Langages de Programmation, Interprétation, Compilation

Christine Paulin
Christine.Paulin@universite-paris-saclay.fr

Département Informatique, Faculté des Sciences d'Orsay, Université Paris-Saclay

LDD3 Informatique, Mathématiques - Magistère Informatique

2025–26

Fonctions et piles d'appel

Introduction

- L'organisation des programmes en fonction est un point clé des langages de programmation
- Transfert du flot de contrôle
- Transmission de données entre calculs (nécessite de fixer des règles)

Appel de fonction : vocabulaire

appel de fonction f(6)

```
1    function g() {
2      let y;
3      y = f(6);
4      print(y);
5    }

6    function f(x) {
7      return x*7;
8    }
```

- 6 est le paramètre effectif (on dit aussi argument) de l'appel f(6).
- le bloc de code des lignes 2 à 4 est le contexte appelant,
- la fonction f définie aux lignes 6 à 8 est la fonction appelée.
- du côté de la fonction appelée f, la variable x est un paramètre formel
- l'expression x*7 est le résultat renvoyé par la fonction.

Articulation appelant/appelé

Transfert de données :

- un ou plusieurs paramètres effectifs transmis par le contexte appelant à la fonction appelée,
- un résultat, renvoyé au contexte appelant par la fonction appelée.

Pour cela, on peut utiliser la pile et/ou les registres, d'une manière à convenir.

Transfert temporaire de contrôle :

- lors de l'appel, l'exécution saute au code de la fonction appelée,
- à la fin de l'appel, l'exécution revient au contexte appelant, en reprenant « juste après » le point où l'on a réalisé l'appel.

Transfert de contrôle en MIPS

- au moment de l'appel : mémoriser l'adresse de l'instruction à laquelle il faudra revenir
- En MIPS, on utilise le registre $ra (Return Address).
- instruction `jal` (Jump And Link) : saute au code de la fonction, mémorise dans $ra l'adresse de l'instruction suivante du contexte appelant
  - étiquette (statique) de la fonction appelée
  - variante `jalr` avec l'adresse donnée par un registre

```
jal     f
jalr    $t0
```

- Fin de l'appel : retour à l'appelant avec un saut à l'adresse dans $ra.

```
jr      $ra
```

Exemple

- l'unique argument de f est dans $a0
- son résultat est placé dans $v0.

```
12  g:  li   $a0, 6          # déf. param. effectif
16      jal  f               # appel
20      move $a0, $v0        # affiche du résultat
24      li   $v0, 1
28      syscall

36  f:  li   $v0, 7          # calcul
40      mul  $v0, $a0, $v0
44      jr   $ra             # fin de la fonction
```

Trace d'exécution assembleur

L'exécution du saut `jal` (adresse 16)

- stocke dans $ra l'adresse suivante 20
- saute à l'adresse 36.

instruction	$a0	$v0	$ra	pc	
				12	
`li $a0, 6`	6			16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42

Problème : appels imbriqués

On ajoute un code principal réalisant un appel à notre fonction g.

```
00  main:   jal   g             # appel à g (sans argument)
04          li    $v0, 10       # fin du programme principal
08          syscall
12  g:      li    $a0, 6
16          jal   f
20          move  $a0, $v0
24          li    $v0, 1
28          syscall
32          jr    $ra           # fin de g
36  f:      li    $v0, 7
40          mul   $v0, $a0, $v0
44          jr    $ra
```

Appels imbriqués : trace d'exécution

	instruction	$a0	$v0	$ra	pc	
					0	
00	`jal g`			4	12	
12	`li $a0, 6`	6		4	16	
16	`jal f`	6		20	36	
36	`li $v0, 7`	6	7	20	40	
40	`mul $v0, $a0, $v0`	6	42	20	44	
44	`jr $ra`	6	42	20	20	
20	`move $a0, $v0`	42	42	20	24	
24	`li $v0, 1`	42	1	20	28	
28	`syscall`	42	1	20	32	affiche 42
32	`jr $ra`	42	1	20	20	
20	`move $a0, $v0`	1	1	20	24	
24	`li $v0, 1`	1	1	20	28	
28	`syscall`	1	1	20	32	affiche 1
32	`jr $ra`	1	1	20	20	
20	`move $a0, $v0`	1	1	20	24	
...	...	...	...	...		

Partie 3 : Typage, fonctions, mémoire

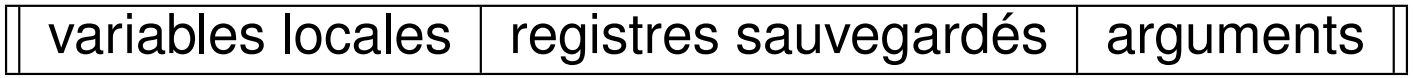
Pile d'appels

- plusieurs appels actifs à un même moment
- l'appel le plus récent est en cours d'exécution
- un certain nombre d'autres appels plus anciens sont en suspens

Tableaux d'activation

- Chaque appel de fonction actif possède son propre contexte,
  - les arguments qui lui ont été passés,
  - les valeurs de ses variables locales,
  - l'adresse de retour stockée dans $ra.
- Ce contexte doit être gardé jusqu'à ce que l'appel soit terminé
- La structure utilisée pour stocker les informations est appelée tableau d'activation (TA) (en anglais call frame).

