Liens statiques, Représentation intermédiaire

- Liens statiques
  - les blocs d'activation
  - la difficulté avec la portée statique
  - une solution: les attributs *level* et *offset*
- 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 une structure de données appelée *pile de blocs d'activation*.

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).

La pile de blocs d'activation

C'est une zone mémoire qui fonctionne comme une pile, sur laquelle on peut empiler ou dépiler des blocs de cases mémoire (les *blocs d'activation*), mais à différence d'une pile, on peut aussi accéder par leur adresse physique à des cases mémoires qui se trouvent à l'intérieur de la pile.

On peut mieux représenter cette structure de données comme un vecteur avec une variable spéciale SP (Stack Pointer, pointeur de pile), qui désigne la frontière entre les cases mémoires qui font partie de la pile et les autres, et d'une autre variable spéciale FP (Frame Pointer, pointeur du bloc), qui pointe sur la première case mémoire dans le vecteur qui appartient au dernier bloc empilé. (Ce bloc est donc délimité par SP et FP).

ATTENTION: conventions…

L'organisation de la pile des blocs d'activation varie de machine à machine. Sur certaines machines, la pile grandit vers le bas (Pentium, Sparc, Mips).

adresse plus grande

$FP \rightarrow$	case 1 du bloc
	$\ldots$
	derniere case
$SP \rightarrow$	

adresse plus petite

Mais sur d'autres elle grandit vers le haut (HPPA)…

adresse plus grande

$SP \rightarrow$	
	derniere case
	$\ldots$
$FP \rightarrow$	case 1 du bloc

adresse plus petite

conventions…suite

Selon le cas, les formules d'accès aux variables dans les blocs que l'on voit dans ce cours devraient être changée (les - deviennent des + et vice-versa).

De même, où exactement on sauvegarde FP, et comment on organise la structure interne de chaque bloc dépend de l'architecture (les constructeurs donnent un *standard layout* pour permettre les appels entre fonctions écrites en langages différents).

Un compilateur bien fait est organisé de la sorte que ces détails soient encapsulés dans un module `Frame` approprié.

Aussi, le registre SP pointe sur une case qui sépare la partie utilisée de celle libre de la pile, mais est-ce que cette case fait partie de la partie libre ou utilisée? C'est une convention qui dépende de la machine cible.

Pour simplicité, dans ce cours nous assumons simplement que la pile croît vers le bas, et que SP pointe sur la première case vide de la pile…

conventions… fin

et que l'organisation du bloc d'activation soit la suivante (le pourquoi sera plus clair avant):

adresse plus grande

	in-arg 1
	…
	in-arg n
$FP \rightarrow$	lien statique
	place pour FP
	var. loc. 1
	…
	var. loc. k
	adresse de retour
	autres…
$SP \rightarrow$	

adresse plus petite

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

Voyons comment peut se dérouler un appel de fonction f, en supposant que chaque paramètre et variable occupe exactement une case mémoire.
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 (c'est bien des "empilements", avec SP qui décroît…)
- l'appelant mets le liens statique de f dans SP (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 et *alloue son bloc* (de taille K)

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

- 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 "dépile" son bloc et retourne le contrôle à l'appelant

