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

P. Manoury

Janvier 2018

Contents

1 *APS1*: noyau impératif

On étend la capacité d'expression des calculs de *APS0* avec les traits des langages impératifs. Nous avions, dans *APS0* quelques traits impératifs: l'instruction d'affichage qui a un *effet* sur l'environnement sans avoir de valeur propre et l'enchaînement de calculs sous forme de *séquence* (suite) de commandes. Pour approcher de ce que l'on trouve dans les langages usuels, on ajoute à *APS0* les trois *instructions* de base des langages impératifs: l'affectation, la structure de contrôle alternative et une boucle.

Stricto sensu APS1 ne permettra pas de faire plus de calculs qu'*APS0* mais il donne plus de moyens pour exprimer la manière d'obtenir un résultats. En effet, du point de vue théorique, *APS0* est *Türing complet*; il permet de programmer toutes les fonctions *calculables* sur les entiers[1]– «en théorie» si l'on fait l'hypothèse, un peu irréaliste, que l'implantation d'un langage de programmation sait atteindre l'infinité des valeurs entières.

L'ajout le plus significatif est celui de l'instruction d'affectation. L'utilisation de l'affectation dans un programme rompt ce que l'on appele la *transparence référentielle* des identificateurs. En effet, dans *APS0* on ne peut définir que des *constantes* et en lisant le code source d'un programme on peut connaître la valeur associées aux noms des constantes car le langage ne fournit aucun moyen de *modifier* l'association entre noms et valeurs. En revanche, dans *APS1* l'instruction d'affectation a précisément ce rôle: modifier la valeur associée à un nom. Ainsi, dans *APS1* nous aurons réellement une notion de *variable*.

Nous signifierons dans notre langage cette différence de traitement des noms en introduisant une clause spécifique de déclaration pour les variables.

Nous avions doté *APS0* de la capacité de définir des *fonctions* permettant d'enrichir le lanage des expressions. Nous doterons *APS1* de la capacité de définir des *procédures* permettant, en quelque sorte, d'enrichir le langage des instructions. Nous reviendrons, avec *APS3* sur ces notions de *fonctions* et de *procédures*.

1.1 Syntaxe

1.1.1 Lexique

L'extension du lexique de *APS0* pour obtenir *APS1* consiste simplement à ajouter des mots clef pour les définitions de variables et de procédures, les instructions d'affectation, d'alternatives, de boucle et d'appel de procédure.

Mots clef `VAR PROC`
`SET IF WHILE CALL`

1.1.2 Grammaire

L'extension de la grammaire est essentiellement réalisée au niveau des déclarations (Dec) et des instructions (Stat). Nous ajouterons également une nouvelle constante de type : `void`. En effet, nous voulons permettre le définition de procédure *d'ordre supérieur* pouvant accepter des procédures en arguments.

Dec	::=	...
	|	`VAR` ident Type
	|	`PROC` ident `[` Args `]` Prog
	|	`PROC REC` ident `[` Args `]` Prog
Stat	::=	...
	|	`SET` ident Expr
	|	`IF` Expr Prog Prog
	|	`WHILE` Expr Prog
	|	`CALL` ident Exprs
Type	::=	...
	|	`void`

[1] L'expression *Türing complet* vient du nom d'Allan Türing qui fut le premier à proposer une définition logico-mathématique du concept de *fonction calculable*.

Notez que nous avons choisi de

- ne pas forcer l'initialisation d'une variable au moment de sa déclaration;
- n'avoir qu'une alternative *bilatère* (pas de «if» sans «else»);
- de n'avoir qu'un seul type de boucle.

Nous n'avons pas trouvé utile d'introduire des *procédures anonymes*.

L'instruction alternative `IF` (en capitale) ne doit pas être confondue avec l'opérateur d'expression `if` (en minuscule) ce dernier est l'analogue de la construction e_1 `?` e_2 `:` e_3 du langage C. Dans *APS1* il n'est pas possible d'utiliser l'une à la place de l'autre: il y a une barrière étanche d'usage entre les expressions et les instructions.