Format des tableaux d'activation



- la partie « registres sauvegardés » contient en particulier la valeur de $ra.
- tableaux stockés dans la mémoire
- un registre dédié $fp (Frame Pointer) pour localiser le tableau actif
- à partir de $fp, on accéde aux
- le pointeur de base $fp pointe sur le dernier mot dédié aux registres sauvegardés.

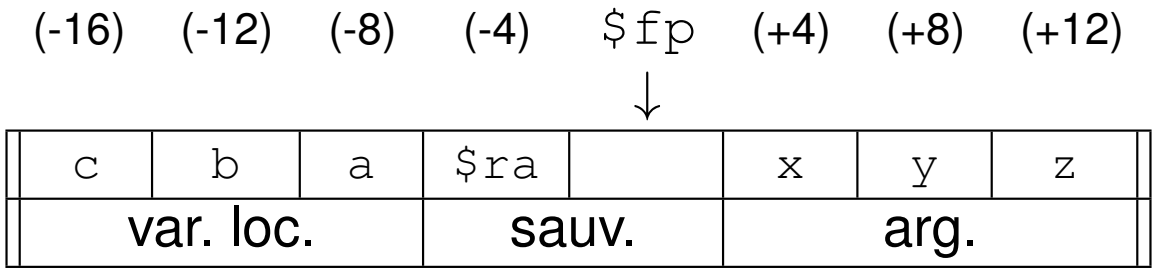
- On accède donc aux arguments à partir de $fp avec un décalage positif, et aux variables locales avec un décalage négatif.

Exemple

fonction avec trois paramètres x, y et z et trois variables locales a, b et c

```
function h(x, y, z) {
  let a, b, c;
  ...
```

tableau d'activation :



la case à l'adresse fp servira à un autre registre sauvegardé.

Pile d'appels

- Vie des tableaux d'activation :
  - création d'un tableau d'activation au début d'un appel,
  - destruction du tableau à la fin de l'appel.
- Dynamique :
  - tout appel se termine avant de rendre la main à son contexte appelant ;
  - c'est l'appel le plus récent qui termine en premier
  - c'est toujours le TA le plus récent qui sera détruit en premier.
- L'ensemble des TA a une structure de pile (Last In First Out), nommée pile d'appels (call stack).
- La pile d'appels est réalisée dans la pile MIPS elle-même

Accès aux éléments du tableau d'activation

- le premier mot du tableau d'activation correspond au sommet de la pile : pointeur $sp (Stack Pointer).

- pour faciliter l'accès au tableau d'activation précédent (celui du contexte appelant), on sauvegarde la valeur précédente de $fp,
- chaînage des tableaux d'activation, du contexte courant vers les contextes plus anciens, jusqu'à l'appel de la fonction principale de notre programme.

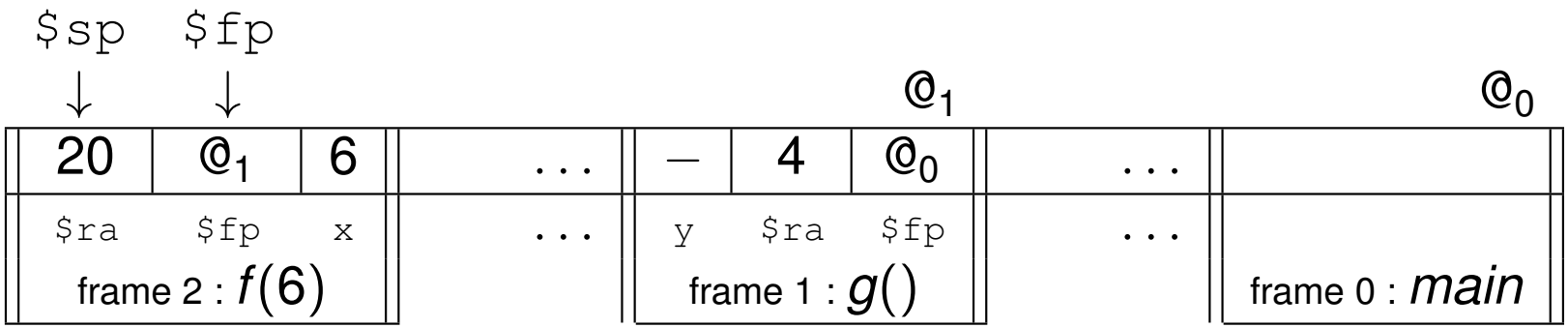
Exemple

```
1   function g() {
2     let y;
3     y = f(6);
4     print(y);
5   }
```

```
6   function f(x) {
7     return x*7;
8   }
```

Pendant l'appel f(6), trois tableaux d'activation :

- contexte principal (*main*)
- TA de g(), avec emplacement non initialisé pour la variable locale y
- TA de f(6)
  - emplacement pour le paramètre x.
  - pointeur $fp courant pointe dans ce dernier tableau
  - Les tableaux de f(6) et g() contiennent une adresse $ra sauvegardée.



