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

P. Manoury

Janvier 2017

3 APS2: fonctions et procédures

Le langage *APS1* est l'extension de *APS0* avec la possibilité d'exprimer et de définir de nouvelles fonctions. Pour ce, le langage des expressions a été étendu avec les *expressions fonctionnelles*. Sur ce modèle, nous avons également ajouté des *expressions procédurales* que nous avons utilisées pour *définir*, pour un programme, des nouvelles *instructions* atomiques.

Pour construire *APS2*, nous allons fusionner les deux nouveaux aspects introduits dans *APS1* 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 particilère; elle doit produire une valeur. Pour ce, nous introduisons la commande *return*.

3.1 La commande *return*

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.

3.1.1 Syntaxe

Lexique on ajoute le mot clef `RETURN`

Grammaire on ajoute un non-terminal spécifique au `RETURN`.
Ret ::= `RETURN` Expr

3.1.2 Typage

On donne à l'instruction *return* le type de l'expression dont elle transmet la valeur.

(Ret) Si $G \vdash_{\text{Cmds}} e : t$ alors $G \vdash (\texttt{RETURN}\ e) : t$

Nota: la commande `Return` est toujours la dernière commandes d'une suite de commandes. On intègre sa relation de tyoage à celle des suites de commandes.

3.1.3 Sémantique opérationnelle

La commande `RETURN` donne une valeur qui est celle de l'expression qui lui est attachée

(ret) si $r, m \vdash_{\text{Cmds}} e \rightsquigarrow v$ alors $r, m \vdash \texttt{RETURN}\ e \rightsquigarrow v$

3.1.4 Sémantique dénotationnelle

On ajoute aux fonctions sémantiques la fonction
$\mathbf{R} : \textsc{Ret} \to E \to S \to V$
Et on pose simplement:
$\mathbf{R}[[\texttt{RETURN}\ e]]\rho\sigma \quad = \quad \mathbf{E}[[e]]\rho\sigma$

3.2 Suites de commandes et instructions

Il nous faut maintenant inscrire la place de la commande `RETURN` dans les suites de commandes. Nous fixons que la commande `RETURN` ne pourra apparaître qu'en fin d'une suite de commandes. Ce que l'on peut contraindre syntaxiquement.

3.2.1 Syntaxe

$\textsc{Cmds} \quad ::= \quad \textsc{Ret} \mid \textsc{Stat} \mid \textsc{Dec}\ \texttt{;}\ \textsc{Cmds} \mid \textsc{Stat}\ \texttt{;}\ \textsc{Cmds}$
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*.

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. Mais, auparavant, afin de voir *l'effet de rupture* de la commande `RETURN`, nous allons définir la sémantique (opérationnelle) de *APS2*.

3.2.2 Sémantique opérationnelle

Dans *APS1* une suite de commandes produit un nouvel état mémoire, mais dans *APS2* elle peut également produire une valeur. La relation sémantique des suites de commandes donc être modifiée pour *APS2*. Elle y devient une relation entre un environnement, une mémoire, une suite de commandes et soit simplement une mémoire, soit un couple formé d'une valeur et d'un état mémoire.

En reprenant les domaines de la sémantique dénotationnelle, on pourrait poser que le domaine de la relation sémantique des suites de commandes dans *APS2* est:

$$E \times S \times \textsc{Cmds} \times (M \oplus (V \times M))$$

Nous utilisons la somme disjointe car il nous faut pouvoir récupérer, le cas échéant, la valeur produite (une simple union ne nous l'aurait pas permis).

On peut toutefois adopter une autre vision qui est que toutes les suites de commandes produisent une valeur en assignant à l'instruction atomique d'affectation une *valeur vide* notée ε[1].

On modifie donc le domaine des valeurs : $U = \{\varepsilon\}$ et $V = N \cup B \cup U$

Le domaine de la relation sémantique pour les suites de commandes devient $E \times S \times \textsc{Cmds} \times (V \times M)$

Notez que ce changement affecte également le domaine de la relation sémantique des instructions puisque les instructions `IF` et `WHILE` contiennent des suites de commandes. Le domaine de la relation sémantique pour les instructions sera donc $E \times S \times \textsc{Stat} \times (V \times M)$

[1] C'est la solution envisagée par le langage Ocaml avec la valeur notée `()`, unique valeur du type `unit`

Les instructions d'affectation et d'alternative subissent un changement minime:

