Lien statiques, Frames et Représentation intermédiaire

- Liens statiques
  - les blocs d'activation
  - la difficulté avec la portée statique
  - une solution: les attributs *level* et *offset*
- Analyse d'échappement
- Représentation intermédiaire
  - arbres de commandes et expression: définition
  - exemples de traduction

Exécution des fonctions

L'exécution d'un programme CTigre qui comporte des fonctions peut se faire en utilisant la *pile de blocs d'activation* que nous avons utilisé pour l'éxécution de fonctions en assembleur.

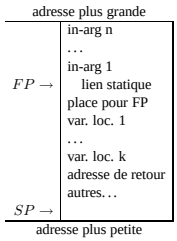
On reviendra plus avant sur les caractéristiques de CTigre qui font en sorte qu'une telle pile est suffisante. Pour l'instant il suffira de remarquer que ce n'est pas toujours le cas (notamment, Scheme et Ocaml ne peuvent se satisfaire d'une machine à pile).

Conventions...

Rappel, dans ce cours nous assumons que:

- la pile croît vers le bas,
- SP pointe sur la dernière case utilisée de la pile
- l'organisation du bloc d'activation est la suivante

(le pourquoi sera plus clair avant):



Appel d'une fonction, création d'un bloc sur la pile

Voyons à nouveau comment peut se dérouler un appel de fonction f[1]
Il y a une partie du travail qui est fait par l'appelant:

- l'appelant mets en place les m paramètres actuels de la fonction appelée f (ce sont bien des "empilements", avec SP qui décroît...)
- l'appelant empile le lien statique ls de f ($SP \leftarrow SP-1; M[SP] \leftarrow ls$, voir suite)
- l'appelant appelle la fonction f (instruction assembleur `CALL f`)

Et une partie du travail qui est fait par l'appelé...

Appel d'une fonction, création d'un bloc sur la pile

prologue f "empile son bloc" sur la pile

- l'appelé, f, sauvegarde la valeur de FP...

[1] en supposant que chaque paramètre et variable occupe exactement une case mémoire.


```
    SP <- SP-1
M[SP] <- FP
```

et *alloue son bloc* (de taille K)

```
FP <- SP+1
SP <- SP-K+1
```

- f a reçu par l'appelant dans un registre spécial ret l'adresse de retour, elle peut le sauver dans son frame, si nécessaire.

calcul
on exécute le corps de f, le résultat est dans un registre spécial res

épilogue f *désalloue son bloc* et restaure les valeurs de SP et FP, ...

```
SP <- FP-1
FP <- M[SP]
SP <- SP+1
```

et saute à l'adresse de retour[2]:

```
JUMP ret
```

Un exemple

Considérons le programme suivant

```
let g(x:int): int =
    let a := 50 in
    let f(y:int,z:int):int = y*y+z
    in f(x,a)+a
in g(3)
```

et ignorons pour l'instant le lien statique

Évolution de SP et FP

A l'exécution, on instancie les variables locales des fonctions exécutées et l'évolution de la pile suit le niveau d'imbrication des fonctions.

$FP \rightarrow 1100$	var. globales
	...
$SP \rightarrow 1002$	$dern.\ case$

[2] eventuellement restaurée

empile params g

$FP \rightarrow 1100$	var. globales
	...
1002	
$SP \rightarrow 1001$	$x = 3$

appel de g

1100	var. globales
	...
1002	
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	1100 (vieux FP)
$SP \rightarrow 998$	$a = 50$
997	

Évolution de SP et FP

g empile les args. de f

1100	var. globales
	...
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	1100 (vieux FP)
998	$a = 50$
997	$y = 3$
$SP \rightarrow 996$	$z = 50$

appel de f dans g

1100	var. globales
	...
1001	$x = 3$
1000	lien statique
999	$1100(vieux FP)$
998	$a = 50$
997	$y = 3$
996	$z = 50$
$FP \rightarrow 995$	lien statique
$SP \rightarrow 994$	1000 (vieux FP)

Évolution de SP et FP

f retourne

1100	var. globales
	...
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	1100 (vieux FP)
998	$a = 50$
997	$y = 3$
$SP \rightarrow 996$	$z = 50$

g dépile params de f

1100	var. globales
	...
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	1100 (vieux FP)
$SP \rightarrow 998$	$a = 50$

L'attribut *offset*

Pour pouvoir accéder à ses propres variables, le code machine produit par la fonction devra connaître la *position dans son propre bloc d'activation* de ces variables.

Pour cela il est important d'attacher à chaque variable locale un attribut, traditionnellement appelé *offset*, qui donne cette position et qui sera utilisé pour générer le code

qui accédera à cette variable.

