

SU/FS/master/info/4I503 APS  
Notes de cours

P. MANOURY

Janvier 2019

## Contents

<b>3</b>	<b>APS3: fonctions procédurales</b>	<b>2</b>
3.1	Syntaxe . . . . .	2
3.1.1	Lexique . . . . .	2
3.1.2	Grammaire . . . . .	2
3.2	Typage . . . . .	3
3.2.1	Expressions . . . . .	4
3.2.2	Instructions . . . . .	4
3.2.3	Déclarations . . . . .	4
3.2.4	Suite de commandes, dont RETURN . . . . .	4
3.3	Sémantique . . . . .	5
3.3.1	Domaines sémantiques . . . . .	5
3.3.2	Expressions . . . . .	5
3.3.3	Instructions . . . . .	6
3.3.4	La commande RETURN . . . . .	7
3.3.5	Déclaration . . . . .	7
3.3.6	Suites de commandes . . . . .	8
3.3.7	Blocs . . . . .	9
3.3.8	Programme . . . . .	9

## 3 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 la primitive `alloc` qui est un constructeur 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. Toutefois, cette suite de commandes doit avoir une propriété particulière; elle doit produire une valeur. Pour ce, nous introduisons la commande *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 susceptibles de produire une valeur.

### 3.1 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.

#### 3.1.1 Lexique

Mot clef `RETURN`

#### 3.1.2 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*.

Notez encore que cette tentative de contrôle syntaxique du code mort n'est toutefois 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.

## 3.2 Typage

**Typage et analyse de flot** Puisqu'une suite de commandes peut produire une valeur, elle n'est plus nécessairement de type void. Ceci implique également qu'une alternative ou une boucle n'a plus nécessairement le type void. La situation est encore plus délicate avec l'alternative qui contient deux blocs distincts.

En effet, considérons le fragment de code suivant, qui donnera true ou false selon qu'une fonction s'annule ou non sur un intervalle délimité par les valeurs de a et b:

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

Ce code est syntaxiquement correct et nous voulons pouvoir l'évaluer. *A priori*, ce fragment de code doit pouvoir être typé de type bool. Mais cela soulève la difficulté d'attribuer un type à l'alternative

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

dont les deux branches n'ont pas un type uniforme.

On a donc un code syntaxiquement et opérationnellement acceptable mais dont l'un des composant (l'alternative) n'obéit à la discipline de type naturelle où les deux branches d'une alternative ont le même type. Faut-il ou non accepter de telles constructions ? Il y a là un point de *conception de langage*.

On peut décider pas la négative en interdisant ce style de programmation. Mais cela nous priverait d'un idiome de programmation assez commun<sup>1</sup>. Nous décidons donc d'accepter cette possibilité et nous devons définir la discipline de type qui l'acceptera.

Si l'on y regarde d'un peu plus près, le dilemme sur l'alternative n'est pas général. En effet, si l'on veut attacher une discipline de type à un langage, il ne serait pas raisonnable d'accepter une construction comme celle-ci:

```
IF (lt x 42) [ RETURN x ] [ RETURN false ]
```

Le problème se pose plutôt lorsque l'une des branches d'une alternative est de type void et l'autre ne l'est pas car on peut accepter qu'une branche ne produise rien si la suite du programme vient combler ce *vide* en produisant une valeur, comme dans notre exemple précédent. Il va alors de soi que le type de la valeur produite par la suite du programme doit être cohérente avec celle éventuellement produite par l'alternative. De cette analyse, on conclut

1. on peut accepter des constructions alternatives dont l'une des branches est de type void et l'autre non. On note ce cas figure en attribuant à l'alternative un type *mixte* ou *type union* que l'on notera  $t + \text{void}$  (avec  $t \in \{\text{bool}, \text{int}\}$ ).
2. si, dans une suite de commandes, l'une d'elle est de type  $t + \text{void}$  alors la (sous)suite de commandes qui vient après elle doit être de type  $t$ .

L'ensemble des types que l'on peut obtenir est donc:  $\{\text{void}, \text{bool}, \text{int}, \text{bool} + \text{void}, \text{int} + \text{void}\}$ . On pose que  $t + \text{void} + \text{void} = t + \text{void}$ .

---

<sup>1</sup>Dans la *vraie vie* on rencontre souvent ce style de programmation avec l'utilisation d'un *if* unilatère

**Signature des relations de typage** Les suites de commandes pouvant produire (ou non) une valeur, la relation de typage pour les suites de commandes, et, donc, des instructions, changent de signature.