(SET) si $r(x) = @a$ et si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow v$ alors $r, m \vdash_{\text{STAT}} (\texttt{SET}\ x\ e) \rightsquigarrow (\varepsilon, m[a := v])$

(ALT1) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \#t$ et $r, m \vdash_{\text{BLOCK}} blk_1 \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{STAT}} (\texttt{IF}\ e\ blk_1\ blk_2) \rightsquigarrow (v, m')$

(ALT0) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \#f$ et $r, m \vdash_{\text{BLOCK}} blk_2 \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{STAT}} (\texttt{IF}\ e\ blk_1\ blk_2) \rightsquigarrow (v, m')$

Toutefois ce changement n'est minime qu'en apparence: la valeur v produite par l'alternative peut-être soit la «non valeur» ε, soit une «vraie valeur» (numérique ou booléenne). Avec la boucle, le changement devint très visible:

(LOOP0) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \#f$ alors $r, m \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (\varepsilon, m)$

(LOOP1) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \#t$ et $r, m \vdash_{\text{BLOCK}} blk \rightsquigarrow (\varepsilon, m')$ et $r, m' \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, m'')$ alors $r, m \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, m'')$

(LOOP2) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \#t$ et $r, m \vdash_{\text{BLOCK}} blk \rightsquigarrow (v, m')$, avec $v \neq \varepsilon$ alors $r, m \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, m')$

Dans la règle (LOOP2), l'obtention de la valeur $v \neq \varepsilon$ est l'indication que l'évaluation de blk a rencontré un `RETURN`. Dans ce cas, le contrôle ordinnaire de la boucle est rompu. Avec la commande `RETURN`, il y a désormais deux manières de sortir d'une boucle: soit de manière régulière selon la valeur de la condition; soit de manière plus radicale à l'occurrence d'un `RETURN`.

Suites de commandes Nous allons retrouver ce phénomène de rupture dans les suites de commandes. Désormais l'évaluation d'une suite de commandes pourra être interrompue par l'occurence d'un `RETURN`, sachant que celle-ci peut se situer dans le bloc d'une alternative ou d'une boucle.

(DEC) pour toute déclaration d, si $r, m \vdash_{\text{DEC}} d \rightsquigarrow (r', m')$ et si $r', m' \vdash_{\text{CMDS}} cs \rightsquigarrow (v, m'')$ alors $r, m \vdash (d\texttt{;}cs) \rightsquigarrow (v, m'')$

(STAT0) pour tout instruction s, si $r, m \vdash_{\text{STAT}} s \rightsquigarrow (\varepsilon, m')$ et si $r, m' \vdash_{\text{CMDS}} cs \rightsquigarrow (v, m'')$ alors $r, m \vdash_{\text{CMDS}} (s\texttt{;}cs) \rightsquigarrow (v, m'')$

(STAT1) pour tout instruction s, si $r, m \vdash_{\text{STAT}} s \rightsquigarrow (v, m')$, avec $v \neq \varepsilon$, alors $r, m \vdash_{\text{CMDS}} (s\texttt{;}cs) \rightsquigarrow (v, m')$

Le cas de base de l'évaluation d'une suite de commandes est le cas d'une suite à une seule commande. Par définition (voir syntaxe), celle-ci ne peut-être qu'une instruction ou la commande `RETURN`. Les règles SET, ALT*, LOOP* et RET sont donc également considéerées comme règles d'évaluation des suites de commandes.

Blocs ou programmes Le changement concerne également le domaine de la relation sémantique des blocs puisque l'on peut obtenir une valeur en retour d'un bloc.

(BLOC) si $r, m \vdash_{\text{CMDS}} cs \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{BLOCK}} \texttt{[}cs\texttt{]} \rightsquigarrow (v, (m'/r))$

3.2.3 Typage et analyse de flot

Dans *APS1* la frontière est stricte entre les construction syntaxiques qui produisent une valeur et celles qui affectent la mémoire. Avec l'introduction du `RETURN` dans *APS2*, les choses se compliquent puisqu'un bloc peut produire une valeur. Un bloc n'a donc plus nécessairement le 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 cherche le 0 d'une fonction `f` entre les valeurs `a` et `b`

