UPMC/master/info/4I503 APS
Notes de cours

P. Manoury

Janvier 2018

Contents

1 APS2: tableaux

Il s'agit ici d'ajouter des structures de tableaux: séquences consécutives en mémoire de valeurs. Cet ajout va modifier le mécanisme d'allocation mémoire. Le langage APS2 fournira une primitive explicite d'allocation d'un *bloc mémoire*. Ces blocs mémoire constitueront une nouvelle catégorie de *valeurs* manipulées par le langage. On trouvera ces valeurs dans les environnements, mais également en mémoire.

Les tableaux seront des structures modifiables (comme les variables). L'ajout de ces structures aura donc un impact sur l'instruction d'affectation. En particulier, *à gauche*[1] d'une affectation pourront figurer des expressions désignant un élément de tableau et non plus simplement un identificateur. Nous aurons donc une nouvelle catégorie syntaxique pour les membres gauches des affectations. Nous emprunterons son nom au terme *lvalue* que l'on trouve, par exemple, dans la grammaire de C.

Du point de vue des types, les tableaux sont des *types paramétrés*. Les fonctions primitives pour les tableaux que nous introduisons dans *APS2* sont applicables à des tableaux quelque soit le type des éléments qu'ils contiennent. En ce sens, ces opérateurs sont *polymorphes*.

1.1 Syntaxe

Le langage est étendu avec des symboles pour les opérations primitives sur les tableaux: création d'un tableau, désignation d'un élément, longueur. Le langage des types est également étendu avec un symbole et une notation pour le *type* des tableaux.

1.1.1 Lexique

Opérateurs primitifs

`len nth alloc`

Type

`vec`

1.1.2 Grammaire

Le principal ajout à la grammaire est le non-terminal LVAL qui peut prendre place comme premier terme (*lvalue*) dans l'instruction d'affectation. On ajoute également une nouvelle construction de type pour les tableaux.

STAT	::=	...
	|	`SET` LVAL EXPR
LVAL	::=	ident
	|	`(nth` LVAL EXPR `)`
TYPE	::=	...
	|	`(vec` TYPE `)`

1.2 Typage

1.2.1 Expressions

Le polymorphisme des opérateurs primitifs est traité au niveau des règles de typage, comme l'était le polymorphisme de l'opérateur fonctionnel d'alternative (le `if`):

(ALLOC) pour tout $t \in$ TYPE, si $\Gamma \vdash_{\text{EXPR}} e$: `int` alors $\Gamma \vdash_{\text{EXPR}}$ `(alloc` e`)` : `(vec` t`)`

(NTH) pour tout $t \in$ TYPE, si $\Gamma \vdash_{\text{EXPR}} e_1$: `(vec` t`)` et si $\Gamma \vdash_{\text{EXPR}} e_2$: `int` alors $\Gamma \vdash_{\text{EXPR}}$ `(nth` e_1 e_2`)` : t

(LEN) pour tout $t \in$ TYPE, si $\Gamma \vdash_{\text{EXPR}} e$: `(vec` t`)` alors $\Gamma \vdash_{\text{EXPR}}$ `(len` e`)` : `int`

[1] L'expression *«à gauche»* est héritée de la notation usuelle de l'affectation par un symbole infixe: `x = x+1`, par exemple.

1.2.2 Instructions

La seule instruction dont le typage est affecté par le passage de *APS1* à *APS2* est l'affectation où le premier terme peut prendre la forme d'une expression.

(SET) si $\Gamma \vdash_{\text{Expr}} lv : t$ et si $\Gamma \vdash_{\text{Expr}} e : t$ alors $\Gamma \vdash_{\text{Stat}} (\texttt{SET}\ lv\ e) : \texttt{void}$

1.3 Sémantique

Pour *APS2*, la sémantique va connaître des changements assez importants. Le premier est l'extension à une nouvelle catégorie de valeurs qui figureront dans les environnement et dans la mémoire: les blocs de valeurs. Le deuxième tient à ce qu'une expression peut maintenant avoir un effet sur la mémoire: l'expression d'allocation. Un dernier impact est que le domaine des adresses ne peut plus rester complètement abstrait car nous y avons besoin de la notions d'adresses *consécutives*.