Si on assume[3] que toutes les variables locales sont mémorisée dans le bloc d'activation, avec la convention que l'on a fixé, cette valeur peut être calculée en suivant l'ordre des déclarations des variables locales: offset vaudra 2 pour la première déclaration, 3 pour la deuxième etc. (les positions 0 et 1 sont prises par le lien statique et FP).

Pour les paramètres formels, qui se trouvent positionnés de l'autre coté de FP, on peut choisir un offset négatif.

Le problème de la portée statique

La notion de bloc, avec portée statique, implique qu'une fonction définie localement à un bloc peut avoir accès à toutes les définitions du bloc englobant, et de celui qui englobe celui-ci, etc.

Considérons l'exemple suivant (qui calcule le même résultat que le précédent)

```
let g(x:int): int =
    let a := 50
    in let f(y:int):int = y*y+a
       in f(x)+a
in g(3)
```

A l'exécution, f aura besoin d'accéder aussi à la valeur de la variable a, qui est locale à g.

$FP \rightarrow 1100$	var. globales
	...
$SP \rightarrow 1002$	$dern.\ case$

appel de g

1100	var. globales
	...
1002	
1001	$x = 3$
$FP \rightarrow 1000$	lien statique
999	1100 (vieux FP)
$SP \rightarrow 998$	$a = 50$

appel de f dans g

1100	var. globales
	...
1001	$x = 3$
1000	lien statique
999	1100 (vieux FP)
998	$a = 50$
997	$y = 3$
996	$z = 50$
$FP \rightarrow 995$	lien statique
$SP \rightarrow 994$	1000 (vieux FP)

Comment la trouver?

[3]La question est plus complexe si on garde une partie des variables dans des registres machine.

L'attribut *level* et le *lien statique*

Dans ce langage toute fonction f en exécution peut accéder (outre ses paramètres, ses propres variables et les variables globales) seulement aux variables définies dans les fonctions $g_1, \ldots, g_n$ qui l'englobent dans le texte du programme.
Ces fonctions ont un *niveau d'imbrication* inférieur à celui de f, et leur bloc d'activation est forcément sur la pile au moment de l'exécution de f (comme dans le cas de g qui englobe f dans l'exemple).

Nous pouvons associer à chaque fonction un attribut, traditionnellement appelé *level*, qui corresponde au niveau d'imbrication des fonctions.
Cet attribut sera associé ensuite à chaque variable locale de la fonction.

L'attribut *level* et le *lien statique*

Dans notre exemple, la fonction g, qui n'est définie à l'intérieur d'aucune autre fonction, aura *level=1*, alors que f aura *level=2*. Donc la variable a locale à g aura *level=1*, *offset=2*.

Maintenant, quand on compile la fonction f et on trouve la référence à la variable a, on connaît, en consultant la table des symboles, ces deux attributs.

Il ne nous reste qu'à écrire le code qui accède à la composante *offset* du *plus récent* bloc d'activation qui se trouve sur la pile et qui corresponde à une fonction *englobant* f de niveau *level*.

L'attribut *level* et le *lien statique*

Comment une fonction f peut retrouver le plus récent bloc d'activation qui se trouve sur la pile et qui corresponde à une fonction *englobant* f de niveau *level*?
Une solution simple consiste à passer à chaque fonction f, au moment de l'exécution, un pointeur vers le bloc d'activation de la fonction g *qui la définit* dans le programme. Ce pointeur est appelé le *lien statique*, et il s'ajoutera aux paramètres de la fonction.

Si nous sommes une fonction f de niveau k et nous cherchons à trouver le bloc d'activation d'une fonction g de niveau $l < k$, il nous suffira de suivre $k - l$ fois le lien statique pour le joindre.
Si nous suivons la convention de mettre toujours le lien statique en première position dans le bloc, si f cherche la variable de niveau l et offset o, elle la trouvera dans $M[\underbrace{M[\ldots M[FP]\ldots]}_{k-l\ fois} - o]$.

L'attribut *level* et le *lien statique*

Trouver la variable de *level* et *offset* donné est possible.

$FP \rightarrow 1100$	var. globales
	...
$SP \rightarrow 1000$	

appel de g

1100	var. globales
	...
1001	$x = 3$
$FP \rightarrow 1000$	lien st. = 1100
999	1100 (vieux FP)
998	$a = 50$
$SP \rightarrow 997$	

appel de f dans g

	...	
1001	$x = 3$	
1000	lien st. = 1100	bloc de g, level=1
999	1100 (vieux FP)	
998	$a = 50$	a: level=1,offset=2
997	$y = 3$	
996	$z = 50$	
$FP \rightarrow 995$	lien st. = 1000	bloc de f, level=2
994	1000 (vieux FP)	
$SP \rightarrow 993$		

Pour f (niveau 2)[4], a (niveau 1, offset 1) est $M[M[fp] - 2] = M[998]$.

