

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*
- 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
	<i>Bloc 1</i>
	\vdots
	<i>Bloc n - 1</i>
<i>FP</i> →	première case du bloc <i>n</i>
	\dots
<i>SP</i> →	dernière case du bloc <i>n</i>
	adresse plus petite

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

	adresse plus grande
<i>SP</i> →	dernière case du bloc <i>n</i>
	\dots
<i>FP</i> →	première case du bloc <i>n</i>
	<i>Bloc n - 1</i>
	\vdots
	<i>Bloc 1</i>
	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 une *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.

Important: les "conventions" sont là pour permettre à des codes objets produits par des compilateurs différents de pouvoir interagir correctement.

conventions... fin

Pour simplicité, dans ce cours nous assumons simplement que la pile croît vers le bas, que SP pointe sur la première case vide de la pile... 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 →	lien statique
	place pour FP
	var. loc. 1
	...
	var. loc. k
	adresse de retour
	autres...
SP →	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 lien statique de f dans $M[SP]$ (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...

$M[SP-1] \leftarrow FP$

et alloue son bloc (de taille K)

$FP \leftarrow SP$

$SP \leftarrow SP-K$

- f a reçu par l'appelant dans un registre spécial ret l'adresse de retour¹. 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 "dépille" son bloc et retourne le contrôle à l'appelant

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

$SP \leftarrow FP$

$FP \leftarrow M[SP-1]$

et saute à l'adresse de retour:

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.

empile params g		appel de g	
		1100	var. globales
			...
		1001	$x = 3$
FP → 1100	var. globales	FP → 1000	lien statique
	...	999	1100 (vieux FP)
SP → 1001		998	$a = 50$
		SP → 997	

Évolution de SP et FP

¹ Ceci est bien mieux que retrouver l'adresse ret sur la pile, pourquoi?

g empile les args. de f		appel de f dans g	
1100	var. globales	1100	var. globales
	...	1001	...
1001	$x = 3$	1000	lien statique
$FP \rightarrow 1000$	lien statique	999	1100 (vieux FP)
999	1100 (vieux FP)	998	$a = 50$
998	$a = 50$	997	$y = 3$
997	$y = 3$	996	$z = 50$
996	$z = 50$	$FP \rightarrow 995$	lien statique
$SP \rightarrow 995$		994	1000 (vieux FP)
		$SP \rightarrow 993$	

Évolution de SP et FP

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

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 accèdera à cette variable.

Si on assume² que toutes les variables locales sont mémorisées 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 côté de *FP*, on peut choisir un *offset* négatif.

²La question est plus complexe si on garde une partie des variables dans des registres machine.

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*.

appel de g		appel de f dans g	
1100	var. globales	1100	var. globales
	...	1001	...
1001	$x = 3$	1000	lien statique
$FP \rightarrow 1000$	lien statique	999	1100 (vieux FP)
999	1100 (vieux FP)	998	$a = 50$
998	$a = 50$	997	
$SP \rightarrow 997$		996	
		$FP \rightarrow 995$	lien 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 (outre ses paramètres, ses propres variables et les variables globales) seulement aux variables définies dans les fonctions g_1, \dots, 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 $M[\underbrace{M[\dots M[FP]\dots]}_{k-l \text{ fois}}] - o$.