1.3.1 Domaines et opérations sémantiques

Domaines Il faut choisir un domaine d'adresses où l'on possède la notion de succession afin de pouvoir obtenir des *intervalles* d'adresse. Un candidat naturel pour définir ce domaine est celui des entiers naturels. Un bloc mémoire peut alors simplement être représenté par l'adresse de départ et la taille du bloc. On pose donc:

Adresses $A = N$ (ordonnées avec incrément)

Blocs mémoires $B = A \times N$

Valeurs $V\oplus = B$

Mémoire $M = A \to N \oplus B$

Opérations Il nous faut pour *APS2* un mécanisme d'allocation d'une plage mémoire (intervalle ou bloc d'adresses). De plus l'opération de *restriction mémoire* doit désormais tenir compte des adresses allouées par bloc.

Allocation *allocn*, de type $S \times N \to (A \times S)$ telle que $allocn(\sigma, n) = (a, \sigma')$ si et seulement si, pour tout $i \in [0, n[$, $a + i$ n'est pas dans le domaine de σ et $\sigma' = \sigma[a = any; \ldots; a + n - 1 = any]$.

Restriction Pour définir l'ensemble des adresses accessibles depuis les identificateurs présents dans l'environnement, il faut non seulement prendre en compte les identificateurs liés à une adresse ($\rho(x) = inA(a)$), mais également ceux liés à des blocs mémoires ($\rho(x) = inB(a, n)$). Et il y a plus: il faut également considérer le cas où la valeur en mémoire est elle-même un bloc mémoire. Le calcul des adresses accessibles doit donc explorer l'environnement et la mémoire. Pour modéliser cela, on définit la suite récursive d'ensembles A_n:

$$\begin{aligned} A_0 &= \bigcup_{x \in \mathsf{ident}} \{a \mid \rho(x) = inA(a)\} \cup \bigcup_{x \in \mathsf{ident}} \bigcup_{i=0}^{n-1} \{a + i \mid \rho(x) = inB(a, n)\} \\ A_{n+1} &= \bigcup_{a \in A_n} \bigcup_{i=0}^{n-1} \{a' + i \mid \sigma(a) = inB(a', n)\} \end{aligned}$$

Intuitivement: on collecte dans A_0 toutes les adresses mentionnées dans l'environnement; puis ont collecte itérativement toutes les adresses mentionnées en mémoire.

L'ensemble des adresses accessibles est $AC = \bigcup_{i \in I\!N} A_i$. On définit (σ/ρ) comme la restriction de σ au domaine AC: $(\sigma/\rho)(a) = \sigma(a)$ si est seulement si $a \in AC$.

1.3.2 Expressions

De manière générale, dans *APS2*, l'évaluation d'une expression produit une valeur ainsi qu'un effet sur la mémoire. Ainsi la signature de $\vdash_{\text{Expr}}$ pour *APS2* est $E \times S \times \text{Expr} \times V \times S$.

On écrit $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (v, \sigma')$.

Le véritable effet sur la mémoire est produit par la primitive `alloc`:

(ALLOC) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inN(n), \sigma')$ et si $allocn(\sigma', n) = (a, \sigma'')$
alors $\rho, \sigma \vdash_{\text{Expr}}$ `(alloc` e`)` $\rightsquigarrow (inB(a, n), \sigma'')$

La primitive `nth` donne la valeur de l'un des éléments d'un bloc mémoire:

(NTH) si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow (inB(a, n), \sigma')$ et si $\rho, \sigma' \vdash_{\text{Expr}} e_2 \rightsquigarrow (inN(i), \sigma'')$
alors $\rho, \sigma \vdash_{\text{Expr}}$ `(nth` e_1 e_2`)` $\rightsquigarrow (\sigma''(a + i), \sigma'')$

La primitive `len` donne la longueur associée à un bloc mémoire:

(LEN) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inB(a, n), \sigma')$ alors $\rho, \sigma \vdash_{\text{Expr}}$ `(len` e`)` $\rightsquigarrow (inN(n), \sigma')$

Les autres règles de $\vdash_{\text{Expr}}$ doivent simplement être amendées pour tenir compte de la nouvelle signature de la relation:

(ID1) si $x \in \mathsf{ident}$, si $\rho(x) = inA(a)$ et si $\sigma(a) = v$ alors $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (v, \sigma)$

(ID2) si $x \in \mathsf{ident}$, si $\rho(x) = v$ et si $v \neq inA(a)$ alors $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (v, \sigma)$

Notez que dans (ID2), si la valeur associée à x dans l'environnement est un bloc, celui-ci est la valeur donnée par la relation.