[4]Nous suivons la convention de mettre toujours le lien statique en première position dans le bloc

Un exemple complexe

```
    type tree ={key: strinf, left: tree, right: tree} in
l=1 let pretty(tree:tree):string =
     let output := "" in
l=2  let write(s: string) = output:=concat(output,s) in
l=2  let show(n:int, t:tree) =
 l=3   let indent(s:string) = (for i=1 to n do write(" ") done;
                               output:=concat(output,s))
       in if t=nil then indent(".")
          else (indent(t.key);show(n+1,t.left);show(n+1,t.right))
     in show(0,tree); output
    in pretty(nil)
```

Ici, il y a plusieurs cas intéressants:

- un appel normal d'une fonction par la fonction qui la définit: pretty appelle show et passe sont propre FP comme lien statique à show
- un appel récursif de show: là on passe comme lien statique à l'appel récursif *le lien statique* du show appelant
- un appel de la part d'une fonction imbriquée d'une fonction définie plus à l'extérieur: indent appelle write et doit lui passer comme lien statique le FP de pretty. Elle l'obtient en suivant les lien statiques jusqu'au lien statique passé à show.
- indent utilise output, définie dans pretty. Elle suit la chaîne statique pour ça

Code Intermédiaire à Arbre

Nous introduisons maintenant le premier langage intermédiaire vers lequel vous allons *traduire* notre langage source.
Il s'agit encore d'une représentation arborescente, mais dans laquelle les instructions disponibles sont beaucoup plus proches des instructions machines; on retrouve en effet:

- des étiquettes, et des sauts (conditionnels ou pas) à des étiquettes
- des accès mémoire
- des déplacements des données
- des opérations de comparaison
- des opérations arithmétiques
- l'instruction CALL

AST du Code Intermédiaire à Arbre

```
module type TREE =
sig type label
    type temp
    type stm = SEQ of stm * stm
             | LABEL of label
             | JUMP of exp * label list
             | CJUMP of relop * exp * exp * label * label
             | MOVE of exp * exp
             | EXP of exp
    and exp = BINOP of binop * exp * exp
            | MEM of exp
            | TEMP of temp
            | ESEQ of stm * exp
            | NAME of label
            | CONST of int
            | CALL of exp * exp list
    and binop = PLUS | MINUS | MUL | DIV | AND | OR
              | LSHIFT | RSHIFT | ARSHIFT | XOR
    and relop = EQ | NE | LT | GT | LE | GE | ULT | ULE | UGT | UGE
end
```

Description des constructeurs Exp

- CONST(i) l'entier i (on code true comme 1, false comme 0)
- NAME(n) le symbole[5] n (une étiquette assembleur)
- TEMP(t) le "registre machine" t
- BINOP(op,e,e) les opérations élémentaires
- MEM(e) la case mémoire d'adresse e (un MOVE(MEM(e),_) sera une écriture de la case mémoire d'adresse e, alors que MOVE(_,MEM(e)) sera la lecture de la case MEM(e))
- CALL(f,l) appel de la fonction f (argument évalué en premier), avec paramètres l (évalués de gauche à droite)
- ESEQ(s,e), la valeur de l'expression e après l'exécution de la commande s

Description des constructeurs Stm

- MOVE(TEMP(t),e) évalue e et mets le résultat dans t
- MOVE(MEM(e1),e2) évalue $e1$ pour obtenir une adresse mémoire a. Ensuite évalue $e2$ et place le résultat dans le mot à l'adresse a

[5] constant

- EXP(e) évalue e, et oublie le résultat
- JUMP(e,ls) évalue e et saute au résultat. e peut être NAME(n) ou un adresse entier. ls contient les valeurs possibles de e (optionnel, sert pour l'analyse du programme)
- CJUMP(o,e1,e2,t,f) évalue e1, puis e2, et compare les résultats avec l'opérateur de comparaison o. Si $vrai$, saute à t, sinon à f
- SEQ(s1,s2) $s1$, puis $s2$
- LABEL(n) définit l'étiquette n égale à l'adresse courante

Traduction

La traduction T vers le code intermédiaire est longue, mais sans surprises. Voyons quelques cas, le reste étant laissé pour le projet.
Nous remarquons que la traduction sera effectuée en ayant accès pour toute variable simple aux attributs *level* et *offset*, et pour toute fonction à l'attribut *level*, calculés comme expliqué avant.