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

Ce code est syntaxiquement correct. Il sera également correctement évalué, donnant soit une valeur de `x` qui annule la fonction, soit la valeur 0 s'il n'y a pas de `x` entre `a` et `b` qui annule `f`. *A priori*, ce fragment de code doit pouvoir être typé de type int. Mais cela soulève la difficulté d'attribuer un type à l'alternative

```
IF (eq (f x) 0) [ RETURN x ] [ 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 une 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[2]. 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. Il va alors de soit 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 et 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 $t + \mathsf{void}$ (avec $t \in \{\mathsf{bool}, \mathsf{int}\}$).

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

Instructions La règle de typage de l'affectation (SET) ne change pas. La règle typage de l'alternative réalise notre première conclusion. La règle de typage de la boucle doit tenir compte du fait que le corps d'une boucle peut ne pas être exécutée.

(IF0) pour tout type t, si $G \vdash_{\text{Expr}} e : \mathsf{bool}$ 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 $t \neq \mathsf{void}$, si $G \vdash_{\text{Expr}} e : \mathsf{bool}$ et $G \vdash_{\text{Block}} blk_1 : \mathsf{void}$ et $G \vdash_{\text{Block}} blk_2 : t$ alors $G \vdash_{\text{Stat}} (\texttt{IF}\ e\ blk_1\ blk_2) : t + \mathsf{void}$

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

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

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

La commande `RETURN`

(RET) pour toute expression e, si $G \vdash e : t$ alors $G \vdash_{\text{CMDS}} (\texttt{RETURN}\ e) : t$

Par définiton du taypage des expressions, t ici est nécessairement bool ou int.

Suites de commandes Il faut réaliser ici notre seconde conclusion.

(STAT0) si $G \vdash_{\text{STAT}} s : \mathsf{void}$ et $G \vdash_{\text{CMDS}} cs : t$ alors $G \vdash_{\text{CMDS}} s\texttt{;}cs : t$

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

(CONST) si d est une déclaration de la forme $(\texttt{CONST}\ x\ t\ e)$, si $G \vdash e : t$ et $G;(x : t) \vdash_{\text{CMDS}} cs : t'$ alors $G \vdash_{\text{CMDS}} d\texttt{;}cs : t'$

(VAR) si d une déclaration de la forme $(\texttt{VAR}\ x\ t)$, si $G;(x : t) \vdash_{\text{CMDS}} cs : t'$ alors $G \vdash_{\text{CMDS}} d\texttt{;}cs : t'$

Il convient ici également de compléter les règles pour les cas de suite réduite à une seule commande: les règles à appliquer sont simplement celles dédiées aux instruction et à la commande `RETURN`

Analyse du flot de contrôle Nos règles de typage rejètent les programmes contenant une suite commençant par une instruction dont le type est soit bool soit int. Par exemple, la suite

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

est rejetée. Ce qui est une bonne chose si l'on veut interdire l'occurrence de code mort. En effet si l'on peut assigner le type bool ou le type int à une instruction, c'est que tout ses chemins d'exécution rencontrent un `RETURN` et que le code qui la suit est du code mort.

En revanche, nous accepterons

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

si dans la suite du programme nous obtenons un type cohérent avec celui de `x`.

3.3 Fonctions procédurales

Le terrain est à présent prêt pour atteindre notre objectif: définir une fonction dont le corps est un bloc contenant une suite de commandes. Ces fonctions pourront être utilisées comme opérateurs dans les expressions.

3.3.1 Syntaxe

Lexique on ajoute le mot clé `FUN`

Grammaire On ajoute aux déclarations la possibilité de définir une fonction.

DEC ::= ...
| FUN ident TYPE [ARGS] PROG
| FUN ident TYPE [] PROG

On a déjà avec *APS1* la possibilité d'appliquer une fonction. On peut y ajouter le cas des fonctions sans argument:

EXPR ::= ...
| (ident)

Exemple:

