Chapitre 5

Table des symboles, niveaux, décalages

Sommaire

5.1 Table des symboles **1**
5.1.1 Portée des identificateurs 1
5.1.2 Table des symboles 3
5.2 Level et offset **6**
5.2.1 Blocs d'activation 6
5.2.2 Attributs *level* et *offset* 9
5.2.3 Analyse d'échappement (escape analysis) 11

5.1 Table des symboles

5.1.1 Portée des identificateurs

Dans les langages modernes, les identificateurs ont presque toujours une portée *statique* limitée au bloc dans lequel ils sont définis. En CTigre, (comme en Pascal, C++ ou ML, quoique avec une syntaxe différente), considérons un programme contenant le fragment :

```
var a := e1 in
e2
```

L'identificateur `a` est lié à la valeur `e1` dans le corps de l'expression `e2`, et cette liaison (*binding* en anglais) est *détruite* dès que l'on sort de `e2`.

Dans les différentes phases de la compilation, on parcourt l'arbre abstrait pour l'analyse statique (typage, etc.) et la traduction, et on a besoin de savoir, à chaque instant, quelle est la valeur des attributs (type, level, offset, etc.) associés à un identificateur donné. La *table des symboles* permet de centraliser cette information.

Definition 1 (Liaison)**.** On notera dans la suite $x \mapsto v$ la *liaison* entre un identificateur x et un objet v.

Definition 2 (Environnement)**.** On appelle *environnement* un ensemble de liaisons. Selon la nature des valeurs associées aux identificateurs, (constantes, cases mémoire, fonctions, types, etc.) on aura divers environnements. Un environnement s'écrira $\{x_1 \mapsto v_1, \ldots x_n \mapsto v_n\}$.

Lors de la compilation on a besoin de connaître, quand on rencontre un usage d'un identificateur, quelle est la définition *active* pour cet usage.

Un même nom d'identificateur, défini à l'entrée d'un bloc, peut être *redéfini* dans un bloc plus interne, et donc il ne fera pas forcément référence au même objet.Exemple : considérons le programme

```
0 type a = int in            a : nom de type, visible jusqu'à 8
1 var a := 3 in              a : nom de variable, visible en 2,3,6,7,8
2    function f(x : a) =     x : nom de variable, visible en 3,4,5,6,7
3       var b :=             b : nom de variable, visible en 4,5,6,7
4          var a := 'A'      a : nom de variable, visible en 5
5          in ord(a)+x
6       in print("Ord(A)+"); printint(x); print (" = ");
7          printint(b); print(" a vaut "); printint(a)
8    in f(a)
```

L'identificateur `a` défini comme une case mémoire entière contenant l'entier 3 à la ligne 1 est en principe visible dans toute l'expression entre les lignes 3 et 8, mais à la ligne 4 on trouve une redéfinition qui fait en sorte que la liaison entre `a` et 3 est *cachée* dans la ligne 5 par la liaison entre `a` et `'A'` définie à la ligne 4. Cependant, la redéfinition de la ligne 1 ne cache pas la définition de type de la ligne 0 !

Exemple

Suivons l'évolution de l'environnement des liaisons de type et de l'environnement qui donne le type des identificateurs sur le programme de l'exemple :

Ligne	Prog	Tenv (Valeur du type)	TVEnv (Type de l'identificateur)
0	`type a = int in`	$\emptyset$	$\emptyset$
1	`var a := 3 in`	$\{a \mapsto int\}$	$\emptyset$
2	`function f(x :a) =`	$\{a \mapsto int\}$	$\{a \mapsto (l1, int)\}$
3	`var b :=`	$\{a \mapsto int\}$	$\{a \mapsto (l1, int);\ x \mapsto (l2, a)\}$
4	`var a := 'A'`	$\{a \mapsto int\}$	$\{a \mapsto (l1, int);\ x \mapsto (l2, a)\}$
5	`in ord(a)+x`	$\{a \mapsto int\}$	$\{a \mapsto (l5, char);\ x \mapsto (l2, a)\}$
6	`in ...`	$\{a \mapsto int\}$	$\{a \mapsto (l1, int);\ x \mapsto (l2, a);\ b \mapsto (l3, int)\}$
7			
8	`in f(a)`	$\{a \mapsto int\}$	$\{a \mapsto (l1, int)\}$

À retenir pour les langages avec structures de blocs :

portée statique la portée d'une définition d'identificateur (variable, fonction, procédure, type), i.e. la partie du programme où la liaison pour cet identificateur créée par la définition est active, peut-être obtenu *statiquement* (en analysant le texte du programme).

redéfinitions/hiding une liaison peut être *cachée* temporairement par une nouvelle liaison *de même nature* pour un même identificateur, mais seulement dans un bloc plus interne du bloc de la première définition.