Notez enfin que notre grammaire autorise les déclarations dans le corps des procédures ainsi que dans les suites de commandes associées aux instructions d'alternative et de boucle (non terminal PROG). Nous appelerons *blocs* ces occurences de suites de commandes encloses entre crochets. Nous verrons que la sémantique leur réservera un traitement particulier.

1.2 Typage

Puisque nous l'avons introcit dans le langage des types de *APS1*, nous utiliserons `void` dans nos règles de typage. En particulier, les procédures seront analysées comme des fonctions dont le «type de retour» est `void`.

Pour obtenir les règles de typage de *APS1*, il suffit d'étendre la définition des relations de tyapges $\vdash_{\text{DEC}}$ et $\vdash_{\text{STAT}}$ pour les nouvelles constructions syntaxiques.

1.2.1 Déclarations

(VAR) $\Gamma \vdash_{\text{DEC}} (\texttt{VAR}\ x\ t) : \Gamma[x : t]$

(PROC) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n] \vdash_{\text{CMDS}} (cs\texttt{;}\varepsilon) : \texttt{void}$
alors $\Gamma \vdash_{\text{DEC}} (\texttt{PROC}\ x\ \texttt{[}x_1\texttt{:}t_1, \ldots, x_n\texttt{:}t_n\texttt{]}\ \texttt{[}cs\texttt{]}) : \Gamma[x : t_1\ \texttt{*}\ \ldots\ \texttt{*}\ t_n\ \texttt{->}\ \texttt{void}]$

(PROCREC) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n; x : t_1\ \texttt{*}\ \ldots\ \texttt{*}\ t_n\ \texttt{->}\ \texttt{void}] \vdash_{\text{CMDS}} (cs\texttt{;}\varepsilon) : \texttt{void}$
alors $\Gamma \vdash_{\text{DEC}} (\texttt{PROC REC}\ x\ \texttt{[}x_1\texttt{:}t_1, \ldots, x_n\texttt{:}t_n\texttt{]}\ \texttt{[}cs\texttt{]}) : \Gamma[x : t_1\ \texttt{*}\ \ldots\ \texttt{*}\ t_n\ \texttt{->}\ \texttt{void}]$

1.2.2 Instructions

(SET) si $\Gamma(x) = t$ et si $\Gamma \vdash_{\text{EXPR}} e : t$ alors $\Gamma \vdash_{\text{STAT}} (\texttt{SET}\ x\ e) : \texttt{void}$

(IF) si $\Gamma \vdash_{\text{EXPR}} e : \texttt{bool}$, si $\Gamma \vdash_{\text{CMDS}} cs_1 : \texttt{void}$ et si $\Gamma \vdash_{\text{CMDS}} cs_2 : \texttt{void}$ alors $\Gamma \vdash_{\text{STAT}} (\texttt{IF}\ e\ \texttt{[}cs_1\texttt{]}\ \texttt{[}cs_2\texttt{]}) : \texttt{void}$

(WHILE) si $\Gamma \vdash_{\text{EXPR}} e : \texttt{bool}$ et si $\Gamma \vdash_{\text{CMDS}} cs : \texttt{void}$ alors $\Gamma \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ \texttt{[}cs\texttt{]}) : \texttt{void}$

(CALL) si $\Gamma(x) = t_1\ \texttt{*}\ \ldots\ \texttt{*}\ t_n\ \texttt{->}\ \texttt{void}$, si $\Gamma \vdash_{\text{EXPR}} e_1 : t_1$, ... et si $\Gamma \vdash_{\text{EXPR}} e_n : t_n$
alors $\Gamma \vdash_{\text{STAT}} (\texttt{CALL}\ x\ e_1 \ldots e_n) : \texttt{void}$

1.3 Sémantique

Dans *APS0*, nous n'avions qu'un seul *effet*: celui de l'instruction d'affichage. Dans *APS1* nous en avons un nouveau: celui de l'affectation.