```
FUN z int [f:int -> int, a:int, b:int] [
  VAR x int;
  SET x a;
  WHILE (lt x b) [
    IF (eq (f x) 0)
      [ RETURN x ]
      [ SET x (add x 1) ]
  ]
  RETURN 0
]
```

3.3.2 Typage

Définition L'analyse de type d'une définition de fonction intervient lors de l'analyse de type d'une suite de commandes. On y ajoute la règle:

(FUN) si $G, x_1 : t_1; \ldots; x_n : t_n \vdash_{\text{CMDS}} cs : t$ et si $G; f : (t_1 * \ldots * t_n)$ `->` $t) \vdash_{\text{CMDS}} cs' : t'$
alors $G \vdash_{\text{CMDS}}$ (`FUN` f t `[`$x_1 : t_1, \ldots, x_n : t_n$`]` `[`$cs$`]` `;` cs') $: t'$

(FUN0) si $G \vdash_{\text{CMDS}} cs : t$ et si $G; f : ()$ `->` $t \vdash cs' : t'$ alors $G \vdash_{\text{CMDS}}$ (`FUN` f t `[]` `[`cs`]` `;` cs') $: t'$

Nous distinguons le cas des fonctions sans argument dont nous notons le type () `->` t.

Application Nous avons déjà dans *APS1* une règle de typage pour l'application. Nous pouvons l'étendre au cas des fonctions sans argument:

(APP0) si $G \vdash_{\text{EXPR}} f : ()$ `->` t alors $G \vdash_{\text{EXPR}}$ `(`f`)` $: t$

3.3.3 Sémantique opérationnelle

Définition Dans *APS1* nous avons défini des fermetures pour les procédures. Celles-ci conviennent parfaitement pour donner une valeur au fonctions «procédurales». La règle sémantique associée à une déclaration de fonction est très similaire à celle des déclarations de procédures:

(FUN) $r, m \vdash_{\text{DEC}}$ `FUN` x t `[`x_1`:`$t_1, \ldots, x_n$`:`t_n`]` `[`cs`]` $\rightsquigarrow r[x = \langle cs, r + [x_1, \ldots, x_n]\rangle], m$

Impact sur la sémantique des expressions L'application des fonctions définies avec un bloc de commandes est proche de celles définies avec une expression focntionnelle. Toutefois leur utilisation comme élément d'une expression a un impact important sur la sémantique des expressions.

En effet, même si cela n'est pas recommandé, les fermetures procédurales sont suceptibles d'affecter des variables globales et donc d'avoir un effet sur la mémoire. Donc, une expression construite par application d'une fermeture procédurale, outre délivrer une valeur, elle peut avoir un effet sur la mémoire. Ce qui a pour conséquence qu'il nous faut modifier la signature de la relation sémantique pour les expressions:

$$E \times S \times \text{EXPR} \times (V \times S)$$

Un autre impact est que la sémantique va désormais expliciter *l'ordre d'évaluation des arguments*. En effet, dans l'application d'un opérateur binaire, l'évaluation d'un paramètre affecte l'état mémoire avec lequel l'autre est évalué.

Expressions booléennes Ici, l'impact est simplement l'apparition des modifications d'états mémoire car les opérateurs booléens binaires ont été définis en *APS0* comme séquentiels.

(NOT1) si $r, m \vdash_{\text{Expr}} e \rightsquigarrow (\#t, m')$ alors $r, m \vdash_{\text{Expr}} (\texttt{not}\ e) \rightsquigarrow (\#f, m')$

(NOT2) si $r, m \vdash_{\text{Expr}} e \rightsquigarrow (\#f, m')$ alors $r, m \vdash_{\text{Expr}} (\texttt{not}\ e) \rightsquigarrow (\#t, m')$

(OR1) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (\#t, m_1)$ alors $r, m \vdash_{\text{Expr}} (\texttt{or}\ e_1\ e_2) \rightsquigarrow (\#t, m_1)$

(OR2) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (\#f, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v, m_2)$ alors $r, m \vdash_{\text{Expr}} (\texttt{or}\ e_1\ e_2) \rightsquigarrow (v, m_2)$

(AND1) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (\#f, m_1)$ alors $r, m \vdash_{\text{Expr}} (\texttt{and}\ e_1\ e_2) \rightsquigarrow (\#f, m_1)$

(AND2) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (\#t, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v, m_2)$ alors $r, m \vdash_{\text{Expr}} (\texttt{and}\ e_1\ e_2) \rightsquigarrow (v, m_2)$

Comparaisons

(EQ1) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v_1 = v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{eq}\ e_1\ e_2) \rightsquigarrow (\#t, m_2)$