Partie 3 : Typage, fonctions, mémoire

Convention d'appel

Pour assurer les bons transferts d'information et de contrôle, et la bonne gestion de la pile d'appels, le contexte appelant et la fonction appelée doivent respecter un protocole commun appelé convention d'appel.

Important aussi lorsqu'il y a des appels entre fonctions issues de différents compilateurs

Protocole minimal simplifié

Un protocole d'appel se découpe en plusieurs étapes, à la charge de l'un ou l'autre des participants.

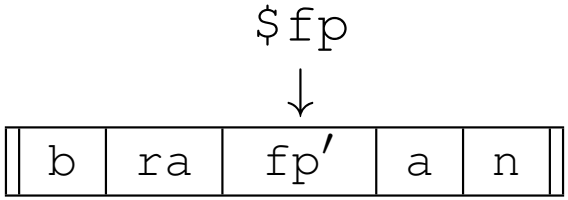
1. **Appelant, avant l'appel :** évalue les arguments et place les valeurs obtenues sur la pile, puis déclenche l'appel avec `jal` ou `jalr` pour stocker l'adresse de retour dans $ra tout en passant la main à l'appelé.
2. **Appelé, au début de l'appel :** mise en place du TA : sauvegarde les valeurs de $fp et $ra sur la pile, réserve de l'espace supplémentaire pour ses variables locales, et donne à $fp sa nouvelle valeur
3. **Appelé :** exécution du corps de la fonction appelée.
4. **Appelé, à la fin de l'appel :** place le résultat renvoyé dans le registre $t0, restaure $fp et $ra, libère l'espace de pile réservé à l'étape 2, et redonne la main à l'appelant avec `jr $ra`.
5. **Appelant, après l'appel :** retire les arguments placés sur la pile.

Exemple ImpScript

fonction d'exponentiation rapide.

```
function power(a, n) {
  if (n == 0) { return 1; }
  else {
    let b = power(a*a, n>>1);
    if (n&1 == 0) { return b; }
    else { return a*b; }
  }
}
console.log(power(2, 9));
```

La fonction power a deux arguments a et n, et une variable locale b.
Un tableau d'activation pour un appel de cette fonction a donc la forme suivante.



On accède aux arguments a et n respectivement aux adresses fp+4 et fp+8, et à la variable locale b à l'adresse fp−8.

Protocole d'appel à `power` (étape 1)

Appel power(2, 9) : placer les deux arguments au sommet de la pile,

```
li      $t0, 9              # chargement de 9
addi    $sp, $sp, -4        # passage sur la pile
sw      $t0, 0($sp)
li      $t0, 2              # chargement de 2
addi    $sp, $sp, -4        # passage sur la pile
sw      $t0, 0($sp)
```

appeler la fonction,

```
jal     power
```

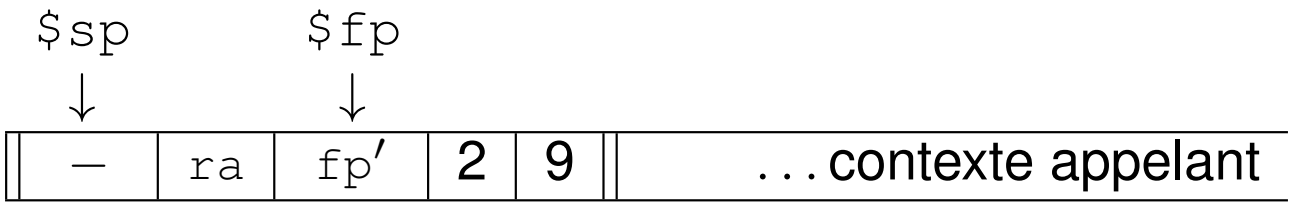
La fonction power prend alors le relais.
La pile contient déjà les deux arguments

Protocole d'appel à `power` (étape 2)

Construire le reste du tableau d'activation.