Dans la sémantique de *APS0*, nous avions réalisé la liaison entre noms (de constantes) et valeurs avec la structure d'environnement. Et, dans *APS0*, cette liaison est *statique*: elle n'est jamais modifiée lors du processus d'évaluation – il ne faut pas confondre ici la modification d'une liaison avec sa *redéfinition*.

Nous souhaitons conserver le caractère statique des liaisons réalisée dans les environnements et pour réaliser le caractère *dynamique* des liaisons modifiables par l'affectation, nous introduisons dans les domaines

sématique une nouvelle structure que nous appeleon la *mémoire*. Celle-ci n'établit pas directement une liaison entre noms et valeurs, mais plutôt une liaison netre *adresses* et valeurs. Ainsi, un nom de varaible sera statiquement lié à une adresse (dans l'environnement) et les adresses seront dynamqiuement liées à des valeurs (dans la mémoire). Ce choix de modélisation s'approche des modèles d'exécution des langages où une zone mémoire est *allouée* aux variables des programmes.

Les variables peuvent naturellemnt intervenir dans les expressions. La sémantique des expression devra donc tenir compte des valeurs présentes en mémoire, en particulier lorsque l'expression est réduit à un identificateur (constante ou variable).

Pour ce qui est des procédure, nous introduisons une notion de *fermeture procédurale* très proche de la notion de fermeture que nous avions utilisée pour la sémantique du noyau fonctionnel.

Enfin, nous introduirons, pour définir notre sématique, une vision particulière des suites de commandes: les *blocs*. Ceux correspondront aux suites de commandes associées aux instructions d'alternative et de boucle ainsi que celles associées aux définitions de procédures.

1.3.1 Domaines et opérations sémantiques

Domaines Nous nous donnons un domaine d'adresses abstrait. Pour *APS1*, une mémoire est simplment une fonction (partielle) des adresses vers les entiers.

Adresse A

Fermetures procédurales $P = \textsc{Cmds} \times (V^* \to E)$

Fermetures procédurales récursives $PR = PR \to P$

Valeurs $V\oplus = A \oplus P \oplus PR$

Mémoire $S = A \to N$ (fonction partielle)

Les remarques que nous avions faites sur la définition des fermetures récursives FR valent également pour PR.

Valeurs et opérations Pour modéliser le mécanisme de gestion de la mémoire, nous devons modéliser celui de l'*allocation* d'une nouvelle cellule en mémoire. N'ayant pas précisé trop la structure de la mémoire, nous modélisons l'allocation de manière *axiomatique*: nous décrivons simplement ce que l'on entend par une *nouvelle adresse*. Outre l'opération d'extension de la mémoire, nous devons également poser la définition de la *modification* d'une liaison en mémoire.

Enfin, les blocs associés aux déclarations de procédures ou aux instruction de boucle ou d'alternative peuvent contenir des déclarations que nous voulons considérées comme *locales* aux blocs. Pour modéliser cela, nous nous donnons une opération de *restriction* de la mémoire que l'on peut voire comme une opération de libération mémoire: lorsque le fil d'exécution «sort» d'un bloc, les variables locales allouées sont libérées et leur espace mémoire peut être réalloué.

Mémoire vide $\emptyset$, fonction jamais définie

Valeur indéterminée *any*

Extension mémoire fonction notée $\sigma[a = any]$ avec σ élément de S et a, élément de A

Allocation *alloc* de type $S \to (A \times S)$ telle que $alloc(\sigma) = (a, \sigma')$ si et seulement si a n'est pas dans le domaine de σ (nouvelle adresse) et $\sigma' = \sigma[a = any]$

Modification mémoire fonction de type $S \to A \to N \to S$, notée $\sigma[a := v]$ telle que $\sigma[a = v'][a := v] = \sigma[a = v]$ et $\sigma[a' = v'][a := v] = \sigma[a := v][a' = v']$ lorsque a est différent de a'. Notez que $\emptyset[a := v]$ n'est pas défini.

