EQ_12";
        MOVE(TEMP 103,CONST 0);
        LABEL "L6__EQ_donel"],
        TEMP 103),
      CONST 1,
      "L1__then",
      "L2__else");
    LABEL "L1__then";
    MOVE(TEMP 102,CONST 4);
    JUMP("L3__done");
    LABEL "L2__else";
    MOVE(TEMP 102,CONST 5);
    LABEL "L3__done"],
    TEMP 102))
```

Problème [7 points]

Sauf dans la question 1, nous nous plaçons dans le cas général d'un compilateur pour lequel les paramètres des fonctions peuvent être des pointeurs vers le tas, des entiers, codés sur un mot, ou des flottants double précision, codés sur deux mots.

1. Proposez des modifications du langage CTigre du projet pour ajouter du polymorphisme paramétrique (à la OCaml) : syntaxe pour les types et modifications éventuelles du compilateur. Expliquez quelle(s) caractéristique(s) de notre compilateur rend cette extension simple. On ne demande pas les modifications à apporter à la vérification (ni à l'inférence) des types.
2. Quelle solution proposez-vous pour retourner une valeur de type flottant double précision avec le processeur MIPS ?
3. Quel(s) problème(s) le polymorphisme paramétrique pose-t-il pour la gestion automatique de la mémoire (garbage collector) ? Nous allons dans la suite explorer plusieurs solutions possibles.
4. Une première solution pour compiler une fonction polymorphe consiste à générer un code monomorphe pour tous les cas d'utilisation de la fonction dans le programme. On apelle cela la *monomorphisation*. C'est la solution utilisée par Ada, ou pour les templates de C++.
   - Voici un exemple de programme écrit avec la syntaxe d'OCaml :

     ```
     let rec f (n : int) (x : 'a) : 'a list =
       if n = 0 then [] else x::(f (n-1))
     in
     let _ = f 3 4 in
     let _ = f 3 [] in
     ()
     ```

     Écrivez le programme équivalent après monomorphisation.
   - Proposez un algorithme de monomorphisation.
   - La monomorphisation fonctionne-t-elle dans le cas suivant et pourquoi ? (toujours avec la syntaxe OCaml) :

     ```
     let rec blowup (n : int) (x : 'a) : 'a =
       if n = 0 then x else List.hd (blowup (n-1) [x])
     in
     blowup 4 0
     ```

     (`List.hd` est une fonction qui renvoie la tête d'une liste).
   - Quels sont les avantages et inconvénients de la monomorphisation ?
5. Une autre solution consiste à « boxer » (*mettre en boîte*) toutes les valeurs, c'est-à-dire placer toutes les données (y compris les entiers) dans le tas et utiliser des pointeurs pour les manipuler.
   - Quels sont les avantages et les inconvénients de cette technique ?
   - Quelle variante de cette technique est utilisée par OCaml pour ne pas avoir à « boxer » les entiers ou les caractères ? Quelles sont les avantages et les inconvénients de cette variante ?
6. Une autre variante de la solution précédente consiste à ne « boxer » les valeurs immédiates que lorsqu'elles sont utilisées comme paramètres polymorphes. Des coercions automatiques sont ajoutées là où elles sont nécessaires.

- Rajoutez explicitement les fonctions de coercion dans le programme suivant :

```
let id (x : 'a) : 'a = x in
let rec f (n : int) (x : 'a) : 'a list =
  if n = 0 then [] else x::(f (n-1))
in
let _ = id [] in
let _ = 1 + id 4 in
let _ = f 1 [] in
let _ = 1 + somme (f 1 4) in                    (***)
()
```

  où `somme : int list -> int` fait la somme des éléments d'une liste d'entiers.
- Expliquez en détail ce que font les fonctions de coercion utilisées ci-dessus.
- Décrire l'évolution de la pile et du tas lors de l'exécution de la ligne marquée *(***)*.
- Quels sont les avantages et inconvénients de cette technique ?
- Proposez une variante des fonctions de coercion pour les caractères (8 bits) qui évite d'allouer dans le tas (en s'inspirant de l'astuce utilisée par OCaml). Est-il possible de faire la même chose pour les entiers ?

7. Enfin une dernière solution consiste à passer explicitement le type à la fonction, en plus des valeurs polymorphes.
   - En fait il n'est pas nécessaire de donner toute l'information sur le type. En supposant les flottants double précision boxés, quelle information minimale est suffisante pour le garbage collector ?
   - Même question si les flottants double précision ne sont pas boxés.
   - Quelle(s) autre(s) modification(s) apporter au compilateur si les flottants double précision ne sont pas boxés ? Détailler les modifications à apporter à chaque phase du compilateur.

Annexe : code intermédiaire

```
type binop =
  | PLUS | MINUS | MUL | DIV
  | AND | OR
  | LSHIFT | RSHIFT | ARSHIFT | XOR
type relop = EQ | NE | LT | GT | LE | GE | ULT | ULE | UGT | UGE
type stm =
  | LABEL of Temp.label
  | JUMP of Temp.label
  | CJUMP of relop * exp * exp * Temp.label * Temp.label
  | MOVE of exp * exp
  | EXP of exp
and stms = stm list
and exp =
  | BINOP of binop * exp * exp
  | MEM of exp
  | TEMP of Temp.temp
  | ESEQ of stms * exp
  | CONST of int
  | STR of Temp.label
  | CALL of Temp.label * exp list
```