Génération de code-II
Données composées, appels de fonctions

13 octobre 2011

Données composées

- On peut vouloir représenter des données structurées (produits, structures, tableaux)
- Ces objets peuvent être manipulés comme expressions, passés en argument d'une fonction, renvoyés en résultat.
- Ils occupent une place de plus d'un mot en mémoire:
  - stockés sur la pile (structures en C).
  - une adresse d'un objet alloué dans le tas (ML, Java).
- Il faut préciser le mode de représentation des objets complexes.
- La taille de l'objet est-elle connue statiquement ?
  - La taille du tableau apparait-elle dans le type ?
  - La taille du tableau peut-elle être passée en argument d'une fonction ?
- Quand la taille du tableau n'est connue que dynamiquement, elle apparaît dans la représentation.

Sommaire

Types structurés

On suppose comme en Pascal des types tableaux où l'indice du début et de fin est donné.
Un tableau Caml, C ou Java de taille n est un tableau d'indice 0 à $n-1$.

```
type typ = Tbool | Tint
  | Tprod of typ * typ | Tarr of int * int * typ
```

A chaque type est associé la taille des données correspondantes:

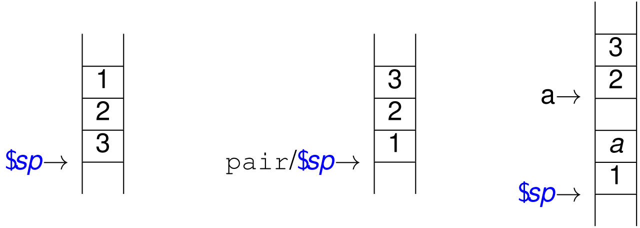
```
let rec size = function
    Tbool | Tint -> 1
  | Tprod(t1,t2) -> size t1 + size t2
  | Tarr(f,l,t) -> max 0 (l-f+1) * size t
```

Paires représentées par valeur dans la pile

- On ajoute dans le langage des expressions, la possibilité de former le couple de deux expressions, d'accéder à chacune des composantes d'un produit
$$E ::= (E_1, E_2) \mid \textbf{fst}\, E \mid \textbf{snd}\, E$$
- La valeur d'une expression composée ne tient plus dans un registre.
- On peut la stocker dans la pile.
- La valeur d'un object composé (e_1, e_2) peut être
  - un bloc formé de la valeur de e_1 et de la valeur de e_2;
  - l'adresse de l'emplacement mémoire à partir duquel on trouve les valeurs des composants (qui tient dans un registre).
  - Il faut choisir comment on positionne les composants et l'adresse par rapport aux composants.

Exemples de stockage de valeurs composées

$(1, (2, 3))$




```
.data
pair:
.word   1
.word   2
.word   3
```

Calcul de valeurs dans la pile

Schéma retenu

- Réserver la place dans un bloc sur la pile;
- Lorsque l'on veut manipuler un objet structuré on a besoin de connaître
  - l'adresse du bas du bloc dans lequel est stockée la valeur
  - la taille de la donnée (connue statiquement ou de manière dynamique)
  - Les décalages des composants (cas des structures à plusieurs champs).

Code pour les produits

- La fonction *codep* empile la valeur du résultat sur la pile
- une variable est associée à son adresse et sa taille.

$codep(E) = code(E) \mid \texttt{pushr}\ \$a0$ $\quad E$ expression arithmétique

$codep((E_1, E_2)) = codep(E_2) \mid codep(E_1)$
$codep(\mathbf{snd}_{p_1,p_2}(E)) = codep(E) \mid \mathbf{add}\ sp, sp, (4 \times p_1) \mid$

$codep(\mathbf{fst}_{p_1,p_2}(E)) =$ recopier les bonnes valeurs au bon endroit ...

$codep(\mathrm{id}_n) = \mathbf{la}\ \$a1\ \texttt{adr}(\mathrm{id}) \mid \mathbf{lw}\ \$a0\ (4 \times (n-1))(\$a1) \mid \texttt{pushr}\ \$a0$
$\quad \mid \ldots \mid \mathbf{lw}\ \$a0\ 0(\$a1) \mid \texttt{pushr}\ \$a0$

$codep(\mathrm{id}_n := E) = codep(E) \mid \mathbf{la}\ \$a1\ \texttt{adr}(\mathrm{id})$
$\quad \mid \texttt{popr}\ \$a0 \mid \mathbf{sw}\ \$a0\ 0(\$a1) \ldots$
$\quad \mid \texttt{popr}\ \$a0 \mid \mathbf{sw}\ \$a0\ (4 \times (n-1))(\$a1)$

Sommaire

Tableaux

- L'espace nécessaire pour stocker le tableau dépend du type de données (entier, réel, tableau, record) et du choix de représentation.
- On suppose la taille des données connue à la compilation.
- Les tableaux sont représentés par des suites de cases adjacentes.
- Un tableau multi-dimensionnel sera linéarisé en juxtaposant les lignes.
- Accès à des tableaux : décalages calculés à l'exécution.
  Utilisation d'arithmétique sur les adresses
- L'adresse à accéder dépend d'une adresse de base a et d'un décalage n calculé à l'exécution.