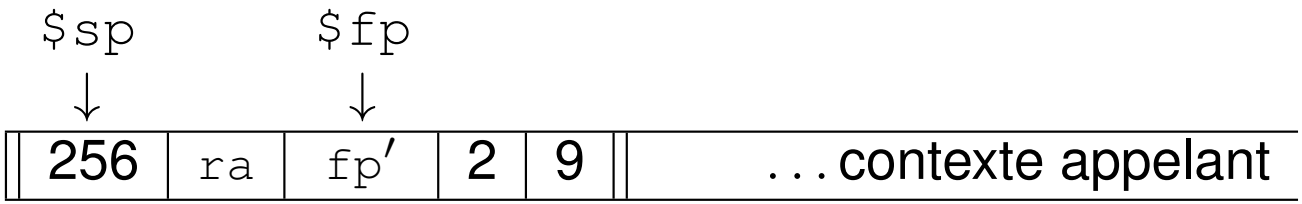
```
addi    $sp, $sp, -12       # allocation
sw      $fp, 8($sp)         # sauvegarde fp
sw      $ra, 4($sp)         # sauvegarde ra
addi    $fp, $sp, 8         # définition fp
```

déplacement de 12 octets (3 mots mémoire) : place pour la variable b et les deux registres sauvegardés.



Protocole d'appel à `power` (étape 3)

- le corps de la fonction s'exécute.
- après divers événements (appels récursifs) on a l'instruction return a*b;
- la variable locale b a la valeur 256 (power(4, 4)).



Protocole d'appel à `power` (étape 4)

On peut calculer le résultat a*b (512) et le placer dans $t0,

```
lw      $t0, 4($fp)      # t0 <- a
lw      $t1, -8($fp)     # t1 <- b
mul     $t0, $t0, $t1    # t0 <- a*b
```

restaurer $ra et $fp
libérer l'espace sur la pile,

```
lw      $ra, 4($sp)      # restauration ra
lw      $fp, 8($sp)      # restauration fp
addi    $sp, $sp, 12     # nettoyage de la pile
```

rendre la main à l'appelant.

```
jr      $ra
```

Protocole d'appel à `power` (étape 5)

- l'appelant reprend la main, la pile a retrouvé l'état d'avant l'appel.
- Les arguments sont encore en place, on les retire :

```
addi    $sp, $sp, 8         # nettoyage de la pile
```

- L'appelant peut utiliser le résultat de l'appel (registre $t0).

```
move    $a0, $t0
li      $v0, 1
syscall                     # affichage
```

Code de l'exponentiation, dans l'ordre

- Le protocole permet de gérer un nombre arbitraire d'appels de fonction imbriqués.
- Il s'applique aux appels récursifs de notre fonction power
- Le code assembleur de la fonction power commence à s'exécuter lorsque les arguments sont déjà placés sur la pile.
- Il commence par l'étape 2 du protocole d'appel (appelé, au début de l'appel), pour finir de mettre en place le tableau d'activation.

Code assembleur de l'exponentiation

```
    power:
    # Construction du tableau d'activation (ÉTAPE 2)
052     addi    $sp, $sp, -12
056     sw      $fp, 8($sp)
060     sw      $ra, 4($sp)
064     addi    $fp, $sp, 8
    # Exécution du corps de la fonction (ÉTAPE 3)
068     lw      $t0, 8($fp)     # test (n == 0)
072     bnez    $t0, power_rec
076     li      $t0, 1          # t0 <- résultat (return 1)
080     b       power_end
power_rec:
    # Déclenchement de l'appel récursif (ÉTAPE 1)
084     lw      $t0, 8($fp)     # push argument (n>>1)
088     sra     $t0, $t0, 1
092     addi    $sp, $sp, -4
096     sw      $t0, 0($sp)
100     lw      $t0, 4($fp)     # push argument (a*a)
104     mul     $t0, $t0, $t0
108     addi    $sp, $sp, -4
112     sw      $t0, 0($sp)
116     jal     power           # appel
```

Code assembleur de l'exponentiation (suite)

```
    # Après l'appel récursif (ÉTAPE 5)
120     addi    $sp, $sp, 8     # nettoyage arguments
    # Retour à l'exécution du corps de la fonction (ÉTAPE 3, suite)
124     sw      $t0, -8($fp)    # b <- résultat power(a*a, n>>1)
128     lw      $t0, 8($fp)     # test (n&1 == 0)
132     andi    $t0, $t0, 0x1
136     bnez    $t0, power_odd
140     lw      $t0, -8($fp)    # t0 <- résultat (return b)
144     b       power_end
    power_odd:
148     lw      $t0, 4($fp)
152     lw      $t1, -8($fp)
156     mul     $t0, $t0, $t1   # t0 <- résultat (return a*b)
    power_end:
    # Destruction du tableau d'activation, fin de l'appel (ÉTAPE 4)
160     lw      $ra, 4($sp)
164     lw      $fp, 8($sp)
168     addi    $sp, $sp, 12
172     jr      $ra
```