Posons  $\text{TYPE}_{\text{void}} = \text{TYPE} \cup \{t + \text{void} \mid t \in \text{TYPE}\} \cup \{\text{void}\}$

$\vdash_{\text{CMDS}}$  est une relation dans  $G \times \text{CMDS} \times \text{TYPE}_{\text{void}}$

### 3.2.1 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.

### 3.2.2 Instructions

Pour l'alternative, nous devons réaliser la première conclusion de notre analyse qui ouvre la possibilité de lui attribuer une *type mixte*.

(IF0) pour tout type  $t$ , si  $G \vdash_{\text{EXPR}} e : \text{bool}$  et  $G \vdash_{\text{BLOCK}} \text{blk}_1 : t$  et  $G \vdash_{\text{BLOCK}} \text{blk}_2 : t$  alors  $G \vdash_{\text{STAT}} (\text{IF } e \text{ blk}_1 \text{ blk}_2) : t$

(IF1) pour tout  $t \neq \text{void}$ , si  $G \vdash_{\text{EXPR}} e : \text{bool}$  et  $G \vdash_{\text{BLOCK}} \text{blk}_1 : \text{void}$  et  $G \vdash_{\text{BLOCK}} \text{blk}_2 : t$  alors  $G \vdash_{\text{STAT}} (\text{IF } e \text{ blk}_1 \text{ blk}_2) : t + \text{void}$

(IF2) pour tout  $t \neq \text{void}$ , si  $G \vdash_{\text{EXPR}} e : \text{bool}$  et  $G \vdash_{\text{BLOCK}} \text{blk}_1 : t$  et  $G \vdash_{\text{BLOCK}} \text{blk}_2 : \text{void}$  alors  $G \vdash_{\text{STAT}} (\text{IF } e \text{ blk}_1 \text{ blk}_2) : t + \text{void}$

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

(WHILE) pour tout type  $t$ , si  $G \vdash_{\text{EXPR}} e : \text{bool}$  et  $G \vdash_{\text{BLOCK}} \text{blk} : t$  alors  $G \vdash_{\text{STAT}} (\text{WHILE } e \text{ blk}) : t + \text{void}$

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

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

### 3.2.3 Déclarations

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

(FUNP) si  $\Gamma[x_1 : t_1; \dots; x_n : t_n] \vdash_{\text{BLOCK}} \text{bk} : t$   
alors  $\Gamma \vdash_{\text{DEC}} (\text{FUN } x \text{ t } [x_1 : t_1, \dots, x_n : t_n] \text{ bk}) : \Gamma[x : t_1 * \dots * t_n \rightarrow t]$

(FUNRECP) si  $\Gamma[x_1 : t_1; \dots; x_n : t_n; x : t_1 * \dots * t_n \rightarrow t] \vdash_{\text{BLOCK}} \text{bk} : t$   
alors  $\Gamma \vdash_{\text{DEC}} (\text{FUN REC } x \text{ t } [x_1 : t_1, \dots, x_n : t_n] \text{ bk}) : \Gamma[x : t_1 * \dots * t_n \rightarrow t]$

### 3.2.4 Suite de commandes, dont RETURN

Il faut réaliser ici notre seconde conclusion. Celle-ci ne concerne que l'occurrence des instructions. La règle des suites commençant par une déclaration ne change pas par rapport à *APS2*.

Nous considérons trois cas dans l'analyse de type d'une suite de commandes commençant par une instruction:

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

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

Notez que faisant cela, nous éliminons par analyse de type les suites de longueur strictement supérieure à 1 commençant par une instruction du genre `IF e [ RETURN e1 ] [ RETURN e2 ]`. Nous n'interdisons cependant pas qu'une suite se termine par une telle alternative.

Nous avons pu fixer syntaxiquement que la commande `RETURN` ne peut apparaître qu'en fin de suite. Il est donc suffisant de définir une règle pour typer la suite qui contient uniquement `RETURN`:

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

L'analyse de type des suites qui ne contiennent pas de `RETURN` se terminent par l'application de

(END)  $\Gamma \vdash_{\text{CMDS}} \varepsilon : \text{void}$

### 3.3 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. À la manière dont nous avons complété l'ensemble des commandes avec la *commande vide* (notée  $\varepsilon$ ), nous complétons l'ensemble des valeurs avec une *valeur vide* (que nous noterons également  $\varepsilon$ ). Cette pseudo valeur nous servira à détecter l'occurrence 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*.