Code - Expressions de tableau

On ajoute à notre langage des tableaux que l'on suppose unidimensionnels.

$$
\begin{array}{ll}
E & ::= \mathrm{id}[E] \\
I & ::= \mathrm{id}[E_1] := E_2
\end{array}
$$

Cas simple

Tableau indicé à partir de 0 de taille n d'objets de taille 1.

- `adr`(id) adresse de base du tableau

Le code engendré est :

$$
\begin{array}{ll}
\mathit{code}(\mathrm{id}[E]) & = \mathit{code}(E) \mid \textbf{la}\ \$a1\ \texttt{adr}(\mathrm{id}) \\
 & \mid \textbf{add}\ \$a1,\ \$a1,\ \$a0 \mid \textbf{lw}\ \$a0,\ 0(\$a1) \\
\mathit{code}(\mathrm{id}[E_1] := E_2) & = \mathit{code}(E_1) \mid \textbf{la}\ \$a1\ \texttt{adr}(\mathrm{id}) \mid \textbf{add}\ \$a1,\ \$a1,\ \$a0 \\
 & \mid \texttt{pushr}\ \$a1 \mid \mathit{code}(E_2) \mid \texttt{popr}\ \$a1 \mid \textbf{sw}\ \$a0,\ 0(\$a1)
\end{array}
$$

Tableaux-Calculs d'indice

- Si le tableau t a des indices compris entre m et M, démarre à l'adresse a et contient des éléments de taille k alors il occupe une place de $(M - m + 1) \times k$.
- Pour calculer l'adresse correspondant à une expression $t[E]$, il faut calculer la valeur n de E puis accéder à l'adresse $a + (n - m) \times k$.
- Si on connaît la valeur de m à la compilation alors on peut partiellement évaluer cette expression en précalculant $a - m \times k$ et gardant cette valeur dans la table des symboles `base(t)`.
- Il suffira ensuite d'effectuer l'opération $\mathtt{base}(\mathtt{t}) + n \times k$.

Code - Expressions de tableau - cas général

Valeurs précalculées (tableau indicé entre m et M)

- `adr`(id) adresse de base du tableau
- k=`taille`(id) taille d'un élément du tableau
- `base`(id)=`adr`(id)-m×`taille`(id)

Le code engendré est :

$$\begin{array}{lcl}
\mathit{codep}(\mathrm{id}[E]) & = & \mathit{code}(E) \mid \mathbf{mul}\ \$a0,\ \$a0,\ k \mid \\
& & \mathbf{la}\ \$a1\ \texttt{base}(\mathrm{id}) \mid \mathbf{add}\ \$a1,\ \$a1,\ \$a0 \\
& & \mid \mathbf{lw}\ \$a0,\ (k-1)(\$a1) \mid \texttt{pushr}\ \$a0 \mid \ldots \\
& & \ldots \mid \mathbf{lw}\ \$a0,\ (\$a1) \mid \texttt{pushr}\ \$a0 \\
\mathit{code}(\mathrm{id}[E_1]{:=}E_2) & = & \mathit{code}(E_1) \mid \mathbf{mul}\ \$a0,\ \$a0,\ k \\
& & \mid \mathbf{la}\ \$a1\ \texttt{base}(\mathrm{id}) \mid \mathbf{add}\ \$a1,\ \$a1,\ \$a0 \mid \texttt{pushr}\ \$a1 \\
& & \mid \mathit{codep}(E_2) \mid \mathbf{lw}\ \$a1,\ (4\texttt{size}(E_2))\$sp \\
& & \mid \texttt{popr}\ \$a0 \mid \mathbf{sw}\ \$a0,\ 0(\$a1) \ldots \\
& & \ldots \mid \texttt{popr}\ \$a0 \mid \mathbf{sw}\ \$a0,\ (k-1)(\$a1) \mid \mathbf{addi}\ \$sp, \$sp, 4
\end{array}$$

Résumé

- Manipuler des données structurées, nécessite de réserver de l'espace en mémoire.
- Peut se faire sur la pile quand on controle la durée de vie, (calculs d'adresse à prendre en compte).
- Une solution plus uniforme est de manipuler les objets structurés comme des pointeurs dans le tas.
- Pour une programmation plus sûre, on ajoutera des tests que l'indice est dans les bornes du tableau.