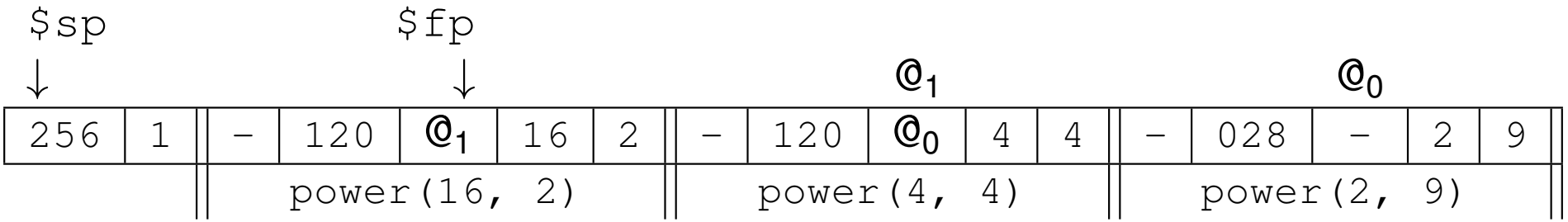
Code de l'instruction principale

Pour mémoire, voici également le code complet console.log(power(2, 9))

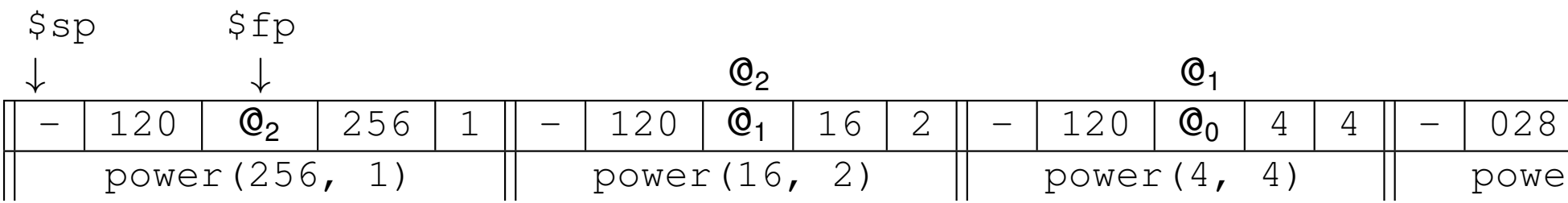
```
    # Appel power(2, 9)
000     li      $t0, 9              # préparation des arguments
004     addi    $sp, $sp, -4
008     sw      $t0, 0($sp)
012     li      $t0, 2
016     addi    $sp, $sp, -4
020     sw      $t0, 0($sp)
024     jal     power               # appel
028     addi    $sp, $sp, 8         # nettoyage des arguments
    # Affichage du résultat
032     move    $a0, $t0
036     li      $v0, 1              # code 1 : affichage d'un entier
040     syscall
    # Fin du programme
044     li      $v0, 10             # code 10 : arrêt de l'exécution
048     syscall
```

Suivi de la pile d'appels

power(2, 9) $\longrightarrow$ power(4, 4) $\longrightarrow$ power(16, 2) $\longrightarrow$ power(256, 1)
Au début de l'appel (ligne 116), la pile a la forme suivante :



Finalisation du TA (lignes 52–64)



À la fin de l'appel (lignes 160–168) retour au premier schéma.

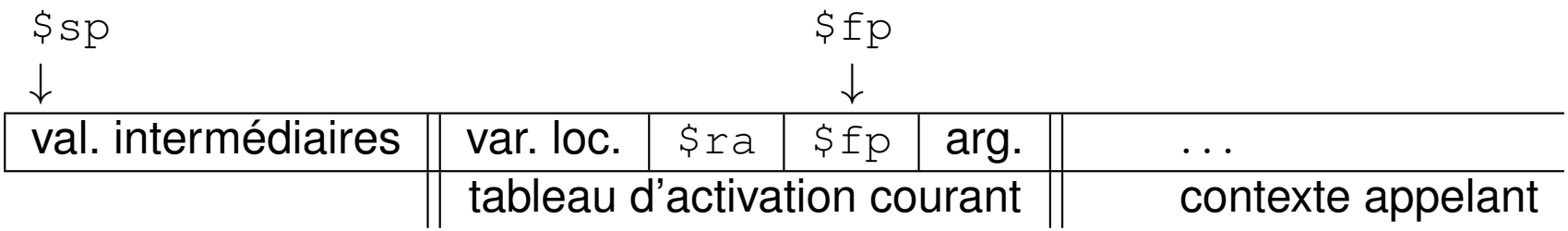
Partie 3 : Typage, fonctions, mémoire

1. Fonctions et pile d'appels
   - Appel de fonction : mécanisme de base
   - Pile d'appels
   - Convention d'appel
   - Génération de code pour ImpScript
   - Convention d'appel avec registres
   - Optimisation des appels terminaux

Introduction

On fixe les éléments suivants pour cette traduction :

