SU/FS/master/info/MU4IN503 APS
Notes de cours

P. Manoury

Janvier 2022

Contents

5 APS3: fonctions procédurales

Jusqu'ici, nous avons maintenu une barrière assez étanche entre l'aspect fonctionnel du langage (les *expressions* de *APS0*) et son aspect impératif (les *instructions* de *APS1*). On peut lire cette séparation dans la distinction entre les définitions de *fonctions* qui ne font appel qu'à des expressions et celles des *procédures* qui font appel à des suites d'instructions (plus généralement, des blocs de commandes). Dans *APS1* (qui contient *APS0*), il y a donc une stricte séparation entre le monde des *valeurs*, produites par les expressions, et celui des *effets* produits par les instructions (affichage et affectation).

En élaborant *APS2*, nous avons fait une première petite entorse à notre discipline avec les primitives `alloc` et `vset` qui sont des constructeurs d'expression ayant un effet sur la mémoire. Nous allons, dans *APS3* abandonner radicalement cette discipline en offrant la possibilité de définir une fonction dont le corps n'est pas une expression, mais une suite de commandes. Celle-ci doit posséder une propriété particulière: produire une valeur. L'ajout essentiel de l'extension *APS3* est celui d'une *commande* capable de délivrer une valeur, à savoir: `RETURN`.

Si l'impact de cette extension est syntaxiquement assez faible, il n'en va pas de même pour le typage et la sémantique qui devront être assez profondément revisités. En effet, avec `RETURN`, nous avons une commande qui produit une valeur, par conséquent, une suite de commandes, et donc une instruction composée (comme l'alternative et la boucle), seront également suceptibles de produire une valeur.

5.1 Syntaxe

La commande *return* a un statut hybride. En tant que donnant une valeur, elle s'apparente aux expressions. Mais sa place attendue dans une suite de commandes l'apparie aux instructions. Nous lui donnerons donc un statut à part dans la syntaxe.

Nous complétons cet ajout par de nouvelles clauses de définition de fonctions et de fonctions récursives.

5.1.1 Lexique

Mot clef `RETURN`

5.1.2 Grammaire

CMDS	::=	...
	|	RET
RET	::=	`RETURN` EXPR
DEC	::=	...
	|	`FUN` ident TYPE `[` ARGS `]` `[` CMDS `]`
	|	`FUN REC` ident TYPE `[` ARGS `]` `[` CMDS `]`

Notez comment, dans CMDS, nous forçons `RETURN` à ne figurer qu'en fin de suites de commandes. Cette restriction de l'occurrence de `RETURN` en fin de suite est motivée par l'intention sémantique attachée à la commande `RETURN` qui est de délivrer une valeur en rompant la séquentialité. Faire suivre un `RETURN` d'autres commandes n'a dès lors pas d'objet puisque ces dernières ne seraient pas évaluées. Elles constitueraient du *code mort*.

Toutefois, cette tentative de contrôle syntaxique du code mort n'est pas complète. En effet, le (fragment de) programme

```
IF (eq x 0) [RETURN 0] [RETURN 1]; SET x 42
```

est syntaxiquement acceptable et il comporte néanmoins du code mort: l'affectation `SET x 42`. Nous verrons comment traiter ce point au cours de *l'analyse statique* de type.

Une autre difficulté est le cas d'une instruction alternative dont l'une des branches se termine par `RETURN` et l'autre non, comme dans

```
IF (eq x 0) [SET x 1] [RETURN 1]
```

Là encore, l'analyse statique de type nous aidera à définir les suites acceptées. Ce qui vaut pour l'alternative vaudra également pour les boucles.

Nota Bene: on pourrait abandonner les clauses de définition `PROC` et `PROC REC` pour les remplacer par des définitions de fonctions dont le *type de retour* est `void`. Nous ne le ferons pas.

5.2 Typage

5.2.1 Typage et analyse de flot

Puisqu'une suite de commandes peut produire une valeur, elle n'est plus nécessairement de type `void`. Une suite de commandes qui s'achève par un `RETURN` e sera sensée avoir le type de l'expression e.