- f *désalloue son bloc*, restaure les valeurs de SP et FP, et saute à l'adresse de retour:

```
SP <- FP
FP <- M[SP-1]
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 liens 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
	$\ldots$
$SP \rightarrow 1001$	

empile params g

$FP \rightarrow 1100$	var. globales
	$\ldots$
1001	$x = 3$
$SP \rightarrow 1000$	

appel de g

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

Évolution de SP et FP

g empile les args. de f

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

appel de f dans g

1100	var. globales
	$\ldots$
1001	$x = 3$
1000	liens statique
999	$1100(vieuxFP)$
998	$a = 50$
997	$y = 3$
996	$z = 50$
$FP \to$ 995	liens statique
994	1000 (vieux FP)
$SP \to$ 993	

Évolution de SP et FP

f retourne

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

g dépile params de f

1100	var. globales
	$\ldots$
1001	$x = 3$
$FP \to$ 1000	liens statique
999	1100 (vieux FP)
998	$a = 50$
$SP \to$ 997	

Les variables de la fonction courante se trouvent dans une suite contiguë de cases sur la pile, nommée *bloc*. Ce bloc est délimité par les valeurs de deux registres, nommés FP (ang. frame pointer = pointeur de bloc) et SP (ang. stack pointer = pointeur de pile).

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 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 liens statique et FP).

La question deviendra plus complexe si on décide de garder une partie des variables dans des registres machine.

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
	$\ldots$
$SP \rightarrow 1000$	

appel de g

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

appel de f dans g

1100	var. globales
	$\ldots$
1001	$x = 3$
1000	liens statique
999	$1100(vieuxFP)$
998	$a = 50$
997	$y = 3$
996	$z = 50$
$FP \rightarrow 995$	liens statique
994	1000 (vieux FP)
$SP \rightarrow 993$	

Comment la trouver?

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

Dans ce langage toute fonction f en exécution peut accéder (à part 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*

Cela fait, trouver la variable de *level* et *offset* donné ne pose plus de problèmes.

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

appel de g

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

appel de f dans g

1100	var. globales
	...
1001	$x = 3$
1000	liens st. = 1100
999	$1100 (vieux FP)$
998	$a = 50$
997	$y = 3$
996	$z = 50$
$FP \rightarrow 995$	liens st. = 1000
994	1000 (vieux FP)
$SP \rightarrow 993$	

Si nous suivons la convention de mettre toujours le lien statique en première position dans le bloc, pour f, de niveau 2 la variable a, de niveau 1 et offset 1 n'est rien d'autre que $M[M[fp] - 2] = M[998]$.

Un exemple complexe

```
type tree ={key: strinf, left: tree, right: tree}
in
let pretty(tree:tree):string =
 let output := "" in
 let write(s: string) = output:=concat(output,s) in
 let show(n:int, t:tree) =
       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 write et passe sont propre FP comme liens statique à show
- un appel récursif de show: là on passe comme liens statique à l'appel récursif *le liens 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 liens statique le FP de pretty. Elle l'obtient en suivant les liens statiques jusqu'au lien statique passé à show.

- indent utilise output, définie dans pretty. Elle suit la chaîne statique pour ça

Regles pour le calcul et l'utilisation du lien statique

- fonction g de niveau k appelle une fonction h de niveau $k+1$ definie localement a g: on passe comme lien statique le FP de g

- fonction g de niveau k appelle une fonction h de niveau k: on passe comme lien statique de h le lien statique de g (cela fonctionne aussi bien pour la recursion)

- fonction g de niveau k appelle une fonction h definie a l'exterieur (niveau $k' < k$): on doit passer a h le lien statique pour le niveau k', qui s'obtient en suivant $k - k'$ fois la chaîne statique

Optimisations

On peut optimiser ce mechanisme de suivi du liens statique de plusieurs façons:

displays on garde un tableau où chaque case i contien le pointeur sur le FP du bloc de niveau i empilé le plus récémment. Cela “comprime” la chaîne statique au prix constant d'une sauvegarde et restauration de la valeur de cette case à chaque entrée et sortie de bloc.
C'est la solution utilisée dans les compilateurs Pascal.

lambda-lifting on peut transformer le code de façon à toujous passer explicitement toute variable non-locale comme paramètre supplémentaire (on n'a plus bésoin de chaîne statique, mais le passage des paramètres devient couteux).

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
```

Commentaire

Voici une description des constructeurs Exp:

expressions (exp)

- CONST(i) l'entier i (on codera true comme 1 et false comme 0)
- NAME(n) la constante symbolique 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

Commentaire

Voici une description des constructeurs Stm:

commandes (stm)

- MOVE(TEMP(t),e) évalue e et mets le résultat dans t
- MOVE(MEM(e1,k),e2) évalue $e1$ pour obtenir une adresse mémoire a. Ensuite évalue $e2$ et place le résultat dans les k bytes à partir de a
- 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. La liste ls est l'ensemble des valeurs possibles de e (optionnel, sert pour une 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 comme partie du projet.

Mais nous remarquons maintenant 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 aussi que chaque variable occupe exactement un mot mémoire (de taille W bytes, selon la machine).
Aussi, pour les types complexes, ce mot mémoire contiendra *un pointeur* vers la structure allouée dans le tas et pas dans la pile.

Traduction: Variables

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 liens statique:

$$\texttt{MEM(BINOP(MINUS,(}\underbrace{\texttt{MEM(... MEM(}}_{\texttt{level(g)-l fois}}\texttt{TEMP(fp))...),CONST(o))))}$$

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(n),
    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` est plus simple de la visualiser 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 liens 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.