Appels de fonctions

Introduction

- Compilation modulaire: code pour la fonction utilisant des paramètres formels, code de l'appel qui instancie ces paramètres.
- Différentes sémantiques pour le mode de liaison des paramètres
- Allocation dynamique de nouvelles variables (paramètres, variables locales)

Sommaire

Exemple

Allocation de la mémoire pour les paramètres d'une fonction récursive avec appel par valeur

```
int j;
void p(int i; int j) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}
main () {read(j); p(j,j);}
```

- Le nombre d'exécutions de *p* dépend de la valeur lue en entrée.
- À chaque entrée dans la procédure *p* deux nouvelles variables *i* et *j* sont allouées.
- A la sortie de la procédure, les emplacement mémoires sont libérés.

Exemple-exécution

```
p(i,j:int) {if (i+j>0) {j=j-1; p(i-2,j); p(j,i);}}
```

int j;
read(j); j=1
p(j,j) int i=j, j=j;
j=j-1;
p(i-2,j); int i=i-2,j=j;
p(j,i); int i=j,j=i;
j=j-1;
p(i-2,j); int i=i-2,j=j;
p(j,i) int i=j,j=i;

j	1
i	1
j	1 0
i	-1 0
j	0 1
i	-2 0
j	0 0

Sommaire

Adresses mémoires et valeurs

- Certaines expressions de programme représentent des adresses mémoires
  - les variables introduites dans le programme;
  - les cases d'un tableau;
  - les composants d'une structure;
  - les pointeurs.
- Dans ces cases mémoires, sont stockées des valeurs.
- Certaines expressions de programme peuvent désigner des cases mémoires ou bien la valeur qui y est stockée.
  - La valeur gauche d'une expression représente la case mémoire à partir de laquelle est stockée l'objet (utilisée dans les affectations)
  - La valeur droite de l'expression représente la valeur de l'objet stockée dans la case mémoire.
- Le passage de paramètres aux procédures peut se faire en transmettant les valeurs gauches ou les valeurs droites des objets manipulés.

Variables-Environnement et état

- un environnement lie des noms à des adresses mémoires : fonction partielle qui associe une adresse à un identificateur.
- un état lie des adresses à des valeurs : une fonction partielle qui associe une valeur à une adresse.
- Lors d'une affectation d'une valeur à une variable, seul l'état change.
- Lors de l'appel d'une procédure comportant des paramètres ou des variables locales, l'environnement change.
  La variable introduite correspond à une nouvelle liaison entre un nom et une adresse.

Procédures et fonctions

- Une procédure a un nom, des paramètres, une partie de déclarations de variables ou de procédures locales et un corps.
- Une fonction est une procédure qui renvoie une valeur.

- Les paramètres formels sont des variables locales à la procédure qui seront instanciées lors de l'appel de la procédure par les paramètres effectifs.
- La procédure peut déclarer des variables locales qui seront initialisées dans le corps de la procédure.

Passage de paramètres

- Les paramètres formels de la procédure sont des variables qui sont initialisées lors de l'appel de la procédure.
- Il y a plusieurs manières d'effectuer cette initialisation.
- On suppose que l'on a une procédure p avec un paramètre formel x qui est appelée avec le paramètre effectif e.
- On examine différents mode de passage de ce paramètre.

Passage par valeur

Dans le passage de paramètre par valeur, x est une nouvelle variable allouée localement par la procédure dont la valeur est le résultat de l'évaluation de e.

- Après la fin de la procédure, la place mémoire allouée à la variable x est libérée.
- Les modifications apportées à x ne sont plus visibles.
- En l'absence de pointeurs, les seules variables modifiées sont les variables non locales à la procédure explicitement nommées dans les instructions du programme.
- Il est nécessaire de réserver une place proportionnelle à la taille du paramètre ce qui peut être coûteux dans le cas de tableaux.

Passage par référence ou par adresse

On calcule la valeur gauche de l'expression *e*
(si *e* n'a pas de valeur gauche on crée une variable que l'on initialise à la valeur droite de *e* et on utilise la valeur gauche de cette variable).

- La procédure alloue une variable *x* qui est initialisée par la valeur gauche de *e*.
- Toute référence à *x* dans le corps de la procédure est interprétée comme une opération sur l'objet situé à l'adresse stockée en *x*.
- Ce mode de passage occupe une place indépendante de la taille du paramètre (une adresse).

Remarques:

- En C, le passage par référence est explicitement programmé par le passage d'un pointeur (adresse mémoire) par valeur.
- En Java, le passage se fait par valeur mais les objets ont pour valeur une référence.

Passage par nom

- Remplacement textuel dans le corps de la procédure des paramètres formels par les paramètres effectifs.
- Mécanisme de macro qui peut provoquer des problèmes de capture.

Exemple

```
swap(int x; int y) {int z; z=x; x=y; y=z;}
```

Si z et t sont des variables globales du programme `swap`(z, t)

```
{int z; z=z; z=t; t=z;}
```

Renommage

- Renommer les variables locales.

```
swap(int x; int y) {int zz; zz=x; x=y; y=zz;}
```

- Cela ne suffit pas par exemple l'appel par nom de `swap`(i,a[i]) ne donne pas le résultat attendu.

```
{int z; z=i; i=a[i]; a[i]=z;}
```

- Méthode utile lors de la compilation de procédures de petite taille (coût de la gestion de l'appel est important).