Mais, une suite de commandes n'a pas qu'une seule manière de «s'achever par un `RETURN` e». Par exemple, une suite de commande s'achevant pas `IF` e `[ RETURN 1 ] [ RETURN 0]` exécute un `RETURN` quelle que soit la valeur de e; toute exécution de cette suite s'achève donc bien par un `RETURN`.

On serait alors tenté de poser comme règle de typage de *l'instruction alternative* `IF` une règle analogue à celle de *l'alternative fonctionnelle* `if`: les deux branches de l'alternative doivent avoir le même type: `void`, `bool`, `int`, etc.

Avec une telle décision, l'instruction `IF` e `[ RETURN true ] [ RETURN 0 ]` n'est pas typable, ce qui est légitme. Mais alors `IF` e `[ RETURN true ] [ SET x (add x 1) ]` n'est pas typable non plus. Si cette instruction est la dernière d'une séquence, notre décision est légitime. Toutefois, dans le contexte d'une suite de commandes, on peut toujours «rattraper» la lacune de la branche négative de cette alternative en la faisant suivre d'un `RETURN`:

```
IF  e [ RETURN true ] [ SET x (add x 1) ];
RETURN false
```

Toute cette séquence délivre une valeur de type `bool`. Elle est acceptable. Calquer le typage de l'instruction d'alternative sur celle de l'alternative fonctionnelle semble donc excessif.

Considérons à présent le cas de l'instruction de boucle. Quel type donner à l'instruction

```
WHILE  e [ RETURN true ]
```

Ce peut être `bool`, si e prend la valeur `true`, mais pas si e a la valeur `false`. Dans ce cas, l'instruction `WHILE` e `[ RETURN true ]` est assimilée à «l'instruction vide», qui n'a aucun effet. Son type naturel serait plutôt `void`.

Mélant les questions posées par les instructions d'alternative et de boucle, on a la séquence suivante:

```
SET x a;
WHILE (lt x b)
  [ IF (eq (f x) 0)
       [ RETURN true ]
       [ SET x (add x 1) ] ];
RETURN false
```

Ce fragment de code donnera `true` ou `false` selon qu'une fonction `f` s'annule ou non sur un intervalle délimité par les valeurs de `a` et `b`. C'est un code syntaxiquement et opérationnellement acceptable et nous voudrions que l'analyse de type l'accepte.

Si l'on y regarde d'un peu plus près, notre dilemne n'est pas général:

1. Une séquence de commandes doit avoir un type, soit `void`, soit n'importe quel autre.
2. On peut accepter `IF` e bk_1 bk_2 lorsque bk_1 et bk_2 ont le même type (soit `void`, soit n'importe quel autre).

3. On veut accepter `IF` e bk_1 bk_2 lorsque bk_1 ou bk_2 ont l'un ou l'autre le type `void`.

4. On peut rejeter `IF` e bk_1 bk_2 lorsque bk_1 et bk_2 ont deux tyoes diférents dont aucun n'est `void`.

5. Le type de `WHILE` e bk a pour type soit `void`, soit un autre type.

L'analyse de type devra donc savoir traiter des instructions (dont des séquences) dont le type est *non complètement déterminé*: `void` ou n'importe quel $t \neq$ `void`.

On notera $t +$ `void`. On pose que `void` + `void` = `void`, d'où $t +$ `void` + `void` $= t +$ `void`.

5.2.2 Expressions

Les règles d'analyse de type pour *APS3* restent identiques à celles que l'on avait dans *APS2*. L'application d'une fonction procédurale ne se distingue pas de l'application d'une fonction «pure».

5.2.3 Instructions

Pour l'alternative, nous devons ouvrir la possibilité de lui attribuer un *type mixte*.