(TRUE) $\rho, \sigma \vdash_{\text{Expr}}$ `true` $\rightsquigarrow (inN(1), \sigma)$

(FALSE) $\rho, \sigma \vdash_{\text{Expr}}$ `false` $\rightsquigarrow (inN(0), \sigma)$

(NUM) si $n \in \mathsf{num}$ alors $\rho, \sigma \vdash_{\text{Expr}} n \rightsquigarrow (inN(\nu(n)), \sigma)$

(PRIM) si $x \in \mathsf{oprim}$, si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow (inN(n_1), \sigma_1)$, ..., si $\rho, \sigma_{k-1} \vdash_{\text{Expr}} e_k \rightsquigarrow (inN(n_k), \sigma_k)$
et si $\pi(x)(n_1, \ldots, n_k) = n$
alors $\rho, \sigma \vdash_{\text{Expr}}$ `(`x $e_1 \ldots e_n$`)` $\rightsquigarrow (inN(n), \sigma_k)$

(IF1) si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow (inN(1), \sigma')$ et si $\rho, \sigma' \vdash_{\text{Expr}} e_2 \rightsquigarrow (v, \sigma'')$ alors $\rho, \sigma \vdash_{\text{Expr}}$ `(if` e_1 e_2 e_3`)` $\rightsquigarrow (v, \sigma'')$

(IF0) si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow (inN(0), \sigma')$ et si $\rho, \sigma' \vdash_{\text{Expr}} e_3 \rightsquigarrow (v, \sigma'')$ alors $\rho, \sigma \vdash_{\text{Expr}}$ `(if` e_1 e_2 e_3`)` $\rightsquigarrow (v, \sigma'')$

(ABS) $\rho, \sigma \vdash_{\text{Expr}}$ `[`x_1`:`t_1`,` $\ldots$ `,`x_n`:`t_n`]`$e \rightsquigarrow (inF(e, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n]), \sigma)$

(APP) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inF(e', r), \sigma')$,
si $\rho, \sigma' \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, \sigma_1)$, ..., si $\rho, \sigma_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, \sigma_n)$
et si $r(v_1, \ldots, v_n), \sigma_n \vdash_{\text{Expr}} e' \rightsquigarrow (v, \sigma'')$
alors $\rho, \sigma \vdash$ `(`e $e_1 \ldots e_n$`)` $\rightsquigarrow (v, \sigma'')$

(APPR) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inFR(\varphi), \sigma')$, si $\varphi(\varphi) = inF(e', r)$,
si $\rho, \sigma' \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, \sigma_1)$, ..., si $\rho, \sigma_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, \sigma_n)$
et si $r(v_1, \ldots, v_n), \sigma_n \vdash_{\text{Expr}} e' \rightsquigarrow (v, \sigma'')$
alors $\rho, \sigma \vdash$ `(`e $e_1 \ldots e_n$`)` $\rightsquigarrow (v, \sigma'')$

1.3.3 Déclarations

Le changement de $\vdash_{\text{Expr}}$ induit un changement dans la déclaration des constantes puisque, en particulier, lorsque l'on définit un tableau, on a un effet sur la mémoire. Pour *APS2*, la signature de la relation $\vdash_{\text{Dec}}$ est $E \times S \times \text{Dec} \times E \times S$.

(CONST) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (v, \sigma')$ alors $\rho, \sigma \vdash_{\text{Dec}} (\texttt{CONST}\ x\ t\ e) \rightsquigarrow (\rho[x = v], \sigma')$

1.3.4 Valeurs gauches

Le premier terme d'une affectation est soit un identificateur simple, soit la désignation d'un élément de tableau avec la primitive `nth`. Dans ces deux cas, il faut obtenir l'adresse visée par l'affectation. Le premier cas sera similaire à ce que nous avions pour *APS1*; dans le second cas, nous devons tenir compte du fait que le calcul de l'indice dans le tableau fait appel à l'évaluation d'une expression et peut donc avoir un effet sur la mémoire.

Nous appelons $\vdash_{\text{Lval}}$ la relation sémantique pour les «valeurs-gauches», elle a pour domaine
$E \times S \times \text{Lval} \times A \times S$.

