Génération de code : langages fonctionnels
13 octobre 2011

Introduction

Langages fonctionnels

- Les fonctions sont des objets de première classe.
- Les fonctions apparaissent en argument d'une fonction ou en résultat.
- Fonctions polymorphes qui travaillent sur des valeurs de n'importe quel type.

Représentation de fonctions

- Une fonction est une suite d'instructions qui sera exécutée au moment de l'appel.
- La difficulté principale est de traiter correctement les variables utilisées par les fonctions.

Sommaire

Cas du langage C

- En C, on peut manipuler des pointeurs de fonction.
- Une variable peut être déclarée comme un pointeur de fonction en précisant les types des arguments et du résultat (le profil).

```
int (*pf) (int, int);
```

- Il est alors possible d'affecter cette variable ou de l'utiliser dans une application:

```
int f(int i, int j) {return i+j;}
int g(int i, int j) {return i*j;}
int choice (int x)
{ int z;
  if (x) pf = &f; else pf = &g;
  z = *pf(1,2);
  return z;
}
```

Cas du langage C (2)

Situation simple:

- Les variables sont globales ou locales.
- Il n'y a que des fonctions globales.
- Une fonction correspond juste à une adresse dans le code.

Langage à la Pascal

- Une fonction/procédure peut utiliser:
  - des variables globales
  - ses paramètres, ses variables locales
  - les paramètres et déclarations locales des fonctions/procédures dans lesquelles elle est définie.
- A l'appel, les tableaux d'activation des procédures parentes se trouvent dans la pile et permettent l'accès aux variables non locales.
  - On connait statiquement le lien de parenté.
  - On chaine dynamiquement les tableaux d'activation.

Passage d'une fonction en paramètre

Pour compiler une procédure qui prend une fonction en paramètre, il faut deux informations:

- l'adresse du code de la fonction
- l'adresse du tableau d'activation du père de la procédure

Exemple (1)

```
let main () =
let b (h:int -> int) = print (h 2)
let c () =
    let m = ref 0 in
    let f n = !m + n
    in  b f
in c ()
```

- Tableau d'activation de *c*: variable *m*, adresse TA du père stat. de *c* (*main*)
- Tableau d'activation de *f*: adresse TA du père stat. de *f* (*c*), paramètre *n*, valeur de retour de *f* dans $*v*0
- Code de *f*:

Exemple (2)

```
let b (h:int -> int) = print (h 2)
```

- Tableau d'activation de *b*: TA du père stat. de *b* (*main*), *b* attend deux arguments: adresse du code de *h*; TA du père stat. de *h* (connu à l'appel).
- Code de *b* (`print` (*h* 2)):

Lorsque *c* doit exécuter *b f*:

- il calcule le lien d'activation du père statique de *f* (ici *c*) comme s'il l'appelait et l'empile
- il empile l'adresse du code de *f*
- il empile l'adresse du TA de *b* (*main* qui est aussi le TA du père de *c*)

Sommaire

Langages fonctionnels

Les fonctions sont des objets de première classe:

- peuvent être passées en argument;
- peuvent être retournées comme valeur.

Exemple fonction comme résultat

```
let f x = let g y = x+y in g
let h = f 3 in h 5
```

- Les fonctions peuvent être anonymes :
  fun $x \rightarrow e$ correspond à f telle que $f(x) = e$.
- On peut se contenter de fonction unaire (un seul argument):
  $f(x, y) = e$ est représentée par **fun** $x \rightarrow$ **fun** $y \rightarrow e$.
- On peut faire des applications partielles :
  $f\,3$ est une fonction.

Renvoyer une fonction comme valeur

Si la fonction f calcule une fonction g

- le corps de g peut utiliser les variables de f.
- l'exécution du corps de g accède aux variables de f.
- le TA de f disparait après la définition de g.

Il faut sauvegarder ces variables pour l'exécution de g.

Exemple

```
let f x = let g y = x+y in g
let h = f 3
let j = f 4
h 5 + j 7
```

- g utilise x et y
- h est la fonction g avec $x = 3$
- j est la fonction g avec $x = 4$
- les appels à h et j instancient y

Clotures

- Une fonction $f\ x = e$ est formée
  - code pour exécuter le corps e
  - les valeurs des variables utilisées dans e
- on choisit de stocker les variables libres de e autres que x dans un environnement linéaire
- on appelle cloture l'association du code et de l'environnement
- l'environnement est alloué dans le tas
- statiquement chaque variable utilisée dans le corps de f est associée à un décalage dans l'environnement
  Une même variable peut avoir des décalages différents dans les corps de deux fonctions différentes.

Définition des variables libres

Une variable x est libre dans une expression e si:

- elle est utilisée dans l'expression e;
- elle n'est pas liée, c'est-à-dire dans la portée d'une déclaration locale:
  fun $x \rightarrow \ldots$ ou **let** $x = t$ **in** $\ldots$

Exemple

```
let g y = x + y
```

equivalent à

```
let g = fun y -> x+y
```

Les variables libres de la définition de g sont $\{x\}$

Exemple

let f x = **let** g y = x+y **in** g

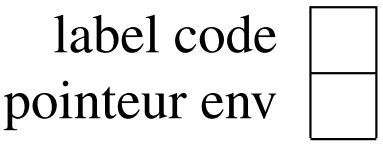
fonction	argument	environnement	corps
f:	x	[]	g
g:	y	[x]	x + y

Il faut déterminer un schéma dans lequel s'éxecute le corps d'une fonction

- On peut supposer que l'environnement sera dans le registre $a0
- Les autres arguments à partir du registre $a1 ou dans la pile
- Valeur de retour dans $v0

Valeur fonctionnelle

- Une valeur de type fonctionnel est formée de l'adresse d'un environnement et de l'adresse du code (alloués dans le tas ou sur la pile).



