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

P. Manoury

Janvier 2018

Contents

1 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 suceptibles de produire une valeur.

1.1 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.

1.1.1 Lexique

Mot clef `RETURN`

1.1.2 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 dernière 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.

1.2 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:

```
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 altenative 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[1]. 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 dilemne 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 construition 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 $t + \texttt{void}$ (avec $t \in \{\texttt{bool}, \texttt{int}\}$).

2. si, dans une suite de commandes, l'une d'elle est de type $t + \texttt{void}$ 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: $\{\texttt{void}, \texttt{bool}, \texttt{int}, \texttt{bool} + \texttt{void}, \texttt{int} + \texttt{void}\}$. On pose que $t + \texttt{void} + \texttt{void} = t + \texttt{void}$.

[1] Dans la *vraie vie* on rencontre souvent ce style de programmation avec l'utilisation d'un *if* unilatère

1.2.1 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.

1.2.2 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 $G \vdash_{\text{Expr}} e : \texttt{bool}$ et $G \vdash_{\text{Block}} blk_1 : t$ et $G \vdash_{\text{Block}} blk_2 : t$ alors $G \vdash_{\text{Stat}} (\texttt{IF}\ e\ blk_1\ blk_2) : t$

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

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

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

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

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

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

1.2.3 Déclarations

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

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

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

1.2.4 Suite de commandes, dont `RETURN`

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

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

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

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

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` e_1 `] [ RETURN` e_2 `]`. 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 $\Gamma \vdash_{\text{Cmds}} (\texttt{RETURN}\ e; \varepsilon) : t$

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

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

1.3 Sémantique

D'avoir ajouté que le corps d'une fonction puisse être une suite de commandes implique que l'application de telles fonctions est suceptible 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 rejoingnent: 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 ε), nous complétons l'ensemble des valeurs avec une *valeur vide* (que nous noterons également ε). 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*.

1.3.1 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*.

1.3.2 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.

Comme pour l'appel d'une procédure de *APS1*, l'évaluation de l'application d'une fonction procédurale implique l'évaluation d'un bloc. Nous utiliserons donc l'opération de restriction mémoire pour libérer les adresses localement attribuées. Toutefois, avec *APS3*, il faut prendre garde que l'évaluation d'un bloc peut donner une valeur et que cette valeur peut faire référence à une adresse allouée lors de l'évaluation du bloc. C'est le cas d'une fonction procédurale dont le résultat est un nouveau tableau. Il ne faut pas, dans ce cas, désallouer la mémoire associée à ce nouveau tableau. Pour conserver l'éventuel espace mémoire qui peut être le résultat de l'exécution d'un bloc, on l'introduit artificiellement dans l'environnement servant à calculer l'ensemble des adresses accessibles en l'associant à un identificateur quelconque que nous notons δ:

(APP') si $\rho, \sigma, \omega \vdash_{\text{Expr}} e \rightsquigarrow (inP(bk, r), \sigma', \omega')$,
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 $r(v_1, \ldots, v_n), \sigma_n, \omega_n \vdash_{\text{Block}} bk \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow (v, (\sigma''/\rho[\delta = v]), \omega'')$

(APPR') si $\rho, \sigma, \omega \vdash_{\text{Expr}} e \rightsquigarrow (inPR(\varphi), \sigma', \omega')$, si $\varphi(inPR(\varphi)) = inP(bk, r)$,
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 $r(v_1, \ldots, v_n), \sigma_n, \omega_n \vdash_{\text{Block}} bk \rightsquigarrow (v, \sigma'', \omega'')$
alors $\rho, \sigma, \omega \vdash$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow (v, (\sigma''/\rho[\delta = v]), \omega'')$

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

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

(PRIM) si $x \in \mathsf{oprim}$, si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e_1 \rightsquigarrow (inN(n_1), \sigma_1, \omega_1), \ldots,$ 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 $\rho, \sigma, \omega \vdash_{\text{EXPR}} \texttt{(}x\ e_1 \ldots e_n\texttt{)} \rightsquigarrow (inN(n), \sigma_k, \omega_k)$