On supposera que chaque variable occupe exactement un mot mémoire (de taille W bytes, selon la machine).
Pour les types complexes, ce mot mémoire contiendra *un pointeur* vers la structure allouée dans le tas et pas dans la pile.
Ces conventions nous permettent de nous passer de la distinction habituelle entre valeurs gauches et valeurs droites, qu'il faut quand même rappeler brièvement ici.

L-Values (valeurs gauches) et R-values (valeurs droites)

On distingue en litérature (et notamment dans les manuels C), entre deux types de valeurs:

L-values les valeurs qui peuvent apparaître à *gauche* d'une affectation[6]. En CTigre, c'est le cas des

- **variables simples** comme x
- **champs d'enregistrements** comme $a.nom$
- **éléments d'un tableau** comme $a[3+5]$

R-values les valeurs qui peuvent apparaître à *droite* d'une affectation[7].
En CTigre, c'est le cas de tous les L-values, mais aussi d'expressions qui ne sont pas L-values:

- **expressions arithmétiques** comme $1+x-32$
- **fonctions qui retournent des types de base** comme $succ(1)$

[6] les valeurs qui désignent des cases mémoire dans lesquelles on peut écrire
[7] les expressions qui ont une valeur que l'on peut écrire dans des cases mémoire

L-Values (valeurs gauches) et R-values (valeurs droites) (suite et fin)

Comme les L-values sont les plus souvent aussi des R-values, il est nécessaire de regarder le contexte pour savoir s'il faut produire du code qui *lit* une valeur depuis la case mémoire, ou du code qui *écrit* une valeur dans la case mémoire.

Dans le cas de CTigre, nous pouvons éviter cette analyse grâce à *deux* hypothèses simples:

- toutes les variables prennent la même place (cela se fait en forçant variables de type tableaux et enregistrement à contenir un pointeur vers la mémoire plutôt que le tableau ou enregistrement tout entier)
- la construction MEM du langage intermédiaire ne préjuge pas de la lecture ou écriture, qui est décidée par le contexte

Dans ce qui suit, la traduction d'une variable simple ou composée sera donc toujours la même sans se soucier de savoir si ell'est en position droite ou gauche.

Traduction: VarExp

Le cas des variables:
La traduction de l'accès à une `SimpleVar` de *level* et *offset* o sera la suivante:

- **dans la fonction f qui la déclare (l=level(f))**:

```
MEM(BINOP(MINUS,TEMP(fp),CONST(o))
```

- **dans une fonction g englobée par f (l=level(f)<level(g))**, on suit le lien statique:

```
MEM(BINOP(MINUS,( MEM(... MEM(  TEMP(fp))...),CONST(o))))
                  level(g)-l fois
```

Traduction: éléments d'un vecteur

La traduction de l'accès à un élément d'un vecteur, $e[e1]$ sera traité comme suit:

```
MEM(BINOP(PLUS,MEM(T(e)),BINOP(MULT,T(e1),CONST(W))))
```

où W est la taille d'une case mémoire (2 ou 4 bytes d'habitude), et $T(e), T(e1)$ sont les traductions de e et $e1$.

Traduction: boucles

La traduction de `while b do c done` sera


```
SEQ(LABEL(test),
    SEQ(CJUMP(EQ,T(b),CONST(1),cont,done),
        SEQ(LABEL(cont),
            SEQ(T(c),
                SEQ(JUMP(test),LABEL(done))))))
```

Traduction: boucle while

La traduction de `while b do c done` devient plus claire si on la visualise de la façon suivante.

```
test:
       CJUMP(EQ,T(b),CONST(1),cont,done)
cont:  T(c)
       JUMP test
done:
```

Traduction: Appel d'une fonction

Le cas des variables:
La traduction d'un appel de fonction `f(a1,...,an)` est immédiate

```
CALL(NAME(lf),[sl,T(a1),...,T(an)])
```

Mais avec en plus le lien statique sl qui est ajouté en paramètre.
On vous rappelle que pour calculer sl il vous faut le *level* de f (connu, parce que vous l'avez déjà calculé) et celui de la fonction g qui appelle f (facile à connaître, parceque vous êtes en train de traduire g en ce moment).

Traduction: déclaration de variable et fonction

variable La déclaration d'une variable `let a:=e in ...` produira une expression qui initialise cette variable (dans le bloc d'activation courant à *offset* connu) avec T(e).

fonction La déclaration d'une fonction produira une étiquette lf qui est associée à la séquence d'instructions pour le prologue, suivie de la traduction du corps de la fonction, et de l'épilogue. *Important:* la traduction d'une fonction produit un nouveau Tree.stm