- valeur calculée pour une expression : dans le registre $t0,
- valeur intermédiaire du calcul d'une expression : sur la pile,
- les valeurs des arguments d'une fonction : sur la pile,
- passage par valeur et pas par référence



Traduction d'une déclaration de fonction

Structure représentant une définition de fonction ImpScript :

```
type function_def = {
  name:   string;
  params: string list;
  locals: string list;
  body:   seq;
}
```

Initialisation d'une table qui associe à chaque variable locale et chaque argument sa position dans le tableau d'activation (par rapport à $fp).

```
let tr_function fdef =
  let env = Hashtbl.create 16 in
  List.iteri (fun k id -> Hashtbl.add env id (4*(k+1))) fdef.params;
  List.iteri (fun k id -> Hashtbl.add env id (-4*(k+2))) fdef.locals;
```

Les fonction tr_expr, tr_instr et tr_seq utilisent cette table.

Traduction d'une expression (cas de base)

Une variable peut être un paramètre, une variable locale ou une variable globale associée à une étiquette dans les données statiques.

```
let rec tr_expr = function
  | Int n -> li t0 n
  | Var x ->
    if Hashtbl.mem env x then
      let offset = Hashtbl.find env x in
      lw t0 offset(fp)
    else
      la t0 x @@ lw t0 0(t0)

    let op = match bop with
      | Mul -> mul     (*  *   *)
      | Asr -> sra     (*  >>  *)
      | And -> and_    (*  &   *)
      | Eq  -> seq     (*  ==  *)
    in
    tr_expr e1 @@ push t0 @@
    tr_expr e2 @@ pop t1 @@ op t0 t1 t0
```

Traduction d'une expression (appel)

Dans ImpScript, l'appel ne se fait pas sur un nom de fonction mais sur une expression qu'il faut évaluer

```
    | Call(f, params) ->
       tr_params params
      @@ tr_expr f
      @@ jalr t0
      @@ addi sp sp (4 * List.length params)

and tr_params = function
    | []          -> nop
    | e::params -> tr_params params @@ tr_expr e @@ push t0
in
```

Traduction des instructions

Voir première partie pour cas de base

```
let rec tr_instr = function
    | Set(x, e) ->
       tr_expr e
      @@ (if Hashtbl.mem env x then
            let offset = Hashtbl.find env x in
            sw t0 offset(fp)
          else
            la t1 x @@ sw t0 0(t1))

    | Return(e) ->
       tr_expr e
      @@ addi sp fp (-4)
      @@ pop ra @@ pop fp
      @@ jr ra
```

Traduction d'une déclaration de fonction

Le code est constitué du corps fdef.body traduit par tr_seq, entouré des étapes du protocole d'appel qui sont à la charge de l'appelé (étapes 2 et 4).

```
(* étape 2 *)
push fp @@ push ra
@@ addi fp sp 4
@@ addi sp sp (-4 * List.length fdef.locals)
(* exécution du corps de la fonction *)
@@ tr_seq fdef.body
(* étape 4 *)
@@ li t0 0          (* pas de return dans le corps *)
@@ addi sp fp (-4)
@@ pop ra @@ pop fp
@@ jr ra
```

Traduction d'un programme

```
type program = {
  globals:   string list;
  functions: function_def list;
  code:      seq;
}
```

il suffit alors de :

- générer du code pour chaque définition de fonction dans p.functions,
- générer le code principal pour p.code (fonction factice _main, appelée au début de l'exécution),
- déclarer (zone.data) les variables globales de p.globals,
- (pour ImpScript) déclarer une variable globale pour chaque fonction, dont la valeur initiale est l'adresse de la fonction.

Partie 3 : Typage, fonctions, mémoire

Passage des arguments

La convention d'appel standard de MIPS utilise les registres d'une manière plus fine que ce que nous avons fait dans la convention simplifiée.

- les quatre premiers arguments sont passés par les registres $a0 à $a3.
- les arguments suivants éventuels sont passés par la pile.
- le résultat, est renvoyé via le registre $v0.

protocole visible dans les appels systèmes

Usage des registres

- On a surtout utilisé les registres spéciaux $sp, $ra, $fp pour gérer la pile et les appels de fonction, plus $t0 et $t1 de manière temporaire pour les calculs.
- $ra et $fp devaient être sauvegardés chaque fois que leur valeur risquait d'être écrasée, et restaurés ensuite.
- le code assembleur peut utiliser l'ensemble des registres pour stocker des valeurs intermédiaires, ou des variables locales
- la convention d'appel doit alors régler quand et comment la valeur de chacun doit être sauvegardée/restaurée.

Sauvegarde des registres

Les registres sont séparés en deux paquets.

