Cours de Compilation-Exercices

Master d'Informatique M1 2011–2012

Semaine 8 — 7 novembre 2011

1 Rappels

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
```

On choisit un schéma de compilation où toutes les variables (arguments de fonction et variables locales) sont stockées sur la pile. Chaque fonction trouvera ses arguments sur la pile; le code de la fonction devra empiler, au début de son exécution, les valeurs sauvegardées des registres `$ra` et `$fp`, et définir la nouvelle valeur de `$fp` comme le sommet de pile à ce stade; ensuite, on devra réserver sur la pile l'espace nécessaire pour stocker les variables locales; au delà de cette zone, la pile sera utilisée pour stocker des calculs intermédiaires.

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 le code compilé pour les accès aux variables a, x et b.
2. Donner le code compilé pour les déclarations de a et *square* dans le programme P.

Nous allons maintenant reprendre un compilateur déja implémenté selon les conventions décrites plus haut, et réaliser deux extensions. Le code est à télécharger sur la page du cours.

Une fois téléchargé, la commande `make test` devrait compiler et tester le compilateur (les tests portent sur la valeur de la dernière variable déclarée, qui est affichée par le code assembleur généré). Quand vous modifiez le code, assurez vous que les tests passent toujours.

Prenez connaissance du fichier `code.ml`, en particulier la fonction de génération de code mais aussi la fonction `preproc` qui effectue une analyse préalable du programme pour calculer les informations liées aux tableaux d'activation.

2 Récursion terminale

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)
```

1. Soit f un identificateur terminal dans une expression e. Montrer que quand le calcul de e fait un appel à $f(u_1, \ldots, u_n)$, la valeur retournée est directement celle de $f(u_1, \ldots, u_n)$.
2. 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 $f(x_1, \ldots, x_n)$.
   Donner un code assembleur pour `fact_it` qui respecte cette convention.
   Quelle est la taille maximale de pile utilisée par cette implémentation de la factorielle?
3. Modifier la génération de code pour optimiser les appels récursifs terminaux: (1) détecter les fonctions récursives terminales dans la phase de préprocessing, et remplir une table indiquant pour chaque fonction du programme si elle est récursive terminale ou non, puis (2) adapter la génération de code pour optimiser les appels récursifs des fonctions récursives terminales. Vérifier que la génération de code modifiée produit le bon code pour la factorielle.

3 Tableaux

On ajoute à notre langage des tableaux d'entiers. La construction d'un nouveau tableau s'écrit `[a;b;c;d]`, et l'accès à une case est noté `t[i]`. Les tableaux sont passés par référence.

1. Adapter notre code pour supporter cette extension du langage. Pour cela, ajouter:
   - un nouveau type d'expression dans `base.ml` pour les tableaux d'entiers,
   - de nouvelles expression pour dénoter la construction d'un tableau et l'accès à une case dans `ast.mli` et `tree.mli`,
   - les tokens pour les symboles `[` et `]`, et étendre les analyseurs lexicaux et syntaxiques pour accepter ces constructions dans les fichiers sources.

   Étendre enfin l'analyse sémantique: les expressions composant un tableau doivent être des entiers, et l'index utilisé lors de l'accès à une cellule aussi.

Pour la génération de code, on allouera les tableaux sur le tas, et on ne libèrera jamais la mémoire allouée. Pour cela, on utilise l'appel système `sbrk` pour étendre le tas, avec la suite d'instructions MIPS suivante:

```
li $a0, <size> # <size> = nombre d'octets à allouer
li $v0, 9      # code du syscall sbrk
syscall
```

A l'issue de l'exécution de ces instructions, le registre `$v0` contient l'adresse du début de la zone allouée. Les valeurs des cases du tableau seront donc stockées aux adresses `0($v0)`, `4($v0)`, etc.

2. Implémenter la génération de code pour les deux nouveaux types d'expression, de façon à passer les deux derniers tests.

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` ...