LIFO l'ordre dans lequel les liaisons sont introduites est détruites est du genre « Last In First Out », ce qui suggère une pile comme structure de donnée adaptée (tant à la compilation qu'à l'exécution).

5.1.2 Table des symboles

Nous allons utiliser plusieurs environnements lors de la compilation. À chaque parcours de l'arbre (typage, décoration...), cet environnement va évoluer.

Portée des identificateurs : structures de données

On doit trouver des structures de données adaptées à la mise en oeuvre des environnements lors de la compilation.

Voilà nos desiderata :

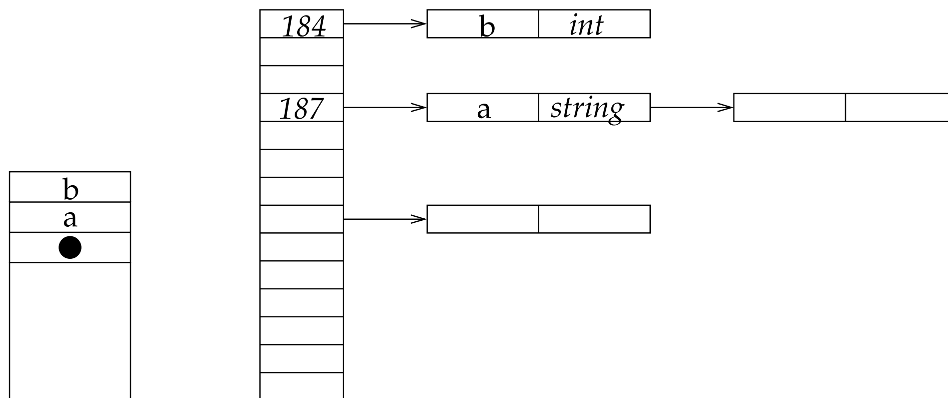
- accès rapide aux liaisons par le nom de l'identificateur
- gestion facile de l'évolution des environnements, notamment de l'opération d'ajout d'une liaison et de suppression d'une liaison en restaurant l'(éventuelle) liaison précédente

Definition 3 (Somme d'environnements)**.** Si σ est un environnement, on écrit $\sigma + \{x \mapsto v\}$ pour l'environnement qui associe v à x, et coïncide avec l'environnement σ sinon. On écrira $\sigma_1 + \sigma_2$ pour l'environnement qui contient les liaisons de σ_1 et σ_2, mais en donnant priorité à celles de σ_2.

N.B. : l'opération $+$ ci-dessus n'est pas commutative.

Solution *impérative*

On utilise une *table de hachage* spécifique, avec une pile de *undo*.



entrée dans un bloc on empile un marqueur, puis on *insère* les définitions locales

insertion on ajoute la valeur en tête de la liste correspondant à la clef dans la table et on empile la clef sur la pile de undo.

sortie du bloc on dépile et on *supprime* toutes les clefs jusqu'au marqueur de bloc

suppression en enlevant l'élément en tête de liste.

Cela peut être mis en place dans un module qui fournit les primitives suivantes :

– `insert` qui fait à la fois l'insertion en table de hachage et sur la pile de undo,
– `begin_scope`, qui met le marqueur sur la pile de undo,
– `end_scope`, qui dépile les liaisons jusqu'au marqueur compris et les enlève de la table.

C'est l'approche de `gcc`.

Solution ***fonctionnelle pure***

Solution simple : listes d'association

Insertion en tête de liste

Recherche de la première valeur avec la clé (qui masque les éventuelles définitions précédentes)

Sortie de bloc on reprend l'ancien environnement

Trouvons une solution plus efficace.
On utilise des arbres binaires équilibrés, sans pile de *undo*.

entrée dans un bloc : on construit un nouvel environnement à partir du précédent,

sortie d'un bloc : on abandonne le nouvel environnement et on utilise l'ancien

```
let checktype tenv venv = function
  ...
  | LetExp(VarDecl([x,v]),e) ->
        let newvenv=insert(venv,x,type(v))
        in checktype tenv newvenv e
  | SeqExp(e1,e2)            -> checktype tenv venv e1;
                                checktype tenv venv e2
  ...
```

Pour que cela soit efficace, il faut savoir construire l'objet modifié sans trop « copier ».