- Les registres callee-saved doivent être préservés par l'appelé.
  - Si l'appelé les utilise, il doit donc au préalable sauvegarder leurs valeurs, puis les restaurer avant de rendre la main.
  - En MIPS : registres $s* (Saved) et $fp.
- Les registres caller-saved peuvent être écrasés par l'appelé.
  - C'est à l'appelant de les sauvegarder avant l'appel et de les restaurer ensuite (s'il a encore besoin de leur valeur).
  - En MIPS : registres $a* (Argument), $t* (Temporary) et $ra (Return Address).

Impact sur le protocole

- étape 1 : l'appelant sauvegarde les valeurs des registres $a* ou $t* dont il aurait encore besoin après l'appel
- étape 5 : restaurer les registres sauvegardés
- étape 2 : l'appelé sauvegarde les valeurs des registres $s* qu'il va modifier.

Impact sur l'usage des registres

- on utilise les registres $t* en priorité pour des valeurs intermédiaires à la durée de vie courte :
  pas besoin de les sauvegarder avant de réaliser un appel.
- les éléments que l'on souhaite préserver sur une durée plus longue seront avantageusement stockés dans les registres $s*
  ils n'auront alors à être sauvegardés que lorsqu'une fonction appelée aura effectivement besoin d'utiliser ces mêmes registres.
- utilisation d'algorithmes de flots de contrôle pour déterminer la durée de vie des variables et choisir les registres

Exponentiation rapide, avec la convention MIPS.

- argument a dans $a0,
- argument n dans $a1,
- variable locale b dans $t2
  (pas d'appel entre la définition et l'utilisation de b)

Le tableau d'activation va maintenant contenir 4 cases :

- stocker $ra et $fp comme avant,
- préserver les arguments $a0 et $a1 en cas d'appel récursif.

Exponentiation rapide, code MIPS

Programme principal

```
    # Appel power(2, 9)
000'    li      $a1, 9          # préparation des arguments
012'    li      $a0, 2
024     jal     power           # appel
    # Affichage
032'    move    $a0, $v0
036     li      $v0, 1
040     syscall
    # Fin du programme
044     li      $v0, 10
048     syscall
```

Exponentiation rapide, code MIPS

Code de power(a, n)

```
    power:
    # Construction du tableau d'activation
052     addi    $sp, $sp, -16   # 4 mots réservés au lieu de 3
056     sw      $fp, 12($sp)    # sauvegarde de fp
060     sw      $ra, 8($sp)     # sauvegarde de ra
064     addi    $fp, $sp, 12    # positionnement de fp
    # Exécution du corps de la fonction
072'    bnez    $a1, power_rec  # test (n == 0)
076     li      $v0, 1          # v0 <- résultat (return 1)
080     b       power_end
```

Exponentiation rapide, code MIPS

```
    power_rec:
    # Appel récursif
084'    sw      $a0, -8($fp)    # sauvegarde a0 et a1
084''   sw      $a1, -12($fp)
088'    sra     $a1, $a1, 1     # prép. argument (n>>1)
104'    mul     $a0, $a0, $a0   # prép. argument (a*a)
116     jal     power           # appel
120'    lw      $a0, -8($fp)    # restauration a0 et a1
120''   lw      $a1, -12($fp)

    # Retour à l'exécution du corps de la fonction
124     move    $t2, $v0        # b <- résultat power(a*a, n>>1)
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136     bnez    $t0, power_odd
140     move    $v0, $t2        # v0 <- résultat (return b)
144     b       power_end
    power_odd:
    156'    mul     $v0, $t2, $a0   # v0 <- résultat (return a*b)
```

Exponentiation rapide, code MIPS

```
power_end:
    # Destruction du tableau d'activation
160     lw      $ra, 8($sp)
164     lw      $fp, 12($sp)
168     addi    $sp, $sp, 16
172     jr      $ra
```

Simplifications possibles :

- On pourrait économiser un registre et ces deux transferts en utilisant directement $v0 pour stocker la valeur de b.
- On peut ensuite réarranger les instructions pour éviter un branchement

```
    # Retour à l'exécution du corps de la fonction
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136'    beqz    $t0, power_end
156'    mul     $v0, $v0, $a0   # v0 <- résultat (return a*b)
    power_end:
```

Autres protocoles

- Les espaces pour les arguments et les variables locales disparaissent après l'appel (modifications perdues)
- Passage par référence :
  - l'argument est une expression qui possède une adresse (variable, champs d'une structure etc)
  - c'est l'adresse de cette expression qui est passée en argument à l'appelé
  - l'appelé peut modifier via cette adresse des valeurs à des adresses qui survivront à l'appel
  - dans le corps de l'appelé les instructions d'accès/mise à jour d'un argument passé par référence changent (indirection)
- Schéma lié à la nature des fonctions autorisées :
  - fonctions imbriquées : accès à des variables non locales/non globales
  - nécessiter de retrouver la valeur :
    - dans le TA où la variable a été allouée si il est encore sur la pile
    - dans une fermeture dans le cas fonctionnel général