Passage par copy-restore

- Lors de l'appel de la procédure la valeur droite de *e* sert à initialiser la variable *x*.
- À la sortie de la procédure la valeur droite de *x* sert à mettre à jour la valeur gauche de *e*.
- Des comportements parfois différents du passage par référence.

```
int a;
p (int x) {x=2; a=0;}
main () {a=1;p(a);write(a);}
```

Equivalent en appel par référence à

```
{a=1; a=2; a=0; write(a);}
```

Equivalent en copy-restore à

```
{a=1; {int x=a; x=2; a=0; a=x}; write(a);}
```

Évaluation stricte/paresseuse

Dans les langages fonctionnels, on distingue les langages stricts des langages dits paresseux.

- Langage strict : les valeurs des arguments sont calculées avant d'être passées en paramètre à une fonction (CAML ou SML).
- Langage paresseux : l'expression passée en argument à une fonction f sera évaluée seulement si f a besoin de cette valeur (on parle d'appel par nécessité, Haskell).
- L'évaluation paresseuse se combine mal avec les effets de bord (difficulté de controler à quel moment l'expression sera évaluée).

Évaluation paresseuse

- Définition $f(x) = t$, on veut calculer $f(e)$
- Lorsque l'évaluation de t nécessitera la valeur de x, le calcul de e sera effectué.
- Si t utilise plusieurs fois x, le calcul sera effectué une seule fois.
- L'expression e vient avec son environnement (les valeurs des variables utilisées par e) ce qui évite les problèmes de capture.
- Mécanisme différent de celui de remplacement textuel (macros).

Sommaire

Compilation des fonctions et procédures

- Déclaration $f(x) = e$
- Appel $f(a)$ correspond à $\textbf{let}\ x = a\ \textbf{in}\ e$
- Compiler le code de e en faisant une hypothèse sur l'emplacement de x
- Placer a à l'emplacement souhaité
- Sémantique de $x = a$
  - valeur de a
  - code de a
  - emplacement mémoire de a
  - ...

Procédures et fonctions-suite

- On ne contrôle pas le nombre d'appels d'une procédure ni le moment où celle-ci sera appelée.
- Il n'est pas possible de donner statiquement les adresses des variables apparaissant dans la procédure.
- Ces adresses pourront être calculées relativement à l'état de la pile au moment de l'appel de la procédure.
- L'espace mémoire alloué pour les paramètres et les variables locales est libéré lorsque l'on sort de la procédure.

Organisation de la mémoire

Dans un appel de procédure ou de fonction, la mémoire est organisée en tableaux d'activation.

- Le tableau d'activation est une portion de la mémoire qui est allouée à l'appel de la procédure et libérée au retour.
- Il contient les informations nécessaires à l'exécution du programme (paramètres, variables locales).
- Il sauvegarde les données qui devront être restaurées à la sortie de la procédure.

Afin de faciliter la communication entre des codes compilés à partir de langages distincts (par exemple pour appeler des procédures de bas niveau à partir d'un langage de haut niveau), il n'est pas rare qu'un format de tableau d'activation pour une architecture donnée soit particulièrement recommandé.

Données à conserver

Lors d'un appel de procédure:

- le contrôle du code est modifié:
  Retour normal de la procédure (pas d'échappement en erreur ou d'instruction de type `goto`) : la suite de l'exécution doit se poursuivre à l'instruction suivant l'instruction de l'appel.
- Le compteur d'instruction doit donc être sauvegardé à chaque appel de procédure.
- Les données locales à la procédure s'organisent dans la pile à partir d'une adresse sur le bloc d'activation (appelée framepointer) qui est déterminée à l'exécution et sauvée dans un registre (noté *fp*).
- Lorsqu'une nouvelle procédure est appelée, cette valeur change, il peut être nécessaire de sauvegarder la valeur courante qui pourra être restaurée à la fin de la procédure.

Appelant-Appelé

- Les opérations à effectuer lors d'un appel de procédure se partagent entre l'appelant et l'appelé.
- Il faut décider si les paramètres et la valeur de retour seront stockés dans la pile ou dans des registres.
- Le code géré par l'appelant doit être écrit pour chaque appel tandis que celui écrit dans l'appelé n'apparait qu'une seule fois.
- L'appelant effectue si nécessaire la réservation pour la valeur de retour dans le cas de fonctions et évalue les paramètres effectifs de la procédure, les place dans la pile ou les registres appropriés.
- L'appelé initialise ses données locales et commence l'exécution.
- Au moment du retour, l'appelé place éventuellement le résultat de l'évaluation à l'endroit réservé par l'appelant et restaure les registres.