(IF0) pour tout type t, si $\Gamma \vdash_{\text{Expr}} e$: `bool` et $\Gamma \vdash_{\text{Block}} blk_1 : t$ et $\Gamma \vdash_{\text{Block}} blk_2 : t$
alors $\Gamma \vdash_{\text{Stat}}$ (`IF` e blk_1 blk_2) : t

(IF1) pour tout $t \neq$ `void`, si $\Gamma \vdash_{\text{Expr}} e$: `bool` et $\Gamma \vdash_{\text{Block}} blk_1$: `void` et $\Gamma \vdash_{\text{Block}} blk_2 : t$
alors $\Gamma \vdash_{\text{Stat}}$ (`IF` e blk_1 blk_2) : $t +$ `void`

(IF2) pour tout $t \neq$ `void`, si $\Gamma \vdash_{\text{Expr}} e$: `bool` et $\Gamma \vdash_{\text{Block}} blk_1 : t$ et $\Gamma \vdash_{\text{Block}} blk_2$: `void`
alors $\Gamma \vdash_{\text{Stat}}$ (`IF` e blk_1 blk_2) : $t +$ `void`

La boucle a nécessairement un *type mixte* car le corps d'une boucle peut ne pas être exécuté.

(WHILE) pour tout type t, si $\Gamma \vdash_{\text{Expr}} e$: `bool` et $\Gamma \vdash_{\text{Block}} blk : t$ alors $\Gamma \vdash_{\text{Stat}}$ (`WHILE` e blk) : $t +$ `void`

Remarquons qu'ici, on peut très bien avoir que $t =$ `void`, dans ce cas, le type de la boucle est simplement `void` en vertu de `void` = `void` + `void`.

Les règles de typage de l'instruction d'affichage, de l'affectation et des appels de procédures ne changent pas.

On limite l'appel de procédures aux «vraies» procédures: celles qui ne produisent pas de valeurs:

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

Cette limitation n'est pas absolument nécessaire. Toutefois, une séquence dont un membre intermédiaire n'est pas purement impératif (*i.e.* produit une valeur) est souvent l'indice d'une rerreur ou d'une imprécision dans le code. D'ailleurs, les compilateurs engendre souvent un avertissemnt dans de tels cas.

5.2.4 Déclarations

On ajoute à l'analyse de type des déclarations de *APS2* le cas des fonctions procédurales

(FUNP) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n] \vdash_{\text{Block}} bk : t$
alors $\Gamma \vdash_{\text{Def}}$ (`FUN` x t `[`x_1`:`t_1`,` ... `,`x_n`:`t_n`]` bk) : $\Gamma[x : t_1$ `*` ... `*` t_n `->` $t]$

(FUNRECP) si $\Gamma[x_1 : t_1; \ldots; x_n : t_n; x : t_1$ `*` ... `*` t_n `->` $t] \vdash_{\text{Block}} bk : t$
alors $\Gamma \vdash_{\text{Def}}$ (`FUN REC` x t `[`x_1`:`t_1`,` ... `,`x_n`:`t_n`]` bk) : $\Gamma[x : t_1$ `*` ... `*` t_n `->` $t]$

5.2.5 Suite de commandes, dont RETURN

Il faut réaliser ici notre première exigence: une suite de commande peut avoir le type `void` ou n'importe quel autre type. Nous ne perdons également pas de vue notre objectif de détecter le code mort. Ceci ne concerne que l'occurence des instructions ou la commande `RETURN` dans les suites de commandes. La règle des suites commençant par une déclaration ne change pas par rapport à *APS2*.

(STAT0) pour tout type t, si $\Gamma \vdash_{\text{STAT}} s : \mathsf{void}$ et $\Gamma \vdash_{\text{CMDS}} cs : t$ alors $\Gamma \vdash_{\text{CMDS}} (s;cs) : t$

(STAT1) si $t \neq \texttt{void}$, si $\Gamma \vdash_{\text{STAT}} s : t + \mathsf{void}$ et $\Gamma \vdash_{\text{CMDS}} cs : t$ alors $\Gamma \vdash_{\text{CMDS}} (s;cs) : t$

(RET) Si $\Gamma \vdash_{\text{EXPR}} e : t$ alors $\Gamma \vdash_{\text{CMDS}} (\texttt{RETURN}\ e) : t$

(END) si $\Gamma \vdash_{\text{STAT}} s : t$ alors $\Gamma \vdash_{\text{CMDS}} (s) : t$

Ces règles éliminent bien le code mort en interdisant les instructions qui ne terminent pas une suite de commande est dont le type n'est ni `void`, ni $t + \texttt{void}$. En cons'équence et par exemple, une instruction du genre `IF` e `[ RETURN` e_1 `] [ RETURN` e_2 `]` ne peut se trouver qu'en fin de séquence.