Interface du module de la table des symboles

```
type symbol = string (* a enlever dans le code en production *)
type 'a table
val symbol : string -> symbol
val name : symbol -> string
val mkempty : unit -> 'a table
val add : symbol -> 'a -> 'a table -> 'a table
val find : symbol -> 'a table -> 'a
...
```

En Ocaml

Le compilateur Ocaml utilise l'approche fonctionnelle, (mais avec quelques subtilités).
Module `typing/ident.ml` :

```
type 'a tbl =
    Empty
  | Node of 'a tbl * 'a data * 'a tbl * int

let mknode l d r =
  let hl = match l with Empty -> 0 | Node(_,_,_,h) -> h
```

```
  and hr = match r with Empty -> 0 | Node(_,_,_,h) -> h in
  Node(l, d, r, (if hl >= hr then hl + 1 else hr + 1))

let balance l d r =
  let hl = match l with Empty -> 0 | Node(_,_,_,h) -> h
  and hr = match r with Empty -> 0 | Node(_,_,_,h) -> h in
  if hl > hr + 1 then
    match l with
    | Node (ll, ld, lr, _)
      when (match ll with Empty -> 0 | Node(_,_,_,h) -> h) >=
           (match lr with Empty -> 0 | Node(_,_,_,h) -> h) ->
        mknode ll ld (mknode lr d r)
    | Node (ll, ld, Node(lrl, lrd, lrr, _), _) ->
        mknode (mknode ll ld lrl) lrd (mknode lrr d r)
    | _ -> assert false
  else if hr > hl + 1 then
    match r with
    | Node (rl, rd, rr, _)
      when (match rr with Empty -> 0 | Node(_,_,_,h) -> h) >=
           (match rl with Empty -> 0 | Node(_,_,_,h) -> h) ->
        mknode (mknode l d rl) rd rr
    | Node (Node (rll, rld, rlr, _), rd, rr, _) ->
        mknode (mknode l d rll) rld (mknode rlr rd rr)
    | _ -> assert false
  else
    mknode l d r

let rec add id data = function
    Empty ->
      Node(Empty, {ident = id; data = data; previous = None}, Empty, 1)
  | Node(l, k, r, h) ->
      let c = compare id.name k.ident.name in
      if c = 0 then
        Node(l, {ident = id; data = data; previous = Some k}, r, h)
      else if c < 0 then
        balance (add id data l) k r
      else
        balance l k (add id data r)

let rec find_name name = function
    Empty ->
      raise Not_found
  | Node(l, k, r, _) ->
      let c = compare name k.ident.name in
      if c = 0 then
        k.data
```

```
else
  find_name name (if c < 0 then l else r)
```

5.2 Level et offset

5.2.1 Blocs d'activation

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'exécution 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).

Lien statique

Le bloc d'activation d'une fonction est un espace de pile qui contient la sauvegarde de registres, les variables locales, et les paramètres des fonctions. Comme les fonctions ont accès aux variables des fonctions englobantes, il faut pouvoir retourner au bloc de ces fonctions. Pour cela, on sauvegarde dans la pile la valeur du FP de la fonction de niveau supérieur. C'est ce que l'on appelle le *lien statique*. Nous en parlerons dans la section suivante.

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 après) :

adresse plus grande

	in-arg n
	...
	in-arg 1
$FP \rightarrow$	lien statique
	var. loc. 1
	...
	var. loc. k
	place pour les temporaires
	place pour FP
	place pour ra
$SP \rightarrow$	place pour les autres registres...
...	place pour les arguments de la fonction
$SP \rightarrow$	qu'on appelle...

adresse plus petite

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

Voyons comment se déroule 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 faite par l'appelant :

- l'appelant réserve la place dans la pile pour les paramètres en décalant SP
- l'appelant met en place les m paramètres actuels de la fonction appelée f dans la partie de son bloc près de SP réservée pour cela

$$\begin{array}{rcl} M[SP+1] & \leftarrow & arg_1 \\ & \dots & \\ M[SP+n] & \leftarrow & arg_n \end{array}$$

- l'appelant met en place le lien statique ls de f

$$M[SP] \quad \leftarrow \quad ls$$

- l'appelant appelle la fonction f (instruction `CALL f`)
- l'appelant libère la place pour les paramètres sur la pile

Et voici la partie du travail qui est faite par l'appelé :

prologue *f alloue son bloc*, de taille `framesize` sur la pile

$$SP \quad \leftarrow \quad SP - \texttt{framesize}$$

- l'appelé sauvegarde la valeur de FP, à la position `frameoffset` dans le bloc...