#### 3.3.1 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 avons déjà dans *APS1*.

#### 3.3.2 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  $\rho, \sigma, \omega \vdash_{\text{EXPR}} rv \rightsquigarrow (v, \sigma', \omega')$

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

(APPR') si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{inPR}(\varphi), \sigma', \omega')$ , si  $\varphi(\text{inPR}(\varphi)) = \text{inP}(bk, r)$ ,  
si  $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1), \dots$ , si  $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$   
et si  $r(v_1, \dots, v_n), \sigma_n, \omega_n \vdash_{\text{BLOCK}} bk \rightsquigarrow (v, \sigma'', \omega'')$   
alors  $\rho, \sigma, \omega \vdash (e e_1 \dots e_n) \rightsquigarrow (v, \sigma'', \omega'')$

Pour mémoire, les règles amendées de *APS2*:

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

- (FALSE)  $\rho, \sigma, \omega \vdash_{\text{EXPR}} \mathbf{false} \rightsquigarrow (\text{in}N(0), \sigma, \omega)$
- (NUM) si  $n \in \text{num}$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} n \rightsquigarrow (\text{in}N(\nu(n)), \sigma, \omega)$
- (ID1) si  $x \in \text{ident}$  et  $\rho(x) = \text{in}A(a)$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} x \rightsquigarrow (\text{in}N(\sigma(a)), \sigma, \omega)$
- (ID2) si  $x \in \text{ident}$  et  $\rho(x) = v$ , avec  $v \neq \text{in}A(a)$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} x \rightsquigarrow (v, \sigma, \omega)$
- (PRIM) si  $x \in \text{oprim}$ , si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (\text{in}N(n_1), \sigma_1, \omega_1), \dots$ , si  $\rho, \sigma_{k-1}, \omega_{k-1} \vdash_{\text{EXPR}} e_k \rightsquigarrow (\text{in}N(n_k), \sigma_k, \omega_k)$  et si  $\pi(x)(n_1, \dots, n_k) = n$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} (x e_1 \dots e_n) \rightsquigarrow (\text{in}N(n), \sigma_k, \omega_k)$
- (IF1) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (\text{in}N(1), \sigma', \omega')$  et si  $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_2 \rightsquigarrow (v, \sigma'', \omega'')$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} (\mathbf{if} e_1 e_2 e_3) \rightsquigarrow (v, \sigma'', \omega'')$
- (IF2) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (\text{in}N(0), \sigma', \omega')$  et si  $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_3 \rightsquigarrow (v, \sigma'', \omega'')$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} (\mathbf{if} e_1 e_2 e_3) \rightsquigarrow (v, \sigma'', \omega'')$
- (ABS)  $\rho, \sigma, \omega \vdash_{\text{EXPR}} [x_1:t_1, \dots, x_n:t_n]e \rightsquigarrow (\text{in}F(e, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n]), \sigma, \omega)$
- (APP) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}F(e', r), \sigma', \omega')$ ,  
si  $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1), \dots$ , si  $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$   
et si  $r(v_1, \dots, v_n), \sigma_n, \omega_n \vdash_{\text{EXPR}} e' \rightsquigarrow (v, \sigma'', \omega'')$   
alors  $\rho, \sigma, \omega \vdash (e e_1 \dots e_n) \rightsquigarrow (v, \sigma'', \omega'')$
- (APPR) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}FR(\varphi), \sigma', \omega')$ , si  $\varphi(\text{in}FR(\varphi)) = \text{in}F(e', r)$ ,  
si  $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1), \dots$ , si  $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$   
et si  $r(v_1, \dots, v_n), \sigma_n, \omega_n \vdash_{\text{EXPR}} e' \rightsquigarrow (v, \sigma'', \omega'')$   
alors  $\rho, \sigma, \omega \vdash (e e_1 \dots e_n) \rightsquigarrow (v, \sigma'', \omega'')$
- (ALLOC) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(n), \sigma', \omega')$  et si  $\text{allocn}(\sigma', n) = (a, \sigma'')$   
alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} (\mathbf{alloc} e) \rightsquigarrow (\text{in}B(a, n), \sigma'', \omega')$
- (NTH) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (\text{in}B(a, n), \sigma', \omega')$  et si  $\rho, \sigma', \omega' \vdash_{\text{EXPR}} (\text{in}N(i), \sigma'', \omega'')$   
alors  $\rho, \sigma \vdash_{\text{EXPR}} (\mathbf{nth} e_1 e_2) \rightsquigarrow (\sigma''(a + i), \sigma'', \omega'')$
- (LEN) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}B(a, n), \sigma', \omega')$  alors  $\rho, \sigma, \omega \vdash_{\text{EXPR}} (\mathbf{len} e) \rightsquigarrow (\text{in}N(n), \sigma', \omega')$

### 3.3.3 Instructions

Outre leur effet sur la mémoire et le flot de sortie, les instructions peuvent produire une valeur. Et, excusez nous du paradoxe, parmi ces valeurs peut figurer la *valeur vide* notée  $\varepsilon$ . 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  $\varepsilon$ .