(EQ2) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v_1 \neq v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{eq}\ e_1\ e_2) \rightsquigarrow (\#f, m_2)$

(LT1) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v_1 < v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{lt}\ e_1\ e_2) \rightsquigarrow (\#t, m_2)$

(LT2) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v_2 \leq v_1$ alors $r, m \vdash_{\text{Expr}} (\texttt{lt}\ e_1\ e_2) \rightsquigarrow (\#f, m_2)$

Nous avons fait le choix de toujours évaluer les arguments de gauche à droite. Il aurait pu être autre. En Ocaml, par exemple, les arguments sont évalués de droite à gauche.

Opérations arithmétiques

(ADD) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v = v_1 + v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{add}\ e_1\ e_2) \rightsquigarrow (v, m_2)$

(SUB) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v = v_1 - v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{sub}\ e_1\ e_2) \rightsquigarrow (v, m_2)$

(MUL) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v = v_1 \times v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{mul}\ e_1\ e_2) \rightsquigarrow (v, m_2)$

(DIV) si $r, m \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $r, m_1 \vdash_{\text{Expr}} e_2 \rightsquigarrow (v_2, m_2)$ et si $v = v_1 \div v_2$ alors $r, m \vdash_{\text{Expr}} (\texttt{div}\ e_1\ e_2) \rightsquigarrow (v, m_2)$

Applications

(APP1) si $r, m \vdash_{\text{Expr}} e \rightsquigarrow (\langle e', r' + [x_1, \ldots, x_n] \rangle, m_0)$, si $r, m_0 \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $\ldots$ et $r, m_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, m_n)$ et si $r'[x_1 = v_1] \ldots [x_n = v_n], m_n \vdash_{\text{Expr}} e' \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{Expr}} (e\ e_1 \ldots e_n) \rightsquigarrow (v, m')$.

(APP2) si $r, m \vdash_{\text{Expr}} f \rightsquigarrow (\langle cs, r' + [x_1, \ldots, x_n] \rangle, m_0)$, si $r, m_0 \vdash_{\text{Expr}} e_1 \rightsquigarrow (v_1, m_1)$ et $\ldots$ et $r, m_{n-1} \vdash_{\text{Expr}} e_n \rightsquigarrow (v_n, m_n)$ et si $r'[x_1 = v_1] \ldots [x_n = v_n], m_n \vdash_{\text{Expr}} cs \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{Expr}} (f\ e_1 \ldots e_n) \rightsquigarrow (v, m')$.

Les instructions sont également affectées puisque leur évaluation demande celle d'une expression.

(SET) si $r(x) = @a$ et si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{STAT}} (\texttt{SET}\ x\ e) \rightsquigarrow (\varepsilon, m'[a := v])$

(ALT1) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (\#t, m')$ et $r, m' \vdash_{\text{BLOCK}} blk_1 \rightsquigarrow (v, m'')$ alors $r, m \vdash_{\text{STAT}} (\texttt{IF}\ e\ blk_1\ blk_2) \rightsquigarrow (v, m'')$

(ALT0) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (\#f, m')$ et $r, m' \vdash_{\text{BLOCK}} blk_2 \rightsquigarrow (v, m'')$ alors $r, m \vdash_{\text{STAT}} (\texttt{IF}\ e\ blk_1\ blk_2) \rightsquigarrow (v, m'')$

(LOOP0) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (\#f, m')$ alors $r, m' \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (\varepsilon, m')$

(LOOP1) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (\#t, m')$ et $r, m' \vdash_{\text{BLOCK}} blk \rightsquigarrow (\varepsilon, m'')$ et $r, m'' \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, m''')$ alors $r, m \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, m''')$

(LOOP2) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (\#t, m')$ et $r, m' \vdash_{\text{BLOCK}} blk \rightsquigarrow (v, m'')$, avec $v \neq \varepsilon$ alors $r, m \vdash_{\text{STAT}} (\texttt{WHILE}\ e\ blk) \rightsquigarrow (v, m'')$

Définitions La définition d'une constante doit être amendée puisque celle-ci entraîne l'évaluation d'une expression.

(CONST) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{DEC}} (\texttt{CONST}\ x\ t\ e) \rightsquigarrow (r[x = \#v], m')$