On écrit $\rho, \sigma \vdash_{\text{Lval}} lv \rightsquigarrow (a, \sigma')$

Pour un identificateur, l'adresse cherchée doit être donnée par l'environnement et il n'y a alors pas d'effet sur la mémoire. Il faut considérer le cas où l'adresse est celle d'un bloc:

(LIDA) si $x \in \mathsf{ident}$ et si $\rho(x) = inA(a)$ alors $\rho, \sigma \vdash_{\text{Lval}} x \rightsquigarrow (a, \sigma)$

(LIDB) si $x \in \mathsf{ident}$ et si $\rho(x) = inB(a, n)$ alors $\rho, \sigma \vdash_{\text{Lval}} x \rightsquigarrow (a, \sigma)$

Dans le cas de l'usage de `nth`, il faut calculer la valeur de l'expression désignant l'indice visé, ce qui peut avoir un effet sur la mémoire:

(LNTH) si $\rho, \sigma \vdash_{\text{Expr}} lv \rightsquigarrow (inB(a, n), \sigma')$ et si $\rho, \sigma' \vdash_{\text{Expr}} e \rightsquigarrow (inN(i), \sigma'')$
alors $\rho, \sigma \vdash_{\text{Lval}} (\texttt{nth}\ lv\ e) \rightsquigarrow (a + i, \sigma'')$

Notez ici l'ordre dans lequel nous avons évalué les composants de $(\texttt{nth}\ lv\ e)$. Il eût pu être inverse.

1.3.5 Instructions

Le principal changement dans la définition de $\vdash_{\text{Stat}}$ est l'usage de $\vdash_{\text{Lval}}$ pour l'évaluation de l'affectation.

(SET) $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (v, \sigma')$ et si $\rho, \sigma' \vdash_{\text{Lval}} lv \rightsquigarrow (a, \sigma'')$ alors $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{SET}\ lv\ e) \rightsquigarrow (\sigma''[a := v], \omega)$

Pour les autres cas, il faut simplement tenir compte du changement de signature de $\vdash_{\text{Expr}}$:

(IF1) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inN(1), \sigma')$ et si $\rho, \sigma', \omega \vdash_{\text{Block}} bk_1 \rightsquigarrow (\sigma'', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{IF}\ e\ bk_1\ bk_2) \rightsquigarrow (\sigma'', \omega')$

(IF0) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inN(0), \sigma')$ et si $\rho, \sigma', \omega \vdash_{\text{Block}} bk_2 \rightsquigarrow (\sigma'', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{IF}\ e\ bk_1\ bk_2) \rightsquigarrow (\sigma'', \omega')$

(LOOP0) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inN(0), \sigma')$ alors $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{WHILE}\ e\ bk) \rightsquigarrow (\sigma', \omega)$

(LOOP1) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow (inN(1), \sigma')$, si $\rho, \sigma', \omega \vdash_{\text{Block}} bk \rightsquigarrow (\sigma'', \omega')$
et si $\rho, \sigma'', \omega' \vdash_{\text{Stat}} (\texttt{WHILE}\ e\ bk) \rightsquigarrow (\sigma''', \omega'')$
alors $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{WHILE}\ e\ bk) \rightsquigarrow (\sigma''', \omega'')$

(CALL) si $\rho(x) = inP(bk, r)$,
si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, \sigma_1)$, ..., si $\rho, \sigma_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, \sigma_n)$
et si $r(v_1, \ldots, v_n), \sigma_n, \omega \vdash_{\text{Expr}} bk \rightsquigarrow (\sigma', \omega')$
alors $\rho, \sigma, \omega \vdash (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (\sigma', \omega')$

(CALLR) si $\rho(x) = inFR(\varphi)$, si $\varphi(\varphi) = inP(bk, r)$,
si $\rho, \sigma \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1)$, ..., si $\rho, \sigma_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n)$
et si $r(v_1, \ldots, v_n), \sigma_n, \omega \vdash_{\text{EXPR}} bk \rightsquigarrow (\sigma', \omega')$
alors $\rho, \sigma, \omega \vdash (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (\sigma', \omega')$

(ECHO) si $\rho, \sigma \vdash_{\text{EXPR}} e \rightsquigarrow (inN(n), \sigma')$ alors $\rho, \sigma \vdash_{\text{STAT}} (\texttt{ECHO}\ e) \rightsquigarrow (\sigma', (n \cdot \omega))$

Notez également ici comment nous avons dû faire un choix dans l'ordre d'évaluation des composants d'une instruction.