- On crée du code lorsqu'il y a le mot clé **fun** ($f\ x = e$ est équivalent à $f =$ **fun** $x \rightarrow e$).
- $(f\ 3)$ est une fonction : code de g, environnement $x = 3$.

Exemple

Compilation du corps de g: $(x + y)$

- Organisation de la mémoire:
  - pointeur sur l'environnement $[x]$ dans $a0,
  - y dans $a1,
  - valeur de retour dans $v0
- code pour g:

```
label_g:
lw $t0,0($a0)        # x
add $v0,$t0,$a1
jr $ra
```

Exemple (2)

Compilation du corps de f: (g)

- Organisation de la mémoire de f :
  - pointeur sur l'environnement [] dans $\$a0$,
  - x dans $\$a1$,
  - valeur de retour dans $\$v0$ (adresse environnement) et dans $\$v1$ (adresse code)

Allocation dans le tas en assembleur MIPS

- Un appel système `syscall` avec $\$v0 = 9$ permet d'allouer un bloc dans le tas.
- La taille du bloc est passée en argument dans $\$a0$.
- L'appel renvoie dans $\$v0$ l'adresse du début du bloc.
- Code possible pour une fonction malloc:

```
malloc:             # allocation dynamique
      li $v0, 9      # appel système sbrk
      syscall        # alloue une taille $a0
      jr  $ra        # retourne le pointeur dans $v0
```

- En pratique, il serait coûteux de faire cet appel à chaque création d'un nouvel objet.
- On alloue un bloc de taille suffisante, on garde dans un registre l'adresse de ce bloc.

Code pour la fonction *f*

```
let f x = let g y = x+y in g
```

```
label_f:
pushr $ra
move $t0,$a0
li $a0,4
li $v0,9
syscall              # alloue l'environnement pour g
move $a0,$t0
sw $a1,0($v0)        # sauve x dans l'environnement
la $v1,label_g
popr $ra
jr $ra
```

Appel

```
let h = f 3 (type int -> int)
let x = h 5 (type int)
```

- *f* et *h* sont des variables dont la valeur est une fonction qui occupe une place double (environnement + code).
- 2 cases mémoires à `dec`(f/h) pour $v0 (env) et `dec`(f/h)+4 pour $v1 (code).
- *code*(let h = f 3) =
  lw $a0,`dec`(f)($fp) | **li** $a1,3 | **lw** $t0,`dec`(f)+4($fp) | **jalr** $t0 |
  sw $v0,`dec`(h)($fp) | **sw** $v1,(`dec`(h)+4)($fp)
- *code*(h 5) =
  lw $a0,`dec`(h)($fp) | **li** $a1,5 | **lw** $t0,`dec`(h)+4($fp) | **jalr** $t0 |
  sw $v0,`dec`(x)($fp)

Autre exemple

```
let twice h = let t x = h (h x) in t
let h = f 3 in twice h
```

fonction	argument	environnement	corps	type
twice:	h	[]	t	$(\alpha \to \alpha) \to \alpha \to \alpha$
t:	x	[h]	h (h x)	$\alpha \to \alpha$

Schéma de compilation général

Compilation de **fun** $x \to e$

- On associe un label de code unique f pour cette fonction et on calcule l'ensemble $x_1, \ldots, x_n$ des variables utilisées dans e qui ne sont ni x, ni locales dans e.
- On engendre pour f le code de e avec pour hypothse que l'adresse de l'environnement est dans $a0 et donc l'accès à x_i : **lw** $t0,dec($x_i$)$a0 (si x_i de taille 1)
- On crée l'environnement associé à $x_1, \ldots, x_n$:
  li $a0,p # (taille totale nécéssaire pour $x_1, \ldots, x_n$)
  li $v0,9 | `syscall`
  + instantiation des x_i
- On renvoie le couple formé de l'environnement et de l'adresse du code.
  li $v1,label_f

Remarques sur la compilation des fonctions

- Dans le cadre d'un langage polymorphe, une valeur fonctionnelle doit aussi avoir une taille unitaire:
  - allocation dans le tas d'une paire formée de l'environnement et de l'adresse du code.
- Considérer les fonctions comme unaires est inefficace (création de clôtures à chaque argument)
  - Compilation d'un code spécifique si la fonction est appliquée à tous ses arguments (cas le plus fréquent qui permet d'utiliser au mieux les registres pour les arguments).

Remarques sur la génération de code

Des hypothèses qui simplifient le traitement :

- Valeurs de taille unitaire (essentiel si polymorphisme)
- Tableaux indicés à partir de zéro
- Vision uniforme des procédures et fonctions (juste la taille de la valeur de retour)
- Variables uniquement locales ou globales