L'attribut *level* et le lien statique

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

appel de g		appel de f dans g	
		1001	...
		1000	$x = 3$
		999	lien st. = 1100
		998	...
		997	1100(vieux FP)
		996	$a = 50$
		995	$y = 3$
		994	$z = 50$
		993	lien st. = 1000
		992	1000 (vieux FP)
		991	
		990	
		989	
		988	
		987	
		986	
		985	
		984	
		983	
		982	
		981	
		980	
		979	
		978	
		977	
		976	
		975	
		974	
		973	
		972	
		971	
		970	
		969	
		968	
		967	
		966	
		965	
		964	
		963	
		962	
		961	
		960	
		959	
		958	
		957	
		956	
		955	
		954	
		953	
		952	
		951	
		950	
		949	
		948	
		947	
		946	
		945	
		944	
		943	
		942	
		941	
		940	
		939	
		938	
		937	
		936	
		935	
		934	
		933	
		932	
		931	
		930	
		929	
		928	
		927	
		926	
		925	
		924	
		923	
		922	
		921	
		920	
		919	
		918	
		917	
		916	
		915	
		914	
		913	
		912	
		911	
		910	
		909	
		908	
		907	
		906	
		905	
		904	
		903	
		902	
		901	
		900	
		899	
		898	
		897	
		896	
		895	
		894	
		893	
		892	
		891	
		890	
		889	
		888	
		887	
		886	
		885	
		884	
		883	
		882	
		881	
		880	
		879	
		878	
		877	
		876	
		875	
		874	
		873	
		872	
		871	
		870	
		869	
		868	
		867	
		866	
		865	
		864	
		863	
		862	
		861	
		860	
		859	
		858	
		857	
		856	
		855	
		854	
		853	
		852	
		851	
		850	
		849	
		848	
		847	
		846	
		845	
		844	
		843	
		842	
		841	
		840	
		839	
		838	
		837	
		836	
		835	
		834	
		833	
		832	
		831	
		830	
		829	
		828	
		827	
		826	
		825	
		824	
		823	
		822	
		821	
		820	
		819	
		818	
		817	
		816	
		815	
		814	
		813	
		812	
		811	
		810	
		809	
		808	
		807	
		806	
		805	
		804	
		803	
		802	
		801	
		800	
		799	
		798	
		797	
		796	
		795	
		794	
		793	
		792	
		791	
		790	
		789	
		788	
		787	
		786	
		785	
		784	
		783	
		782	
		781	
		780	
		779	
		778	
		777	
		776	
		775	
		774	
		773	
		772	
		771	
		770	
		769	
		768	
		767	
		766	
		765	
		764	
		763	
		762	
		761	
		760	
		759	
		758	
		757	
		756	
		755	
		754	
		753	
		752	
		751	
		750	
		749	
		748	
		747	
		746	
		745	
		744	
		743	
		742	
		741	
		740	
		739	
		738	
		737	
		736	
		735	
		734	
		733	
		732	
		731	
		730	
		729	
		728	
		727	
		726	
		725	
		724	
		723	
		722	
		721	
		720	
		719	
		718	
		717	
		716	
		715	
		714	
		713	
		712	
		711	
		710	
		709	
		708	
		707	
		706	
		705	
		704	
		703	
		702	
		701	
		700	
		699	
		698	
		697	
		696	
		695	
		694	
		693	
		692	
		691	
		690	
		689	
		688	
		687	
		686	
		685	
		684	
		683	
		682	
		681	
		680	
		679	
		678	
		677	
		676	
		675	
		674	
		673	
		672	
		671	
		670	
		669	
		668	
		667	
		666	
		665	
		664	
		663	
		662	
		661	
		660	
		659	
		658	
		657	
		656	
		655	
		654	
		653	
		652	
		651	
		650	
		649	
		648	
		647	
		646	
		645	
		644	
		643	
		642	
		641	
		640	
		639	
		638	
		637	
		636	
		635	
		634	
		633	
		632	
		631	
		630	
		629	
		628	
		627	
		626	
		625	
		624	
		623	
		622	
		621	
		620	
		619	
		618	
		617	
		616	
		615	
		614	
		613	
		612	
		611	
		610	
		609	
		608	
		607	
		606	
		605	
		604	
		603	
		602	
		601	
		600	
		599	
		598	
		597	
		596	
		595	
		594	
		593	
		592	
		591	
		590	
		589	
		588	
		587	
		586	
		585	
		584	
		583	
		582	
		581	
		580	
		579	
		578	
		577	
		576	
		575	
		574	
		573	
		572	
		571	
		570	
		569	
		568	
		567	
		566	
		565	
		564	
		563	
		562	
		561	
		560	
		559	
		558	
		557	
		556	
		555	
		554	
		553	
		552	
		551	
		550	
		549	
		548	
		547	
		546	
		545	
		544	
		543	
		542	
		541	
		540	
		539	
		538	
		537	
		536	
		535	
		534	
		533	
		532	
		531	
		530	
		529	
		528	
		527	
		526	
		525	
		524	
		523	
		522	
		521	
		520	
		519	
		518	
		517	
		516	
		515	
		514	
		513	
		512	
		511	
		510	
		509	
		508	
		507	
		506	
		505	
		504	
		503	
		502	
		501	
		500	
		499	
		498	
		497	
		496	
		495	
		494	
		493	
		492	
		491	
		49	

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 symbole⁴ *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(*e*1,*k*),*e*2) évalue *e*1 pour obtenir une adresse mémoire *a*. Ensuite évalue *e*2 et place le résultat dans les *k* bytes à partir de *a*

⁴constant

- 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*,*e*1,*e*2,*t*,*f*) évalue *e*1, puis *e*2, et compare les résultats avec l'opérateur de comparaison *o*. Si *vrai*, saute à *t*, sinon à *f*
- SEQ(*s*1,*s*2) *s*1, puis *s*2
- 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 après les vacances et 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 littérature (et notamment dans les manuels C), entre deux types de valeurs:

L-values les valeurs qui peuvent apparaître à *gauche* d'une affectation⁵. 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 affectation⁶.

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)

⁵les valeurs qui désignent des cases mémoire dans lesquelles on peut écrire

⁶les 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 elle 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 ($l = \text{level}(f)$):

$\text{MEM}(\text{BINOP}(\text{MINUS}, \text{TEMP}(fp), \text{CONST}(o)))$

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

$\text{MEM}(\text{BINOP}(\text{MINUS}, (\underbrace{\text{MEM}(\dots \text{MEM}(\text{TEMP}(fp)) \dots)}_{\text{level}(g)-1 \text{ fois}}), \text{CONST}(o))))$

Traduction: éléments d'un vecteur

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

$\text{MEM}(\text{BINOP}(\text{PLUS}, \text{MEM}(T(e)), \text{BINOP}(\text{MULT}, T(e1), \text{CONST}(W))))$

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 $f(a1, \dots, an)$ est immédiate

$\text{CALL}(\text{NAME}(lf), [sl, T(a1), \dots, T(an)])$

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, parce que 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