La commande `RETURN` Ses règles doivent également être amendées puisque cette commande demande l'évaluation d'une expression.

(RET) si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow (v, m')$ alors $r, m \vdash_{\text{CMDS}} \texttt{RETURN}\ e \rightsquigarrow (v, m')$

Blocs et programme Dans *APS1* les blocs et les programmes sont traités par une seule règle et le type assigné à un bloc est le type assigné à la suite de commandes qui le constitue. Si nous conservons cette manière, nous acceptons des *programmes* (le bloc principal) pouvant avoir n'importe quel type.

C'est ici encore un point de *conception de langage*: voulons nous cela ou préférons nous, par analyse de type forcer un programme à n'avoir que le type `void`. Si nous faisons ce choix, nous interdisons, es programmes pouvant exécuter

3.4 Sémantique dénotationnelle

Pour donner la sémantique dénotationnelle de *APS2*, nous allons utiliser un trait de programmation fonctionnelle avancé: *les continuations*. Elles permettent une expression fonctionnelle de la séquentialité ainsi que de sa rupture introduite en *APS2* par la commande `RETURN`.

3.4.1 Les continuations

Soit l'expression `(mul (add 9 5) 2)`. Dans le processus d'évaluation de cette expression, il faut évaluer la sous-expression `(add 9 5)` puis multiplier par 2 le résultat obtenu, c'est-à-dire appliquer la fonction `[n:int](mult n 2)` à ce résultat. On dit alors que `[n:int](mult n 2)` est *la continuation* de `(add 9 5)`.

Imaginons maintenant que l'addition prenne un troisième argument, un argument fonctionnel qui est sa continuation. Posons `addc : int -> int -> (int -> τ)` avec τ un type quelconque, telle que

$$\texttt{(addc}\ e_1\ e_2\ k\texttt{)} = \texttt{(}k\ \texttt{(add}\ e_1\ e_2\texttt{))}.$$

Imaginons que l'on généralise cela aux autres fonctions primitives et que l'on ait également

`mulc : int -> int -> (int -> τ)`, avec $\texttt{(mulc}\ e_1\ e_2\ k\texttt{)} = \texttt{(}k\ \texttt{(mul}\ e_1\ e_2\texttt{))}$

Alors, l'expression `(mul (add 9 5) 2)` a la même valeur que `(addc 9 5 ([n:int](mulc n 2 ([n:int]n))))`.

De manière générale, une continuation est une fonction qui prend en argument un résultat de calcul pour en produire un autre.

Nous allons appliquer cette idée à la sémantique dénotationnelle des expressions. Et nous verrons comment elle fournit une manière de prendre en compte les fonctions procédurales qui font usage du `RETURN`. Mais avant cela, nous allons faire un détour par une propriété du langage qui sert de base à la défintion des fonctions sémantqiues: le λ-calcul.

3.4.2 (Dé)curryfication

Une fonction à 2 arguments, comme `[x1:int, x2:int](`F `x1 x2)` peut toujours être vue comme une fonction à 1 argument dont le corps est une fonction à 1 argument: `[x1:int][x2:int](`F `x1 x2)`. L'application `(([x1:int, x2:int](`F `x1 x2))` v_1 v_2`)` se transforme alors en `((([x1:int][x2:int](`F `x1 x2))` v_1`)` v_2`)`. Ces deux applications ont même valeur.

On peut donc considérer l'expression fontionnelle `[`x_1`:`τ_1`, ...,`x_n`:`τ_n`]`e comme une abréviation de `[`x_1`:`τ_1`] ...[`x_n`:`τ_n`]`e et l'application `(`e e_1 $\ldots e_n$`)` comme une abréviation des applications imbriquées `(...(`e e_1`) ...`e_n`)`. Nous appliquerons le même traitement à toutes les abstractions: expressions procédurale, définition des fonctions procédurales. Pour l'appel de procédure, `CALL` x $e_1 \ldots e_n$ est vu comme une abréviation de `CALL (...(`x e_1`)...`e_n`)`.

Cette manière nous permet de n'avoir à considérer que les cas simples d'abstraction (`[`x`:`τ`]`e) et d'application `(`e_1 e_2`)`.

3.4.3 Sémantique à continuation

Nous allons enrichir la fonction sémantique pour les expressions d'un argument supplémentaire: une continuation.