Pour les définitions, il suffit de généraliser la règle à un type quelconque:

(DEF) si $d \in \text{DEC}$, si $\Gamma \vdash_{\text{DEF}} d : \Gamma'$, si $\Gamma' \vdash_{\text{CMDS}} cs : t$
alors $\Gamma \vdash_{\text{CMDS}} (d;cs) : t$.

5.3 Sémantique

D'avoir ajouté que le corps d'une fonction puisse être une suite de commandes implique que l'application de telles fonctions est susceptible d'effet sur la mémoire et le flot de sortie et qu'une suite de commandes peut avoir une valeur. Ainsi, la signature des relations sémantiques pour le fragment fonctionnel et pour le fragment impératif se rejoignent: les constructions fonctionnelles et impératives produisent toutes une valeur, un effet mémoire et un effet sur le flot de sortie.

Toutefois, certaines suites de commandes peuvent rester purement impératives et ne pas produire de valeur exploitée par le programme. Pour celà, nous complétons l'ensemble des valeurs avec une *valeur vide* que nous noterons ε[1]. Cette pseudo valeur nous servira à détecter l'occurence de l'évaluation d'un `RETURN` dans une suite de commandes.

On peut noter que, dans un programme bien typé, une expression ne peut jamais produire une *valeur vide*.

5.3.1 Domaines sémantiques

Nous n'avons qu'un ajout aux domaines sémantiques de *APS2*: $V_\varepsilon = V \cup \{\varepsilon\}$. Les valeurs des fonctions procédurales seront les fermetures procédurales que nous avions déjà dans *APS1*.

5.3.2 Expressions

Dans *APS3*, le domaine de la relation $\vdash_{\text{EXPR}}$ est $E \times S \times O \times \text{EXPR} \times (V \times S \times O)$.

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

Toutes les règles sémantiques d'évaluation des expressions de *APS2* doivent être amendées pour tenir compte de la nouvelle signature de $\vdash_{\text{EXPR}}$. Les seules règles propres à *APS3* sont celles définissant l'application de fonctions procédurales qui impliquent l'évaluation d'une suite de commandes.

(APP') si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (inP(bk, (x_1, \ldots, x_n), \rho'), \sigma', \omega')$,
si $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1)$, $\ldots$, si $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$
et si $\rho'[x_1 = v_1, \ldots, x_n = v_n], \sigma_n, \omega_n \vdash_{\text{BLOCK}} bk \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash (e\ e_1 \ldots e_n) \rightsquigarrow (v, \sigma'', \omega'')$

[1] On peut y voir la valeur notée `()`, de type `unit` dans le langage caml, ou le `None` et le `NoneType` de python.

(APPR') si $\rho, \sigma, \omega \vdash_{\text{Expr}} e \rightsquigarrow (inPR(bk, x, (x_1, \ldots, x_n), \rho'), \sigma', \omega')$
si $\rho, \sigma', \omega' \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1)$, ..., si $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$
et si $\rho'[x_1 = v_1, \ldots, x_n = v_n, x = inPR(bk, x, (x_1, \ldots, x_n), \rho')], \sigma_n, \omega_n \vdash_{\text{Block}} bk \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow (v, \sigma'', \omega'')$

Pour mémoire, les règles amendées de *APS2*:

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

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

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

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

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

(PRIM) si $x \in$ oprim, si $\rho, \sigma, \omega \vdash_{\text{Expr}} e_1 \rightsquigarrow (inN(n_1), \sigma_1, \omega_1)$, ..., si $\rho, \sigma_{k-1}, \omega_{k-1} \vdash_{\text{Expr}} e_k \rightsquigarrow (inN(n_k), \sigma_k, \omega_k)$
et si $\pi(x)(n_1, \ldots, n_k) = n$ alors $\rho, \sigma, \omega \vdash_{\text{Expr}}$ `(`$x\ e_1 \ldots e_n$`)` $\rightsquigarrow (inN(n), \sigma_k, \omega_k)$

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

(IF2) si $\rho, \sigma, \omega \vdash_{\text{Expr}} e_1 \rightsquigarrow (inN(0), \sigma', \omega')$ et si $\rho, \sigma', \omega' \vdash_{\text{Expr}} e_3 \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash_{\text{Expr}}$ `(if` $e_1\ e_2\ e_3$`)` $\rightsquigarrow (v, \sigma'', \omega'')$

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

(APP) si $\rho, \sigma, \omega \vdash_{\text{Expr}} e \rightsquigarrow (inF(e', r), \sigma', \omega')$,
si $\rho, \sigma', \omega' \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1)$, ..., si $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$
et si $r(v_1, \ldots, v_n), \sigma_n, \omega_n \vdash_{\text{Expr}} e' \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow (v, \sigma'', \omega'')$

(APPR) si $\rho, \sigma, \omega \vdash_{\text{Expr}} e \rightsquigarrow (inFR(\varphi), \sigma', \omega()$, si $\varphi(inFR(\varphi)) = inF(e', r)$,
si $\rho, \sigma', \omega' \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1)$, ..., si $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$
et si $r(v_1, \ldots, v_n), \sigma_n, \omega_n \vdash_{\text{Expr}} e' \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow (v, \sigma'', \omega'')$

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

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

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

5.3.3 Instructions

Outre leur effet sur la mémoire et le flot de sortie, les instructions peuvent produire une valeur, la *valeur vide* notée ε. Pour *APS3*, le domaine de la relation $\vdash_{\text{Stat}}$ est $E \times S \times 0 \times \text{Stat} \times (V_\varepsilon \times S \times O)$.

On écrit $\rho, \sigma, \omega \vdash_{\text{Stat}} s \rightsquigarrow (v, \sigma', \omega')$. Notez qu'ici, v peut valoir ε.

Les clauses de la définition de $\vdash_{\text{Stat}}$ pour *APS3* suivent celles de *APS2* en ajoutant quel genre de valeur peut être produite. La seule instruction pour laquelle il faut ajouter une règle nouvelle est la boucle. En effet, avec `RETURN`, une boucle peut avoir deux modalité de sortie: la modalité régulière lorsque la condition gourvernant la boucle est fausse et une modalité *exceptionnelle* lorsque l'évaluation du corps de la boucle rencontre un `RETURN`.

Les instructions d'affichage et d'affectation produisent une *valeur vide*

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

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

L'instruction d'alternative prend en compte les éventuels effets de l'évaluation de la condition dans l'évaluation de l'un ou l'autre des blocs alternants:

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

(IF2) si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (inN(0), \sigma', \omega')$ et si $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} bk_2 \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{IF}\ bk_1\ bk_2) \rightsquigarrow (v, \sigma'', \omega'')$

Il y a trois cas à considérer pour la boucle: la sortie régulière de la boucle, la sortie *exceptionnelle* de la boucle et la continuation de la boucle. En cas de sortie règulière, la boucle produit une *valeur vide*. Une boucle n'est itétrée que lorsque l'évaluation de son bloc a produit une *valeur vide*, dans l'autre cas, on a une sortie *exceptionnelle* de la boucle.

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

(LOOP1A) si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (inN(1), \sigma', \omega')$,
si $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} blk \rightsquigarrow (\varepsilon, \sigma'', \omega'')$ et si $\rho, \sigma'', \omega'' \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, \sigma''', \omega''')$
alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, \sigma''', \omega''')$

(LOOP1B) si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (inN(1), \sigma', \omega')$ et si $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} blk \rightsquigarrow (v, \sigma'', \omega'')$, avec $v \neq \varepsilon$
alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, \sigma'', \omega'')$

Les règles d'appel de procédures sont calquées sur les règles d'application de fonction.