L'appelant et l'appelé doivent avoir une vision cohérente de l'organisation de la mémoire

Sommaire

Sous-routines

- Réutiliser à plusieurs endroits la même séquence de code *I*.
- On isole cette partie du code et on lui donne une étiquette.
- Au moment de l'appel il faut sauvegarder le point de retour dans un registre dédié *$ra* (instruction **jal**).
- A la fin du code de la routine, on effectue un saut au point de code sauvegardé.
- Si le corps d'une routine en appelle une autre, le registre *$ra* doit être préservé.

Sous-routine, code

Nous ajoutons des définitions et appels de routines dans notre langage :

$$\begin{array}{lcl} D & ::= & \textbf{proc}\ p;\ \textbf{begin}\ I\ \textbf{end} \\ I & ::= & \textbf{call}\ p \end{array}$$

label-p est une étiquette unique associée à la procédure *p*.

$$\begin{array}{lcl} \mathit{code}(\textbf{proc}\ p;\ \textbf{begin}\ I\ \textbf{end}) & = & \texttt{label-p: pushr}\ \$ra \\ & & \mid \mathit{code}(I) \mid \texttt{popr}\ \$ra \mid \textbf{jr}\ \$ra \\ \mathit{code}(\textbf{call}\ p) & = & \textbf{jal}\ \texttt{label-p} \end{array}$$

Procédures avec paramètres

- Si la procédure a des paramètres alors elle alloue dans la pile la place pour des variables.
- Un registre *fp* peut être positionné au début de l'appel de procédure pour référencer les valeurs locales.
- A la sortie de la procédure l'espace est libéré.

Conventions d'appel (MIPS)

- Les 4 premiers arguments peuvent être passés dans les registres $\$a0$, $\$a1$, $\$a2$, $\$a3$, les autres dans la pile.
- Les registres $\$v0$ et $\$v1$ sont utilisés pour le retour de la fonction.
- Le registre $\$ra$ est utilisé pour l'adresse de retour de la fonction.
- Les registres $\$t_i$ et $\$s_i$ peuvent être utilisés pour des calculs temporaires.
- Un nombre quelconque d'appels de fonctions peut être actif simultanément : les valeurs des registres doivent donc être sauvegardées pour des utilisations futures.
  Il faut toujours sauvegarder $\$ra$ (adresse de retour de la fonction) et si on l'utilise $\$fp$ (adresse du tableau d'activation).
- Certains registres ($\$ti$) sont par convention sauvegardés par l'appelant (caller-save) d'autres par l'appelé (callee-save). Les registres $\$si$ ainsi que $\$ra$ et $\$fp$ sont sauvegardés par l'appelé.

Organisation du tableau d'activation

- Le tableau d'activation contient les paramètres de la fonction (sauf éventuellement les 4 premiers qui peuvent être placés dans les registres *$a0*,… *$a3*), les variables locales, il doit également sauvegarder *$ra* et *$fp*.
- La commande **jal** *label* fait un saut au code à l'adresse *label* et préserve l'adresse de retour dans le registre *$ra*.
- La commande **jr** *$ra* permet de retourner à l'instruction suivant l'appel après avoir restauré les registres nécessaires.
- Le registre de frame (*$fp*) est positionné à un endroit fixe dans le tableau et permet d'accéder au données par un décalage fixe indépendamment de l'état de la pile.

Tableau d'activation

- Paramètres de la fonction : $e_1, \ldots, e_n$
- Variables locales ou registres sauvegardés : $v_1, \ldots, v_m$

	e_1	ou e_5
	$\vdots$	construit par l'appelant
	e_n	
	$fp appelant	
$fp $\rightarrow$	*$ra*	construit par l'appelé
	v_1	
	$\vdots$	
	v_m	
	$\vdots$	pile croissante
	$\downarrow$	adresses décroissantes

Protocole d'appel

L'appelant:

- Sauvegarde les registres dont il a la responsabilité et dont il aura besoin après l'appel.
- Evalue les arguments formels e_i dans les registres et/ou la pile.
- Saute à l'instruction correspondant au label de la fonction en sauvegardant le point de retour dans *$ra* (instruction **jal**).
- Restaure les registres sauvegardés.
- Dépile les arguments formels.

Protocole d'appel

L'appelé:

- Réserve l'espace de pile utile pour la procédure :
  les valeurs $v_1, \dots, v_m$ sont utilisées pour des registres à sauvegarder ou des variables locales.
- Sauvegarde la valeur du registre *$fp* de l'appelant.
- Sauvegarde sa propre valeur de retour (le registre *$ra* peut être modifié lors d'un appel).
- Positionne le registre *$fp* dans le tableau d'activation.
- Sauvegarde d'éventuels autres registres dont il est responsable.
- Exécute les instructions du corps de la fonction.
  Place la valeur de retour dans le registre *$v0* ou à l'emplacement prévu dans la pile
- Restaure le registre *$ra* et les autres registres dont il est responsable.
- Restaure le registre *$fp* de l'appelant.
- Dépile l'espace qui avait été alloué pour le tableau d'activation.
- Saute à l'instruction du registre *$ra* à l'aide de la commande **jr**.