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 modalités de sortie: la modalité régulière lorsque la condition gouvernant 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  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(n), \sigma', \omega')$  alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\mathbf{ECHO} e) \rightsquigarrow (\varepsilon, \sigma', (n \cdot \omega))$
- (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  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\mathbf{SET} lv e) \rightsquigarrow (\varepsilon, \sigma'[x = v], \omega')$

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  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(1), \sigma', \omega')$  et si  $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} \text{bk}_1 \rightsquigarrow (v, \sigma'', \omega'')$   
alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\text{IF } \text{bk}_1 \text{ bk}_2) \rightsquigarrow (v, (\sigma''/\rho), \omega'')$
- (IF2) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(0), \sigma', \omega')$  et si  $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} \text{bk}_2 \rightsquigarrow (v, \sigma'', \omega'')$   
alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\text{IF } \text{bk}_1 \text{ bk}_2) \rightsquigarrow (v, (\sigma''/\rho), \omega'')$

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 régulière, la boucle produit une *valeur vide*. Une boucle n'est itérée 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  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(0), \sigma', \omega')$  alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\text{WHILE } e \text{ blk}) \rightsquigarrow (\varepsilon, \sigma', \omega')$
- (LOOP1A) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(1), \sigma', \omega')$ ,  
si  $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} \text{blk} \rightsquigarrow (\varepsilon, \sigma'', \omega'')$  et si  $\rho, (\sigma''/\rho), \omega'' \vdash_{\text{STAT}} (\text{WHILE } e \text{ blk}) \rightsquigarrow (v, \sigma''', \omega''')$   
alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\text{WHILE } e \text{ blk}) \rightsquigarrow (v, \sigma''', \omega''')$
- (LOOP1B) si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (\text{in}N(1), \sigma', \omega')$  et si  $\rho, \sigma', \omega' \vdash_{\text{BLOCK}} \text{blk} \rightsquigarrow (v, \sigma'', \omega'')$ , avec  $v \neq \varepsilon$   
alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\text{WHILE } e \text{ blk}) \rightsquigarrow (v, (\sigma''/\rho), \omega'')$

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

- (CALL) si  $\rho(x) = \text{in}P(\text{bk}, r)$ ,  
si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} (e_1, \sigma_1) \rightsquigarrow (v_1, \sigma_1, \omega_1), \dots$ , si  $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$   
et si  $r(v_1, \dots, v_n), \sigma_n, \omega_n \vdash_{\text{BLOCK}} \text{bk} \rightsquigarrow (v, \sigma', \omega')$   
alors  $\rho, \sigma, \omega \vdash_{\text{STAT}} (\text{CALL } x \ e_1 \dots e_n) \rightsquigarrow (v, (\sigma'/\rho), \omega')$
- (CALLR) si  $\rho(x) = \text{in}PR(\varphi)$ , si  $\varphi(\text{in}PR(\varphi)) = \text{in}P(\text{bk}, r)$ ,  
si  $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1), \dots$ , si  $\rho, \sigma_{n-1}, \omega_{n-1} \vdash_{\text{EXPR}} e_n \rightsquigarrow (v_n, \sigma_n, \omega_n)$   
et si  $r(v_1, \dots, v_n), \sigma_n, \omega_n \vdash_{\text{BLOCK}} \text{bk} \rightsquigarrow (v, \sigma', \omega')$   
alors  $\rho, \omega \vdash_{\text{STAT}} (\text{CALL } x \ e_1 \dots e_n) \rightsquigarrow (v, (\sigma'/\rho), \omega')$

### 3.3.4 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  $\rho, \sigma, \omega \vdash_{\text{RET}} (\text{RETURN } e) \rightsquigarrow (v, \sigma', \omega')$

### 3.3.5 Déclaration

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

La signature de  $\vdash_{\text{DEC}}$  devient  $E \times S \times O \times \text{DEC} \times E \times S \times O$  dans APS3.