Domaines Le résultat attendu d'une expression est un couple constitué d'une valeur et d'un état mémoire: $V \times S$. Les continuations pour la fonction sémantique d'évaluation des expression prennent en argument un résultat pour en produire un autre, si l'on appelle C le domaine des continuations, on pose:

$$C = (V \times S) \to (V \times S)$$

Pour alléger l'écriture des continuations, nous écrirons $\lambda(v, s).t$ pour $\lambda c.$ let $(v, s) = c$ in t.

Pour *APS1*, nous avons réprsenté les fermetures comme une paire encapsulant une expression e et une fonction de mise à jour de l'environnement ($\lambda v.\rho[\mathtt{x} = v]$). Une fermeture $\langle e, r\rangle$ est utilisée pour évaluer e avec l'environnement lis-à-jour avec une valeur, disons n: $\mathbf{E}[[e]](r\ n)\sigma$. En collapsant ces 2 phases, on aurait pu représenter les fermetures de manière purement fonctionnelle: $\lambda v.\lambda s.\mathbf{E}[[e]](\rho[\mathtt{x} = v])s$. Nous allons adopter cette manière, sachant qu'à présent, il nous faut également une continuation. L'ensemble des fermetures devient donc

- $F_e = \textsc{Expr} \to E \to S \to C \to (V \times S)$
- $F_p = \textsc{Prog} \to E \to S \to C \to (V \times S)$

  $F = Fe \cup F_p$

Pour réaliser la rupture de calcul des `RETURN`, nous allons *capturer* les continuations dans l'environnement. On a donc désormais

$$E = A \oplus V \oplus C$$

Expressions La fonction **E** a pour nouvelle signature

$$\mathbf{E} : \mathsf{Expr} \to E \to S \to C \to (V \times S)$$

Dans le calcul d'une expressions, les expressions atomiques sont les cas de base de la définition récursive de **E**. Elles donnent directement un résultat. Dans la sémantique à continuation, ce résultat est passé à la continuation courante:

$$\mathbf{E}[[b]]\rho\sigma\kappa = \kappa(\beta(b))$$

$$\mathbf{E}[[n]]\rho\sigma\kappa = \kappa(\nu(n))$$

$$\begin{array}{lcl}\mathbf{E}[[x]]\rho\sigma\kappa & = & \text{case } \rho(x) \text{ of} \\ & & \quad inA(a) : \kappa\ (\sigma(a)) \\ & & \mid inV(v) : \kappa\ v\end{array}$$

Pour évaluer l'application (`not` e), on évalue d'abord e puis on calcule la négation de la valeur obtenue avec le combinateur if. Dans la sémantique à continuation, la séquentialité est obtenue en donnat à l'évaluataion de e le calcule de la négation comme continuation. Celle-ci sera de plus combinée avec la continuation courante:

$$\begin{array}{lcl}\mathbf{E}[[\texttt{not } e]]\rho\sigma\kappa & = & \text{let } \kappa' = \lambda(v,s).(\kappa\ (\mathsf{if}\ v\ \mathsf{ff}\ \mathsf{tt})) \text{ in} \\ & & \quad \mathbf{E}[[e]]\rho\sigma\kappa'\end{array}$$

Regardons à présent l'application d'un opérateur binaire, il y a trois phases de calcul: évaluation du premier argument, évaluation du second et calcul du résulat de l'opération. Avec les continuation, la valeur de l'application `(add` e_1 e_2`)` s'obtient en évaluation e_1 avec pour continuation l'évaluation de e_2 qui a elle-même pour continuation le calcule de la somme combinée avec la continuation courante. Cela donne:

$$\begin{array}{lcl}\mathbf{E}[[\texttt{add } e_1\ e_2]]\rho\sigma\kappa & = & \text{let } \kappa' = \lambda(v_1,s_1).\mathbf{E}[[e_1]]\rho s_1(\lambda(v_2,s_2).(\kappa\ ((v_1 +_V v_2), s_2))) \text{ in} \\ & & \quad \mathbf{E}[[e_2]]\rho\sigma\kappa'\end{array}$$

Les opérateurs binaires booléens sont des opérateurs séquentiels. Si nous leur appliquons la méthode d'évaluation ci-dessus, nous perdons cette propriété. Pour la retrouver, il faut utiliser le combinatuer if sur le résultat de l'évalauation du premier argument. Cela donne:

$$\begin{array}{lcl}\mathbf{E}[[\texttt{and } e_1\ e_2]]\rho\sigma\kappa & = & \text{let } \kappa' = \lambda(v_1,s_1).(\mathsf{if}\ v_1\ (\mathbf{E}[[e_1]]\rho s_1(\lambda(v_2,s_2).(\kappa\ (v_2,s_2))))\ (\kappa\ (\mathsf{ff},s_1))) \text{ in} \\ & & \quad \mathbf{E}[[e_1]]\rho\sigma\kappa'\end{array}$$