Exemple

Bien définir le tableau d'activation; suivre à la lettre le protocole.

```
let rec fact n = if n <= 0 then 1 else n * fact (n-1)
```

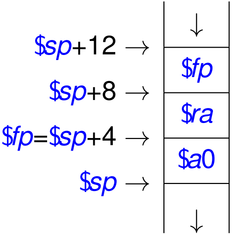
Compilation informelle de `fact`:

```
 fact: save($ra,$fp)
  if ($a0 ≤ 0) goto then
  save($a0)
  $a0 := $a0-1
  call fact
  restore($a0)
  $v0 := $a0 * $v0
  goto end
then: $v0 := 1
end:  restore($ra,$fp)
  goto $ra
```

La valeur de $a0 doit être sauvegardée et restaurée

Tableau d'activation de fact

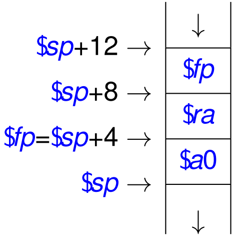
L'argument est passé dans le registre $\$a0$ et la valeur de retour dans $\$v0$.



Code assembleur

```
fact: addiu $sp,$sp,-12
      sw $fp,8($sp)
      sw $ra,4($sp)
      addiu $fp,$sp,4
      blez $a0 then
      sw $a0,0($sp)
      addiu $a0,$a0,-1
      jal fact
      lw $a0,0($sp)
      mul $v0,$a0,$v0
      j end
then: li $v0 1
end:  lw $ra,4($sp)
      lw $fp,8($sp)
      addiu $sp,$sp,12
      jr $ra
```

$a0 contient *n* (sauvegardé)
La valeur de retour est dans *$v0*



Syntaxe du langage avec fonctions

Une syntaxe à la C

$$
\begin{array}{lll}
\mathit{Vs} & ::= & \mathit{Vs}\ V;\ \mid \epsilon \\
V & ::= & T\ \mathrm{id} \\
D & ::= & \mathrm{id}(\mathit{Vs})\ \{\mathit{Vs}\ I\ \} \\
 & \mid & T\ \mathrm{id}(\mathit{Vs})\ \{\mathit{Vs}\ I\ \textbf{return}\ E;\}
\end{array}
$$

Schéma général de l'appel de fonction (par valeur)

$code(f(e_1, \dots, e_n)) =$

```
code(e_1) [ | pushr $a0]
 |  code(e_2)  |  [pushr $a0 ou move $a_1,$a_0]
..
 |  code(e_n)  |  pushr $a0
sauvegarde des registres caller-saved
 |  jal f
restaure les registres caller-saved
 |  addiu $sp,$sp,4×nparams(f)
```

Déclaration de fonction

$code(T\ f(T_1\ x_1; \ldots T_n\ x_n)\{U_1\ z_1; \ldots U_p\ z_p\ I$ **return** $E\}) =$

pushr $\$fp$ |
pushr $\$ra$ |
move $\$fp$, $\$sp$ |
addiu $\$sp$, $\$sp$, $-4 \times$ *nvars*(f) |
sauvegarde des registres callee-saved
code(I) |
code(E) |
move $\$v0$, $\$a0$ |
restaure les registres callee-saved
addiu $\$sp$, $\$sp$, $4 \times$ *nvars*(f) |
popr $\$ra$ |
popr $\$fp$ |
jr $\$ra$

Organisation du tableau d'activation

Pour chaque fonction ou procédure *f*, on peut calculer statiquement:

- `nretour`(f) la taille de la valeur de retour
- `nparams`(f) la taille des paramètres
- `nvars`(f) la taille des variables locales

Pour chaque variable *x*, on stocke :

- `decal`(x) entier indiquant la position relative par rapport à *fp* où est stockée la variable.
  - les paramètres sont à des adresses + grandes que *fp* (décalage positif)
  - les variables locales sont à des adresses plus petites (décalage négatif).
- `taille`(x) si les variables peuvent avoir une taille plus grande que 1
- Mode de passage de *x* si possibilité de passage par référence

Sommaire

Passage par référence

- Dans le cas où une variable est passée par référence, c'est son adresse qui est stockée dans le tableau d'activation (ou les registres).
- Les fonctions d'accès et de mise à jour contiennent une indirection.