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

La seule règles de  $\vdash_{\text{DEC}}$  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  $\rho, \sigma, \omega \vdash_{\text{DEC}} (\text{CONST } x \ t \ e) \rightsquigarrow (\rho[x = v], \sigma', \omega')$

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

$$\begin{aligned}
(\text{FUNP}) \quad & \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{FUN } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ bk) \\
& \rightsquigarrow (\rho[x = \text{inP}(bk, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n])], \sigma, \omega) \\
(\text{FUNPR}) \quad & \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{FUN REC } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ bk) \\
& \rightsquigarrow (\rho[x = \text{inPR}(\lambda f. \text{inP}(bk, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n][x = f])], \sigma, \omega)
\end{aligned}$$

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

$$\begin{aligned}
(\text{VAR}) \quad & \text{si } \text{alloc}(\sigma) = (a, \sigma') \text{ alors } \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{VAR } x \ t) \rightsquigarrow (\rho[x = \text{inA}(a)], \sigma', \omega) \\
(\text{FUN}) \quad & \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{FUN } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ e) \\
& \rightsquigarrow (\rho[x = \text{inF}(e, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n])], \sigma, \omega) \\
(\text{FUNREC}) \quad & \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{FUN REC } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ e) \\
& \rightsquigarrow (\rho[x = \text{inFR}(\lambda f. \text{inF}(e, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n][x = f])], \sigma, \omega) \\
(\text{PROC}) \quad & \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{PROC } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ bk) \\
& \rightsquigarrow (\rho[x = \text{inP}(bk, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n])], \sigma, \omega) \\
(\text{PROCREC}) \quad & \rho, \sigma, \omega \vdash_{\text{DEC}} (\text{PROC REC } x \ t \ [x_1:t_1, \dots, x_n:t_n] \ bk) \\
& \rightsquigarrow (\rho[x = \text{inPR}(\lambda f. \text{inP}(bk, \lambda v_1 \dots v_n. \rho[x_1 = v_1; \dots; x_n = v_n][x = f])], \sigma, \omega)
\end{aligned}$$

### 3.3.6 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'avons 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  $E \times S \times O \times \text{CMDS}_\varepsilon \times V_\varepsilon \times S \times O$  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 séquençement. Dans le cas contraire, le reste de la suite est ignoré.

$$\begin{aligned}
(\text{STAT0}) \quad & \text{si } \rho, \sigma, \omega \vdash_{\text{STAT}} s \rightsquigarrow (\varepsilon, \sigma', \omega') \text{ et si } \rho, \sigma', \omega' \vdash_{\text{CMDS}} cs \rightsquigarrow (v, \sigma'', \omega'') \text{ alors } \rho, \sigma, \omega \vdash_{\text{CMDS}} (s; cs) \rightsquigarrow \\
& (v, \sigma'', \omega'') \\
(\text{STAT1}) \quad & \text{si } \rho, \sigma, \omega \vdash_{\text{STAT}} s \rightsquigarrow (v, \sigma', \omega') \text{ avec } v \neq \varepsilon \text{ alors } \rho, \sigma, \omega \vdash_{\text{CMDS}} (s; cs) \rightsquigarrow (v, \sigma', \omega')
\end{aligned}$$

La règle pour les déclarations change peu.

$$\begin{aligned}
(\text{DEC}) \quad & \text{si } \rho, \sigma, \omega \vdash_{\text{DEC}} d \rightsquigarrow (\rho', \sigma', \omega') \text{ et si } \rho', \sigma', \omega' \vdash_{\text{CMDS}} cs \rightsquigarrow (v, \sigma'', \omega'') \\
& \text{alors } \rho, \sigma, \omega \vdash_{\text{CMDS}} (d; cs) \rightsquigarrow (v, \sigma'', \omega'')
\end{aligned}$$

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.

$$\begin{aligned}
(\text{END0}) \quad & \rho, \sigma, \omega \vdash_{\text{CMDS}} \varepsilon \rightsquigarrow (\varepsilon, \sigma, \omega) \\
(\text{END1}) \quad & \text{si } \rho, \sigma, \omega \vdash_{\text{RET}} r \rightsquigarrow (v, \sigma', \omega') \text{ alors } \rho, \sigma, \omega \vdash_{\text{CMDS}} (r; \varepsilon) \rightsquigarrow (v, \sigma', \omega')
\end{aligned}$$

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

### 3.3.8 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  $\vdash [cs] \rightsquigarrow (\sigma, \omega)$