Restriction mémoire: restriction de σ aux valeurs allouées dans ρ: fonction de type $S \to E \to S$, notée (σ/ρ) telles que $(\sigma/\rho)(a) = \sigma(a)$ si et seulement il existe x tel que $\rho(x) = inA(a)$.

Un *contexte d'évaluation* est formé d'un environnement et d'une mémoire, ainsi que d'un flot de sortie.

1.3.2 Expressions

Pour tenir compte des valeurs en mémoire, la relation d'évaluation des expressions $\vdash_{\text{Expr}}$ change de signature. celle-ci devient: $E \times S \times \text{Expr} \times V$

On écrit $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow v$

Le changement de signature nous obliqge à réécrire toutes les régles d'évalaution des expressions. Toutefois, le seul changement important concerne l'évaluation des identificateurs pour lesquels nous posons deux règles: une pour les constantes, l'autre pour les variables.

La valeur d'une variable est celle que l'on trouve en mémoire à l'adresse qui lui a été assignée.

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

Pour les autres catégories d'identificateurs, on prend ce qui est donné par l'environnement

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

La relation sémantiques des autres expressions est similaire à celle que nous avions donnée dans *APS0* à ce changement près que le contexte d'évaluation inclus maintenant la mémoire:

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

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

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

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

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

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

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

(APP) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow inF(e', r)$,
si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow v_1$, ..., si $\rho, \sigma \vdash_{\text{Expr}} e_n \rightsquigarrow v_n$ et si $r(v_1, \ldots, v_n), \sigma \vdash_{\text{Expr}} e' \rightsquigarrow v$
alors $\rho, \sigma \vdash \texttt{(}e\ e_1 \ldots e_n\texttt{)} \rightsquigarrow v$

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

1.3.3 Déclarations

Dans la mesure où les déclarations de constantes encapsulent des expressions, la relation sémantique des déclaration $\vdash_{\text{Dec}}$ doit également tenir compte de la mémoire et changer de signature. Plus encore, la déclaration d'une variable doit associer au nom de la variable une adresse mémoire pour y stocker la valeur que l'on souhaite associer à cette variable. Cette adresse doit naturellement être «nouvelle». Une déclaration de variable a donc un effet sur la mémoire dont la sémantique doit rendre compte.

De manière générale, une déclaration produit un nouveau contexte d'évaluation à partir d'un ancien. Ainsi, pour *APS1*, la signature de $\vdash_{\text{Dec}}$ devient: $E \times S \times \text{Dec} \times E \times S$.

On écrit $\rho, \sigma \vdash_{\text{DEC}} d \rightsquigarrow (\rho', \sigma')$

La déclaration d'une variable ajoute une liaison entre nom et adresse dans l'environnement et étend la mémoire:

(VAR) si $alloc(\sigma) = (a, \sigma')$ alors $\rho, \sigma \vdash_{\text{DEC}} (\texttt{VAR}\ x\ t) \rightsquigarrow (\rho[x = inA(a)], \sigma')$

Les déclarations de procédures et de procédures récursives engendrent des fermeutures procédurales et des fermetures porcédurales récursives qui sont associées aux noms des procédures dans l'environnement:

(PROC) $\rho, \sigma \vdash_{\text{DEC}} (\texttt{PROC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ bk) \rightsquigarrow (\rho[x = inP(bk, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n]), \sigma)$

(PROCREC) $\rho, \sigma \vdash_{\text{DEC}} (\texttt{PROC REC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ bk)$
$\rightsquigarrow (\rho[x = inPR(\lambda f.inP(bk, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n][x = f]), \sigma)$

La sémantique des déclarations qui existaient dans *APS0* sont amendées pour tenir compte de la nouvelle forme des contextes d'évaluation:

(CONST) si $\rho, \sigma \vdash_{\text{EXPR}} e \rightsquigarrow v$ alors $\rho, \sigma \vdash_{\text{DEC}} (\texttt{CONST}\ x\ t\ e) \rightsquigarrow (\rho[x = v], \sigma)$