Exemple

```
f (ref int x; int y;) {y:=x+y; x:=x*y; }
int u=3;
main(){f(u,u);print(u)}
```

Organisation de la mémoire:

- Résultat ?
- Fonction *f*:
  - La variable *x* peut être passée dans le registre $a0 et *y* dans le registre $a1.
  - Les registres *$fp* et *$ra* ne seront pas écrasés.
- Fonction *main*:
  - le registre *$ra* doit être sauvegardé (appel de *f*)

Exemple-code assembleur

```
f (ref int x; int y;) {y:=x+y; x:=x*y; }
main(){f(u,u);print(u)}
```

```
     .data
 u:    .word 3
     .text
 f:  lw $a2,0($a0)
     add $a1,$a2,$a1
     mul $a2,$a2,$a1
     sw $a2, 0($a0)
     jr $ra
```

```
main:
   add $sp,$sp,-4
   sw $ra,0($sp)
   la $a0,u
   lw $a1,u
   jal f
   lw $a0,u
   li $v0,1
   syscall
   lw $ra,0($sp)
   add $sp,$sp,4
   jr $ra
```

Calcul de la valeur gauche d'une expression

Valeur gauche: adresse où est stockée l'expression
Seules certaines expressions peuvent être des valeurs gauches: (variables, tableaux)
codeg(*\$r*,e) met la valeur gauche de *e* dans le registre de *\$r*

$$
\begin{array}{ll}
\mathit{codeg}(\$r, x) = \textbf{la}\ \$r, \texttt{adr}(x) & \text{si } \neg\, \texttt{islocal}(x) \\
\mathit{codeg}(\$r, x) = \textbf{add}\ \$r, \$fp, \texttt{decal}(x) & \text{si } \texttt{islocal}(x) \\
\mathit{codeg}(\$r, x[E]) = \mathit{code}(E) \mid \texttt{pushr}\ \$a0 & \\
\quad \mid \mathit{codeg}(\$r, x) \mid \texttt{popr}\ \$a0 \mid \textbf{add}\ \$r, \$r, \$a0 &
\end{array}
$$

Si une variable doit être passée par référence, il faut l'allouer sur la pile et pas dans un registre.

Code pour les expressions

$$
\begin{array}{ll}
\mathit{code}(x) = \textbf{lw}\ \$a0, \texttt{adr}(x) & \text{si } \neg\ \texttt{islocal}(\text{x}) \\
\mathit{code}(x) = \textbf{lw}\ \$a0, \texttt{decal}(x)(\$fp) & \text{si } \texttt{islocal}(\text{x})\text{, par valeur} \\
\mathit{code}(x) = \textbf{lw}\ \$a0, \texttt{decal}(x)(\$fp) \mid \textbf{lw}\ \$a0, 0(\$a0) & \text{si } \texttt{islocal}(\text{x})\text{, par référence}
\end{array}
$$

Code appelant

$f(e_1, \ldots, e_2)$

remplacer $\mathit{code}(e_i)$ par $\mathit{codeg}(e_i)$ si le i-ème argument est passé par référence.

Sommaire

Fonctions récursives

- Chaque appel de fonction crée de nouvelles variables
- Dans le cas de fonctions (mutuellement) récursives,
  - les registres sont insuffisants pour stocker les variables;
  - un tableau d'activation doit être réservé sur la pile;
  - plusieurs tableaux d'activation de la même fonction coexistent simultanément.
- Le nombre de tableaux d'activations empilés peut dépendre des valeurs des paramètres et n'est donc pas connu à la compilation.
- La récursion stocke implicitement des valeurs intermédiaires et peut simplifier la programmation (exemple du backtracking).
- La récursion dite terminale est un cas particulier qui peut être compilé de manière efficace.

Exemples

```
let rec hanoi i j k n =
    if n > 0 then begin
      hanoi i k j (n-1);
      Printf.printf "%d -> %d \n" i k;
      hanoi j i k (n-1)
    end
```

Limites:

```
let rec fact n = if n <= 0 then 1 else n * fact (n-1)
```

```
# let _ = fact 1000000;;
Stack overflow during evaluation (looping recursion?).
```

Version récursive terminale:

```
let rec factt k n =
   if n <= 0 then k else factt (n * k) (n-1)
let fact = factt 1
```

Récursion terminale

- On suppose que la fonction $f(x, y)$ fait un appel à $f(t, u)$
- L'appel est terminal s'il n'y a pas de calcul en attente au moment de l'appel récursif:
  - Les valeurs de x et y ne seront pas réutilisés après le calcul de $f(t, u)$.
  - Le résultat de $f(t, u)$ est également le résultat attendu pour $f(x, y)$.
- Le tableau d'activation et les registres de l'appel de $f(x, y)$ peut être réutilisé pour l'appel $f(t, u)$.
- Attention au moment de l'affectation $(x, y) \leftarrow (t, u)$ à sauvegarder si nécessaire la valeur de x.
- L'appel récursif dans le corps de la fonction devient un simple saut.
- Une fonction récursive terminale est compilée aussi efficacement qu'une boucle.