(CALL) si $\rho(x) = inP(bk, (x_1, \ldots, x_n), \rho')$,
si $\rho, \sigma, \omega \vdash_{\text{EXPR}} (e_1, \sigma_1) \rightsquigarrow (v_1, \sigma_1, \omega_1)$, ..., si $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$
et si $\rho'[x_1 = v_1, \ldots, x_n = v_n], \sigma_n, \omega_n \vdash_{\text{BLOCK}} bk \rightsquigarrow (v, \sigma', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (v, \sigma', \omega')$

(CALLR) si $\rho(x) = inPR(bk, x, (x_1, \ldots, x_n), \rho')$,
si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1)$, ..., si $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$
et si $\rho'[x_1 = v_1, \ldots, x_n = v_n, x = inPR(bk, x, (x_1, \ldots, x_n), \rho')], \sigma_n, \omega_n \vdash_{\text{BLOCK}} bk \rightsquigarrow (v, \sigma', \omega'))$
alors $\rho, \omega \vdash_{\text{STAT}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (v, \sigma', \omega')$

5.3.4 La commande `RETURN`

On ajoute la relation $\vdash_{\text{RET}}$ de domaine $E \times S \times O \times \text{RET} \times (V \times S \times O)$. On écrit $\rho, \sigma, \omega \vdash_{\text{RET}} r \rightsquigarrow (v, \sigma', \omega')$

La valeur et les effets de la commande `RETURN` sont ceux de l'expression qui lui est associée.

(RET) si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (v, \sigma', \omega')$ alors $\rho, \sigma, \omega \vdash_{\text{RET}} (\texttt{RETURN}\ e) \rightsquigarrow (v, \sigma', \omega')$

5.3.5 Déclaration

La déclaration de constante dont l'évaluation induit celle d'une expression est suceptible d'avoir un effet sur le flot de sortie. Cela va nous amener à modifier la signature de $\vdash_{\text{DEF}}$. Nous ajoutons également les règles concernant les déclarations de fonctions procédurales qui sont nouvelles dans *APS3*.

La signature de $\vdash_{\text{DEF}}$ devient $E \times S \times O \times \text{DEC} \times (E \times S \times O)$ dans *APS3*.
On écrit $\rho, \sigma, \omega \vdash_{\text{DEF}} d \rightsquigarrow (\rho', \sigma', \omega')$

La seule règles de $\vdash_{\text{DEF}}$ qui change «réellement» est donc celle pour les déclarations de constantes.

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

Les règles de déclaration de fonctions procédurales sont définies sur le modèle des règles de déclarations de fonctions et de procédures:

(FUNP) $\rho, \sigma, \omega \vdash_{\text{Def}} (\texttt{FUN}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ bk)$
$\rightsquigarrow (\rho[x = inP(bk, (x_1, \ldots, x_n), \rho)], \sigma, \omega)$

(FUNPR) $\rho, \sigma, \omega \vdash_{\text{Def}} (\texttt{FUN REC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ bk)$
$\rightsquigarrow (\rho[x = inPR(bk, x, (x_1, \ldots, x_n), \rho)], \sigma, \omega)$

Pour les autre règles, on ajoute simplement la mention du flot de sortie qui reste inchangé.

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

(FUN) $\rho, \sigma, \omega \vdash_{\text{Def}} (\texttt{FUN}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ e)$
$\rightsquigarrow (\rho[x = inF(e, (x_1, \ldots, x_n), \rho)], \sigma, \omega)$

(FUNREC) $\rho, \sigma, \omega \vdash_{\text{Def}} (\texttt{FUN REC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ e)$
$\rightsquigarrow (\rho[x = inFR(e, x, (x_1, \ldots, x_n), \rho)], \sigma, \omega)$

(PROC) $\rho, \sigma, \omega \vdash_{\text{Def}} (\texttt{PROC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ bk)$
$\rightsquigarrow (\rho[x = inP(bk, (x_1, \ldots, x_n), \rho)], \sigma, \omega)$