(FUN) $\rho, \sigma \vdash_{\text{DEC}} (\texttt{FUN}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ e) \rightsquigarrow (\rho[x = inF(e, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n]), \sigma)$

(FUNREC) $\rho, \sigma \vdash_{\text{DEC}} (\texttt{FUN REC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ e)$
$\rightsquigarrow (\rho[x = inFR(\lambda f.inF(e, \lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n][x = f]), \sigma)$

1.4 Les instructions

Au premier rang des instructions se trouve l'affectation. C'est elle qui définit le caractère impératif du langage, c'est donc elle qui guide ce que doit être la sémantique des instructions. À l'instar de l'instruction d'affichage qui ne produit pas de valeur mais a un *effet* sur le flux de sortie, l'affectation ne produit pas de valeur, mais à une *effet* sur la mémoire: *modifier* l'association entre une adresse et une valeur. Ainsi, l'affectation relie un état du contexte d'évaluation (dont la mémoire) à un autre état où la mémoire est *affectée*.

De manière génarale, la relation sémantique $\vdash_{\text{STAT}}$ pour les instruction aura donc la signature $E \times S \times O \times \text{STAT} \times S \times O$.

On écrit $\rho, \sigma, \omega \vdash_{\text{STAT}} s \rightsquigarrow (\sigma', \omega')$

L'affectation remplace la valeur contenue à l'adresse associée à un identificateur avec la valeur obtenue par évaluation de l'expression mentionnée par l'instruction:

(SET) si $\rho(x) = inA(a)$ et si $\rho, \sigma \vdash_{\text{EXPR}} e \rightsquigarrow v$ alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{SET}\ x\ e) \rightsquigarrow (\sigma[a := v], \omega)$

Notez que la relation n'est pas définie lorsque x n'a pas été déclaré comme une variable.

L'instruction d'alternative est interprétée comme sont *alter ego* fonctionnel. Son effet sera celui de l'une ou (exclusif) l'autre de ses branches selon la valeur de la condition:

(IF1) si $\rho, \sigma \vdash_{\text{EXPR}} e \rightsquigarrow inN(1)$ 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 $\rho, \sigma \vdash_{\text{EXPR}} e \rightsquigarrow inN(0)$ et si $\rho, \sigma, \omega \vdash_{\text{BLOCK}} bk_2 \rightsquigarrow (\sigma', \omega')$ alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{IF}\ e\ bk_1\ bk_2) \rightsquigarrow (\sigma', \omega')$

L'instruction de boucle est également définie par deux clauses, selon la valeur de sa condition: si celle-ci est fausse, l'instruction n'a aucun effet; sinon, l'instruction a pour premeir effet celui du *bloc* (suite de commandes) associé à la boucle puis celui de la boucle elle-même dans le contexte ainsi modifié:

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

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

Notez qu'ici notre définition *n'est pas bien fondée* en général puisque l'effet de `WHILE` e bk est défini en fonction de celui de `WHILE` e bk. Dans la pratique, c'est le contexte produit par une itération de la boucle qui permettra ou non la «sortie» de boucle, mais en génaral, il est impossible, dans notre langage, de garantir cette «sortie».

L'appel de procédure et de procédure récursive est proche de l'appel de fonction:

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

(CALLR) si $\rho(x) = inPR(\varphi)$, si $\varphi(inPR(\varphi)) = inP(bk, r)$,
si $\rho, \sigma \vdash_{\text{Expr}} e_1 \rightsquigarrow v_1$, ..., si $\rho, \sigma \vdash_{\text{Expr}} e_n \rightsquigarrow v_n$
et si $r(v_1, \ldots, v_n), \sigma, \omega \vdash_{\text{Expr}} bk \rightsquigarrow (\sigma', \omega')$
alors $\rho, \sigma, \omega \vdash (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (\sigma', \omega')$