Exemple

Organisation de la mémoire de `factt` :

- l'argument k est dans le registre $\$a0$ et n dans le registre $\$a1$.
- la valeur retournée est dans le registre $\$v0$
- le registre $\$ra$ n'a pas besoin d'être sauvegardé (pas d'appel dans le corps)

```
fact:  blez $a1 out
       mul $a0,$a0,$a1
       addi $a1,$a1,-1
       j fact
out: move $v0,$a0
jr $ra
```

Sommaire

Fonctions emboitées

- On a considéré le cas où les variables étaient soit globales, soient locales à la fonction.
- Dans un langage comme Pascal ou CAML (contrairement à C ou Java), il est possible de définir des fonctions dans d'autres fonctions.
  Les fonctions peuvent être imbriquées.
- Une fonction peut alors accéder à:
  - des variables globales : allouées à un emplacement fixe;
  - ses paramètres et variables locales : allouées dans son propre tableau d'activation, registre;
  - les paramètres ou variables locales d'une des procédures dans laquelle elle est définie et qui sont allouées dans le tableau d'activation de cette procédure.

Exemple

```
let main a =
  let e = read_float () in
  let iter z =
    let stop x = abs (a - x * x) < e in
    let next () = (z + (a / z)) / 2 in
    if stop z then z else
    let z' = next () in iter z'
  in iter 1
```

Execution de `main(2)`

Organisation de la mémoire

- Une fonction/procédure peut accéder aux variables des fonctions/procédures dans lesquelles elle est déclarée.
- Où sont stockées ces variables en mémoire ?
- Lorsqu'une fonction est appelée, toutes les fonctions parentes sont actives sur la pile, leurs variables locales sont dans les tableaux d'activation respectifs.
- Les tableaux d'activation sont chaînés pour retrouver l'enchainement statique des fonctions.

Exemple d'arbre de niveau de déclarations

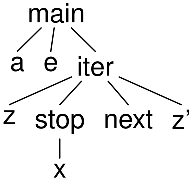
Niveaux

0 : main

1 : a, e, `iter`

2 : z, `stop`, `next`, z'

3 : x

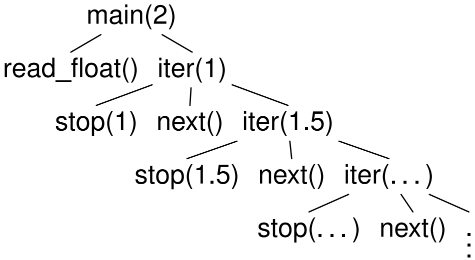


Arbre d'activation

Arbre d'activation

- Représente les exécutions possibles d'un programme.
- Les nœuds de cet arbre représentent des appels de procédures $p(e_1, \ldots, e_n)$.
- Un nœud a k fils $q_1, \ldots q_k$ si l'exécution du corps de la procédure donne lieu directement aux appels de procédures $q_1, \ldots q_k$
(ces appels pouvant eux-mêmes engendrer de nouveaux appels de procédures).
- La racine de l'arbre est l'exécution du programme principal.

Sur notre exemple, on a :



Accès aux variables

- Lors d'une exécution du code on dira qu'une procédure est active si on n'a pas encore fini d'exécuter le corps de cette procédure.
- Toutes les procédures actives ont leur tableau d'activation réservé en mémoire qui contient leurs variables locales.
- Si une procédure active mentionne une variable y celle-ci a été déclarée localement par un de ses ancêtres (peut-être lui-même).
- Le degré de parenté de cet ancêtre est connu à la compilation (différence de niveaux).
- En chainant chaque tableau d'activation d'une procédure p avec le tableau d'activation de son père statique, on retrouve simplement en suivant le bon nombre d'indirections l'endroit où la variable concernée est stockée.

Chaînage des tableaux d'activations

- On doit garder dans le tableau d'activation d'une procédure, l'adresse du tableau d'activation de son père.
- Cette adresse peut être stockée après le calcul des paramètres effectifs, avant l'appel : à l'endroit de $\$fp + 8$.

	e_i	arguments
	f_s	$\$fp$ du père statique (accès variable)
	f_d	$\$fp$ du père dynamique (sauvegarde)
	r	$\$ra$ point de retour (sauvegarde)
$\$fp \rightarrow$	v_j	variables locales

Accès à l'adresse du tableau d'un ancètre de i-ème génération:

move $a0 $fp | **lw** $a0 8($a0) | … (i fois) | **lw** $a0 8($a0)

Cas des langages fonctionnels

- Dans un langage comme Ocaml, les fonctions sont imbriquées mais peuvent aussi être retournées comme valeur.
- Les variables utiles au calcul de la fonction qui ne sont pas définies dans la fonction ne seront pas forcément présentes dans la pile au moment de l'exécution.
- Elles seront sauvegardées dans une clôture avec le code de la fonction dans le tas.