(PROCREC) $\rho, \sigma, \omega \vdash_{\text{Def}} (\texttt{PROC REC}\ x\ t\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ \ldots\ \texttt{,}x_n\texttt{:}t_n\texttt{]}\ bk)$
$\rightsquigarrow (\rho[x = inPR(bk, x, (x_1, \ldots, x_n), bk)], \sigma, \omega)$

5.3.6 Suites de commandes

La sémantique des suites de commandes doit permettre de modéliser la *rupture du flot séquentiel* d'évaluation que provoque la commande `RETURN`. Nous utilisons pour cela la *valeur vide*, comme nous l'avions fait pour détecter la sortie *exceptionnelle* de boucle. Nous aurons donc deux règles pour les suites commençant par une instruction. Nous devons également ajouter la règle concernant en propre la commande `RETURN` qui se trouvera toujours en dernier élément d'une séquence.

Pour tenir compte de la *valeur vide*, la signature de $\vdash_{\text{Cmds}}$ devient $E \times S \times O \times \text{Cmds} \times (V_\varepsilon \times S \times O)$ dans *APS3*.

On écrit $\rho, \sigma, \omega \vdash_{\text{Cmds}} cs \rightsquigarrow (v, \sigma', \omega')$.

Lorsqu'une suite de commandes commence par une instruction qui ne produit pas de valeur, on retrouve la règle usuelle de séquencement. Dans le cas contraire, le reste de la suite est ignoré.

(STAT0) si $\rho, \sigma, \omega \vdash_{\text{Stat}} s \rightsquigarrow (\varepsilon, \sigma', \omega')$ et si $\rho, \sigma', \omega' \vdash_{\text{Cmds}} cs \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash_{\text{Cmds}} (s\texttt{;}cs) \rightsquigarrow (v, \sigma'', \omega'')$

(STAT1) si $\rho, \sigma, \omega \vdash_{\text{Stat}} s \rightsquigarrow (v, \sigma', \omega')$ avec $v \neq \varepsilon$ alors $\rho, \sigma, \omega \vdash_{\text{Cmds}} (s\texttt{;}cs) \rightsquigarrow (v, \sigma', \omega')$

La règle pour les déclarations change peu.

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

Si la suite se termine par un `RETURN` sa valeur et ses effets sont ceux du `RETURN`, sinon, c'est la suite vide qui n'a pas de valeur ni d'effet.

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

(RET) si $\rho, \sigma, \omega \vdash_{\text{Expr}} e \rightsquigarrow (v, \sigma', \omega')$ alors $\rho, \sigma, \omega \vdash_{\text{Cmds}} (r\texttt{;}\varepsilon) \rightsquigarrow (v, \sigma', \omega')$

5.3.7 Blocs

La relation $\vdash_{\text{BLOCK}}$ a pour signature $E \times S \times O \times \text{CMDS} \times (V_\varepsilon \times S \times O)$

On écrit $\rho, \sigma, \omega \vdash_{\text{BLOCK}} bk \rightsquigarrow (v, \sigma', \omega')$

La règle d'évaluation d'un bloc pour *APS3* est celle de *APS2* amendée pour respecter la nouvelle signature de la relation $\vdash_{\text{BLOCK}}$:

(BLOCK) si $\rho, \sigma, \omega \vdash_{\text{CMDS}} cs \rightsquigarrow (v, \sigma', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{BLOCK}} [cs] \rightsquigarrow (v, \sigma', \omega')$

5.3.8 Programme

Un programme n'est pas sensé délivrer une valeur. La signature de $\vdash$ reste donc $\text{PROG} \times (S \times O)$.

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

On ne définit la règle que pour des programmes constitués de suites de commandes produisant la *valeur vide* en partant d'un contexte vide:

(PROG) si $\emptyset, \emptyset, \emptyset \vdash_{\text{BLOCK}} [cs] \rightsquigarrow (\varepsilon, \sigma, \omega)$ alors $\vdash [cs] \rightsquigarrow (\sigma, \omega)$