Programmation TP 5 : Génération de code

{ jrobert, fbouchez, apardon } @ens-lyon.fr
http://perso.ens-lyon.fr/florent.bouchez/

16 octobre 2007

Message de Daniel : Le TP d'aujourd'hui est difficile, pas vraiment dans ce qu'il faut coder, mais plutôt parce qu'il implique de bien comprendre d'une part le principe de la génération de code et d'autre part comment tout cela s'articule au sein des différentes étapes de la compilation. Rassurez-vous, on reviendra à des trucs plus digestes plus tard.

Dans ce TP, on va compiler un langage assez bas niveau, écrit en Ocaml, vers l'assembleur que vous utilisez pour votre projet avec Sebastien Briais. Un interpréteur de cet assembleur est disponible sur son home (version normale ou `.opt`) : `/home/sbriais/bin/risc-emuf`

Commencez par récupérer les fichiers du TP : l'ensemble des opérations à effectuer pourra se faire directement à partir de définitions OCaml. Le fichier `RISC.pdf` contient un rappel de la syntaxe assembleur.

1 Génération de code

La fonction `gen_expr`, qui génère le code correspondant à une expression, est très incomplète. Elle correspond à ce qui a été vu en cours. Les types des expressions et programmes que l'on va compiler sont les suivants : Dans notre langage, un programme est une succession de définitions d'expressions (les variables et fonctions) suivie d'une expression particulière (le 'main'). Voici sa description :

```
type expr =
  | Const of int                  (* constante *)
  | Add of expr*expr              (* addition *)
  | Mul of expr*expr              (* multiplication *)
  | IfTE of expr*expr*expr*expr   (* if e1=e2 then e3 else e4 *)
  | Call of string*expr list      (* appel de fonction *)
  | Var of int                    (* i-eme argument d'une fonction *)

(* une definition: un nom, une arite, un corps *)
type def = string * int * expr

(* un programme : des definitions et une expression *)
type prog = def list * expr
```

Q 1.1 Complétez la fonction `gen_expr` présente dans le fichier récupéré.

2 Optimisation de l'usage de la pile

Q 2.1 Comparez la taille maximale de la pile lors de l'éxecution de « `8+(1+(2+(3+(4+0))))` » et de « `((((0+4)+3)+2)+1)+8` ». Quel est le problème ?

Q 2.2 Écrivez une fonction de réordonnancement d'une expression, telle que sa compilation « gauche-droite » soit optimale, en termes d'usage de la pile.

Par exemple, « `reorder (2+(1+3))` » et « `reorder ((4*(5+1))+1)` » doivent respectivement renvoyer « `((1+3)+2)` » et « `(((5+1)*4)+1)` ».

Ici, on cherche à organiser une suite d'opérations symétrique (l'addition et la multiplication), mais pas reparenthèser une expression.

3 Récursivité terminale

Un appel *récursif terminal* est un appel de fonction tel que le résultat de cet appel est exactement le résultat de la fonction que l'on est en train de définir. Testez vous-même :

```
let rec fact x =
  if x=0 then 1
  else x * fact (x-1) ;;

fact 100000;;
```

```
let rec fact_tail_rec accu x =
  if x=0 then accu
  else fact_tail_rec (accu * x) (x-1) ;;

fact_tail_rec 0 100000;;
```

Dans le premier cas, OCaml ne peut rien faire, car il doit attendre que la fonction récursive renvoie une valeur. Un enregistrement d'activation est donc créé par appel de fonction, et la pile finit par déborder. Dans le deuxième cas, OCaml se rend compte que l'appel récursif est le dernier calcul effectué dans la fonction, on se débrouille alors pour que la fonction appelée utilise le même enregistrement que la fonction en cours.

Q 3.1 Écrivez une fonction qui parcourt une expression et en annote les appels de fonction selon qu'ils sont récursifs terminaux ou non. Modifiez le constructeur `Call` pour qu'il prenne en argument une étiquette : un appel non récursif terminal doit être étiqueté par `(-1)`, et un appel récursif terminal par le nombre d'arguments de l'appelant.

Q 3.2 Modifiez le code de la fonction de compilation pour qu'elle tienne compte de cette annotation, et compile efficacement les appels récursifs terminaux.

4 Transformation de programme : *lambda lifting*

On s'intéresse à la définition suivante :

```
let f (x,y) =
  let g u = u*x in
  (g y) + (g (y+x))
```

Q 4.1 Quelle est la propriété vérifiée par `g` qui pose problème dans le schéma qui a été adopté pour la compilation des fonctions ?

Expliquez la difficulté et illustrez par un exemple.

Pour résoudre ce problème, on peut passer par une phase de transformation du code, afin de « sortir » la définition de `g` du corps de `f` dans l'exemple ci-dessus. L'idée est de rajouter un paramètre à la fonction `g`. Ceci aura pour effet de modifier à la fois la définition de `g` et les appels à `g` :

1.
```
let f (x,y) =
  let g u v = u*v in
(g y x) + (g (y+x) x)
```

2.
```
let g u v = u*v
let f (x,y) =
  (g y x) + (g (y+x) x)
```

On se donne les expressions étendues suivantes :

```
type eexpr =
  | EConst of int
  | EAdd of eexpr * eexpr
  | EMul of eexpr * eexpr
  | EIfTE of eexpr * eexpr * eexpr * eexpr
  | ECall of string * int * eexpr list

    (* let f x1 ... xn = e in e' *)
  | ELet of string * string list * eexpr * eexpr

    (* x *)
  | EVar of string
```

Les variables sont désormais représentées par des noms plutôt que des indices, et l'on peut imbriquer les définitions de fonctions. Notez cependant que l'on ne peut pas définir de fonctions d'ordre supérieur : la construction fonctionnelle `Fun` n'est pas dans la syntaxe. Ainsi, tous les arguments des fonctions que l'on manipule sont des entiers.

Q 4.2 Écrivez une fonction `lift`, qui transforme une expression de type `eexpr` en un programme compilable, de type `prog`.