Optimisation

réorganiser le code de power pour que n ne soit plus utile après l'appel récursif

```
function power(a, n) {
  if (n == 0) { return 1; }
  else { if (n&1 == 0) { return power(a*a, n>>1); }
         else          { return a*power(a*a, n>>1); } }
}
```

La sauvegarde et restitution de $a1 est inutile

Appel terminal : vocabulaire

- La fonction f fait un appel terminal à une fonction g si l'appel est la dernière instruction exécutée par f avant l'étape 4 (nettoyage et retour à l'appelant)
- Une fonction est récursive terminale si tous les appels récursifs sont des appels terminaux
- La même fonction peut être utilisée dans un appel terminal ou un appel non-terminal
- C'est l'appelant qui sait si l'appel est terminal ou non
- Un appel terminal permet d'éviter l'ajout d'un TA sur la pile
- Dans le cas de fonctions récursives on peut retrouver des exécutions en espace mémoire borné

Appel terminal

version de power dans laquelle tous les appels sont terminaux :

```
1   function power(a, n) {
2     return power_aux(a, n, 1);
3   }

4   function power_aux(a, n, acc) {
5     if (n == 0) { return acc; }
6     else {
7       if (n&1 == 0) { return power_aux(a*a, n>>1, acc); }
8       else          { return power_aux(a*a, n>>1, a*acc); }
9     }
10  }

11  console.log(power(2, 9));
```

Appel terminal

l'appel à power_aux ligne 2 est quasiment la dernière action de la fonction power :

- après cet appel, power a juste à nettoyer son propre tableau d'activation et sauter au contexte appelant, en transférant directement le résultat qui a été donné par power_aux.
- la fonction power peut demander à power_aux de transmettre directement son résultat au contexte appelant.
- Pour cela, on réalise l'appel à power_aux par un saut simple
  j power_aux
  au lieu de l'habituel saut
  jal power_aux.
  L'appel à power_aux aura comme adresse de retour la valeur courante de $ra, ie l'adresse à laquelle il fallait reprendre après l'appel à power.

Appel terminal

- power n'a plus besoin de son TA après l'appel à power_aux (sauf pour restaurer $fp)
- On peut restaurer $fp et nettoyer le TA avant de réaliser l'appel à power_aux.
- on peut réutiliser le tableau d'activation de power pour l'appel à power_aux sans modifier les valeurs sauvegardées de $ra et $fp.
- dans notre cas, on peut ne pas du tout construire de TA

Appels récursifs terminaux

- les deux appels récursifs à power_aux, aux lignes 7 et 8, sont terminaux.
- même simplification que celle que nous avons décrite pour power
- Code en passant l'argument acc de power_aux dans $a2,
  sans tableau d'activation superflu.

Code de power

- La fonction power réalise un unique appel terminal à power_aux, sans créér de TA.
- les deux premiers arguments sont déjà dans $a0 et $a1, il suffit de placer le troisième argument dans $a2 puis sauter à la fonction auxiliaire.

```
    power:
32      li      $a2, 1              # init. accumulateur
36      j       power_aux           # appel power_aux(a, n, 1)
```

Fonction power_aux

Pas besoin de TA

```
    power_aux:
40      beqz    $a1, power_end  # cas d'arrêt
44      andi    $t0, $a1, 0x1   # test (n == 0)
48      beqz    $t0, power_even
52      mul     $a2, $a0, $a2   # acc' <- a*acc
    power_even:
56      mul     $a0, $a0, $a0   # a' <- a*a
60      sra     $a1, $a1, 1     # n' <- n>>1
64      j       power_aux
    power_end:
68      move    $v0, $a2        # return acc
72      jr      $ra
```

Résultat

- code qui utilise les registres, pas la pile.
- résultat efficace : pas d'erreur « StackOverflow ».
- code assembleur équivalent à celui d'un programme avec une boucle.

```
function power(a, n) {
  let acc = 1;
  while (n != 0) {
    if (n&1 != 0) { acc = a*acc; }
    a = a*a;
    n = n>>1;
  }
  return acc;
}
```

- optimisation essentielle dans les langages utilisant massivement des fonctions récursives (comme caml),
- aussi dans GCC avec un niveau d'optimisation élevé
- certains langages (python, java) maintiènent une pile d'appels complète (débogage)

Synthèse

A retenir

- Le vocabulaire associé à l'appel de fonction
- Le rôle et la structure d'un tableau d'activation
- Le rôle des registres $ra, $fp, $sp
- L'allocation dynamique des TA sur la pile à l'exécution
- Les conventions d'appel
  - version minimale utilisant la pile
  - convention MIPS avec passage d'arguments dans les registres
  - optimisation des appels terminaux

Savoir faire

- Engendrer le code MIPS d'une fonction en respectant une convention