$$M[SP + \texttt{framesize} - \texttt{frameoffset}] \quad \leftarrow \quad FP$$

  et positionne FP

$$FP \quad \leftarrow \quad SP + \texttt{framesize}$$

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

calcul on exécute le corps de f, le résultat est dans un registre spécial `$v0`

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

$$\begin{array}{rcl} FP & \leftarrow & M[SP + \texttt{framesize} - \texttt{frameoffset}] \\ SP & \leftarrow & SP + \texttt{framesize} \end{array}$$

et saute à l'adresse de retour : `JUMP $ra`

Un exemple

Considérons le programme suivant

```
function g(x:int): int =
    var a := 50 in
    function 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

À 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
	...
$SP \rightarrow 1002$	...

Préparation de l'appel de g

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

appel de g

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

g empile les args. de f

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

appel de f dans g

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

on sort de f

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

on sort de g

$FP \rightarrow 1100$	var. globales
	...
$SP \rightarrow 1002$	...
1001	$x = 3$
$SP \rightarrow 1000$	lien statique

5.2.2 Attributs *level* et *offset*

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 l'on assume (la question est plus complexe si l'on garde une partie des variables dans des registres machine) 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 1 pour la première déclaration, 2 pour la deuxième etc. (la position 0 est prise par le lien statique).

Pour les paramètres formels, qui se trouvent positionnés de l'autre côté 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)

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

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

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*, correspondant au niveau d'imbrication des fonctions. Cet attribut sera associé ensuite à chaque variable locale de la fonction.

Dans notre exemple, la fonction g, qui est définie à l'intérieur du programme principal, aura *level=2*, alors que f aura *level=3*. Donc la variable a locale à g aura *level=2*, *offset=1*.

Maintenant, quand on compile la fonction f et que l'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*.

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

$$M[\underbrace{M[\ldots M[}_{k-l\ fois}FP]\ldots] - o]$$

Étant donnés un *level* et un *offset*, il est possible de trouver une variable :

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

Pour f (niveau 3), a (niveau 2, offset 1) est $M[M[fp] - 1] = M[999]$ (Nous suivons la convention de mettre toujours le lien statique en première position dans le bloc).

Un exemple complexe

```
     type tree = {key: string, left: tree, right: tree} in
l=2    function pretty(tree:tree):string =
       var output := "" in
  l=3    function write(s: string) = output:=concat(output,s) in
  l=3    function show(n:int, t:tree) =
    l=4    function 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 son 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 liens statiques jusqu'au lien statique passé à `show` ;
- `indent` utilise output, définie dans `pretty`. Elle suit la chaîne statique pour cela.

Table pour le calcul du lien statique à passer à l'appelé

niveau appelant	niveau appelé	lien statique à empiler	note
l	l+1	mon FP	
l	l	mon LS	= M[FP]
l	l-k	$\underbrace{M[\ldots M[}_{k fois} LS] \ldots]$	= $\underbrace{M[\ldots M[}_{k+1 fois} FP] \ldots]$

N.B. : au moment ou l'on compile un appel de fonction, on aura sous la main à la fois les informations sur l'appelant (que l'on est en train de compiler) et de l'appelé (dont étiquette et niveau seront déjà connus).

Les informations qui nous servent sur un bloc

À titre d'exemple, voici un extrait des déclarations de type pour un fragment dans notre implémentation du compilateur CTigre :

```
type frame = {
  entry: Temp.label; (** point d'entrée, au début du prologue *)
  mutable maxargs: int; (** nombre maximum d'arguments d'une fonction appelée *)
  mutable numtemps: int; (** nombre de variables/temporaires sur la pile *)
  }
```

– `numtemps` : le nombre de mots utilisés pour variables locales + temporaires on ne le connaîtra qu'à la fin de la chaîne de compilation (même la sélection d'instructions crée des temporaires !). Vous le calculerez juste avant d'émettre prologue et épilogue pour l'assembleur.

Donc, il faudra garder l'information sur le bloc jusqu'au bout !

5.2.3 Analyse d'échappement (escape analysis)

On divise les variables locales d'une fonction f en deux groupes :

variables utilisées seulement par f : ces variables locales peuvent, s'il y a la place, être mises dans des registres machines ;

variables utilisées aussi en dehors de f : par exemple, la variable a du dernier exemple est locale à g, mais utilisée par f ; On dit que ces variables « échappent », et on est en général obligés de les allouer dans la pile, pour qu'on puisse leur donner une adresse utilisable par qui fait référence à ces variables en dehors de la fonction qui les a créées. Dans des langages comme C, une variable dont on prend l'adresse, comme dans

```
void *breakit() {
        int i=100;
        . . .
        return(&i);
}
```

« échappe » aussi.

Approximation dans le projet

Dans le cadre de votre projet, on assumera d'abord que toutes les variables locales échappent, et donc on les allouera toutes en pile. Dans une deuxième version, on procédera à une « analyse d'échappement ». Pour un langage comme CTigre c'est assez simple : en visitant l'AST, on peut savoir si une variable sera ou pas utilisée en dehors de la fonction qui la définit. On peut alors allouer en pile (en calculant level et offset) seulement les variables qui échappent. Cela laisse une chance à l'allocateur de registres de mettre quelques variables dans de vrais registres machine.