(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 $\rho, \sigma, \omega \vdash_{\text{EXPR}} \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \rightsquigarrow (v, \sigma'', \omega'')$

(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 $\rho, \sigma, \omega \vdash_{\text{EXPR}} \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \rightsquigarrow (v, \sigma'', \omega'')$

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

(APP) si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (inF(e', r), \sigma', \omega')$,
si $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1), \ldots,$ 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 $\rho, \sigma, \omega \vdash \texttt{(}e\ e_1 \ldots e_n\texttt{)} \rightsquigarrow (v, \sigma'', \omega'')$

(APPR) si $\rho, \sigma, \omega \vdash_{\text{EXPR}} e \rightsquigarrow (inFR(\varphi), \sigma', \omega()$, si $\varphi(inFR(\varphi)) = inF(e', r)$,
si $\rho, \sigma', \omega' \vdash_{\text{EXPR}} e_1 \rightsquigarrow (v_1, \sigma_1, \omega_1), \ldots,$ 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 $\rho, \sigma, \omega \vdash \texttt{(}e\ e_1 \ldots e_n\texttt{)} \rightsquigarrow (v, \sigma'', \omega'')$

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

(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 $\rho, \sigma \vdash_{\text{EXPR}} \texttt{(nth}\ e_1\ e_2\texttt{)} \rightsquigarrow (\sigma''(a + i), \sigma'', \omega'')$

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

1.3.3 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 le *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'évalaution 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 $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{IF}\ bk_1\ bk_2) \rightsquigarrow (v, (\sigma''/\rho), \omega'')$

(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 $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{IF}\ bk_1\ bk_2) \rightsquigarrow (v, (\sigma''/\rho), \omega'')$

Il y a trois cas à considérer pour la boucle: la sortie régilière 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 $\rho, (\sigma''/\rho), \omega'' \vdash_{\text{Stat}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, \sigma''', \omega''')$
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 $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, (\sigma''/\rho), \omega'')$

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

(CALL) si $\rho(x) = inP(bk, r)$,
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 $r(v_1, \ldots, v_n), \sigma_n, \omega_n \vdash_{\text{Block}} bk \rightsquigarrow (v, \sigma', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{Stat}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (v, (\sigma'/\rho), \omega')$

(CALLR) si $\rho(x) = inPR(\varphi)$, si $\varphi(inPR(\varphi)) = inP(bk, r)$,
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 $r(v_1, \ldots, v_n), \sigma_n, \omega_n \vdash_{\text{Block}} bk \rightsquigarrow (v, \sigma', \omega'))$
alors $\rho, \omega \vdash_{\text{Stat}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (v, (\sigma'/\rho), \omega')$

1.3.4 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 $\rho, \sigma, \omega \vdash_{\text{Ret}} (\texttt{RETURN}\ e) \rightsquigarrow (v, \sigma', \omega')$

1.3.5 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 $\vdash_{\text{Dec}}$. 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 $\rho, \sigma, \omega \vdash_{\text{Dec}} (\texttt{CONST}\ x\ t\ e) \rightsquigarrow (\rho[x = v], \sigma', \omega')$

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

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

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

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

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

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

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

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

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

1.3.6 Suites de commandes

La sémantique des suites de commandes doit permettre de modéliser la *rupture du flot séquentiel* d'évaluation que provoque le 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 $E \times S \times O \times \text{CMDS}_\varepsilon \times V_\varepsilon \times S \times O$ 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 séquencement. Dans le cas contraire, les 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 $\rho, \sigma, \omega \vdash_{\text{DEC}} d \rightsquigarrow (\rho', \sigma', \omega')$ 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.

(END0) $\rho, \sigma, \omega \vdash_{\text{CMDS}} \varepsilon \rightsquigarrow (\varepsilon, \sigma, \omega)$

(END1) si $\rho, \sigma, \omega \vdash_{\text{RET}} r \rightsquigarrow (v, \sigma', \omega')$ alors $\rho, \sigma, \omega \vdash_{\text{CMDS}} (r\texttt{;}\varepsilon) \rightsquigarrow (v, \sigma', \omega')$

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

1.3.8 Programme

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

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

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 $\emptyset, \emptyset, \emptyset \vdash_{\textsc{Block}} \texttt{[}cs\texttt{]} \rightsquigarrow (\varepsilon, \sigma, \omega)$ alors $\vdash \texttt{[}cs\texttt{]} \rightsquigarrow (\sigma, \omega)$