Pour finir, la relation sémantique pour l'instruction d'affichage doit être amendée pour prendre en compte la nouvelle forme des contexte d'évaluation:

(ECHO) si $\rho, \sigma \vdash_{\text{Expr}} e \rightsquigarrow inN(n)$ alors $\rho, \sigma \vdash_{\text{Stat}} (\texttt{RETURN}\ e) \rightsquigarrow (\sigma, (n; \omega))$

1.4.1 Les suites de commandes

Peu de choses changent pour la sémantique des suites de commandes, si ce n'est le contexte d'évaluation (environnement et mémoire – on peut ignorer ici le flux de sortie).

La relation $\vdash_{\text{Cmds}}$ a pour signature $E \times S \times O \times (\text{Cmds}_\varepsilon) \times S \times O$.
On écrit $\rho, \sigma, \omega \vdash_{\text{Cmds}} cs \rightsquigarrow (\sigma', \omega')$.

(DECS) si $\rho, \sigma \vdash_{\text{Dec}} d \rightsquigarrow (\rho', \sigma')$ et si $\rho', \sigma', \omega \vdash_{\text{Cmds}} cs \rightsquigarrow (\sigma'', \omega')$ alors $\rho, \omega \vdash_{\text{Cmds}} (d\texttt{;}\ cs) \rightsquigarrow (\sigma'', \omega')$

(STATS) si $\rho, \sigma, \omega \vdash_{\text{Stat}} s \rightsquigarrow (\sigma', \omega')$ et si $\rho, \sigma', \omega' \vdash_{\text{Cmds}} cs \rightsquigarrow (\sigma'', \omega'')$ alors $\rho, \sigma, \omega \vdash_{\text{Cmds}} (s\texttt{;}\ cs) \rightsquigarrow (\sigma'', \omega'')$

(END) $\rho, \sigma, \omega \vdash_{\text{Cmds}} \varepsilon \rightsquigarrow (\sigma, \omega)$

1.4.2 Blocs de commandes

Les blocs de commandes correspondent aux suite de commandes associées aux instruction d'alternative et de boucle ainsi qu'aux corps des procédures. Nous leur réservons un traitement particulier car nous voulons donner aux déclarations pouvant intervenir dans ces suites un caractère *local*. Cette manière a deux conséquences:

1. Les identificateur déclarés dans un bloc ne sont plus accessibles en dehors du bloc.
2. La mémoire allouée par une déclaration figurant dans un bloc peut être «libérée» lorsque le bloc n'est plus actif.

C'est pour modéliser ce second effet que nous avons introduit l'opération de restriction sur la mémoire : (σ/ρ).

La relation $\vdash_{\text{Block}}$ a pour signature $E \times S \times 0 \times \text{Prog} \times S \times O$
On écrit $\rho, \sigma, \omega \vdash bk \rightsquigarrow (\sigma', \omega')$

(BLOCK) si $\rho, \sigma, \omega \vdash_{\text{Cmds}} cs\texttt{;}\varepsilon \rightsquigarrow (\sigma', \omega')$ alors $\rho, \sigma, \omega \vdash_{\text{Block}} \texttt{[}cs\texttt{]} \rightsquigarrow ((\sigma'/\rho), \omega')$

1.4.3 Programme

L'évaluation d'un programme est l'évaluation de la suite de commandes qui le compose. Le «résultat» de l'évalaution d'un programme sera l'effet produit sur le flot de sortie et sur la mémoire.

La relation $\vdash$ a pour signature $\textsc{Prog} \times S \times O$.

On écrit $\vdash \texttt{[}cs\texttt{]} \rightsquigarrow (\sigma, \omega)$.

(PROG) si $\emptyset, \emptyset, \emptyset \vdash_{\textsc{Cmds}} cs\,\texttt{;}\,\varepsilon \rightsquigarrow (\sigma, \omega)$ alors $\vdash \texttt{[}cs\texttt{]} \rightsquigarrow (\sigma, \omega)$.