Pour l'application en général, nous avons observé que nous pouvons nous contenter du cas unaire: $(e_1\ e_2)$. Le schéma de définition de la sémantique de l'appliaction est donc assez proche de celui de la négation à cette différence que «l'opération» à appliquer ne sera connu que lorsque e_1 sera évalué. La continuation de l'évaluation de l'argument e_2 sera donc l'évalaution de la «fonction» e_1 qui a elle-même pour continuation l'application du résultat de e_1 à la valeur de e_2.

$$\begin{array}{lcl}\mathbf{E}[[(e_1\ e_2)]]\rho\sigma\kappa & = & \text{let } \kappa' = \lambda(v_2,s_2).\mathbf{E}[[e_1]]\rho s_2(\lambda(s_1,v_1).(v_1\ v_2\ s_1\ \kappa)) \text{ in} \\ & & \quad \mathbf{E}[[e_2]]\rho\sigma\kappa'\end{array}$$

Reste à examiner le cas des abstractions. Ici également, nous pouvons nous contenter du cas «unaire». Une abstraction donne simplement une fermeture couplée à l'état mémoire courant. C'est une valeur «immédiate» que l'on passe à la continuation courante.

$$\begin{array}{lcl}\mathbf{E}[[\texttt{[}x\texttt{:}\tau\texttt{]}e]]\rho\sigma\kappa & = & \text{let } f = \lambda v.\lambda s.\lambda k.\mathbf{E}[[e]](\rho[x=v])sk \text{ in} \\ & & \quad (\kappa\ (f,\sigma))\end{array}$$

Fonctions procédurales Nous avons introduit dans *APS2* la possibilité de définir une fonction de manière procédurale. Le résultat de la fonction est communiqué à l'appelant par invocation de la commande `RETURN`. De telles fonctions sont définies par la construction syntaxique `[`x_1`:`τ_1`, ..., `x_n`:`τ_n`] [`cs`]` que l'on peut considérer comme une abréviation pour `[`x_1`:`τ_1`] ... [`x_n`:`τ_n`] [`cs`]`. On peut définir ces constructions syntaxiques de la manière suivante:

EXPRO ::= `[` CMDS `]`
| `[` idents `:` TYPE `]` EXPRO

De ces *expressions procédurales*, nous devons tirer une fermeture «procédurale». Sur le modèle des fermeture purement fonctionnelles, on définit leur construction par récurrence sur les expressions procédurales. La construction commence par transformer les abstractions en opération de mise-à-jour de l'environnement et s'achève par l'invocation de l'évaluation du bloc de commandes.

Lorsque la suite de commandes du bloc qui définit une fonction est évaluée, elle peut délivrer une valeur avant la fin effective de la suite. Dans ce cas, la valeur émise doit être récupérée par la continuation du bloc. C'est ici que nous utilisons la mise en mémoire d'une continuation dans l'environnement. Nous en verrons l'usage lorsque nous définirons la sémantique de `RETURN`.

Pour construire nos fermetures procédurales, nous nous donnons une nouvelle fonction sémantique

$$\mathbf{F} : \textsc{Expro} \to E \to S \to C \to (V \times S)$$

$$\begin{array}{lcl} \mathbf{F}[[\texttt{[}x\texttt{:}\tau\texttt{]}\,p]]\rho\sigma\kappa & = & \text{let } f = \lambda v.\lambda s.\lambda k.\mathbf{F}[[p]](\rho[x = v])sk \text{ in} \\ & & (\kappa\ (f, \sigma)) \end{array}$$

$$\mathbf{F}[[\texttt{[}cs\texttt{]}]]\rho\sigma\kappa \quad = \quad \mathbf{Cs}[[cs]](\rho[\texttt{*ret*} = inC(\kappa)])\sigma\kappa$$

La première clause de la définition de **F** est proche de celle de **E** pour les abstrations fonctionnelles. La seconde passe le relai à la fonction sémantique **Cs** pour l'évaluation des suites de commandes avec un environnement où a été mémorisée, sous le nom – arbitrairement choisi – `*ret*`, la continuation courante du bloc.

Instructions $\mathbf{S} : \textsc{Stat} \to E \to S \to C \to V \times S$

Pour évaluer l'affectation `SET` x e, on évalue e pour modifier l'état mémoire. L'affectation a pour *effet* une modification d'état mémoire qui est passé à la continuation courante.

$$\begin{array}{lcl} \mathbf{S}[[\texttt{SET}\ x\ e]]\rho\sigma\kappa & = & \text{let } inA(a) = \rho(x) \text{ in} \\ & & \mathbf{E}[[e]]\rho\sigma(\lambda(v, s).(\kappa\ (\varepsilon, s[a := v]))) \end{array}$$

Pour évaluation l'alternative `IF` e `[`cs_1`]` `[`cs_2`]`, on évalue e. Le résltat de cette évaluation est passé à la continuation qui va sélectionner l'un des bloc de l'alternative.

$$\mathbf{S}[[\texttt{IF}\ e\ \texttt{[}cs_1\texttt{]}\ \texttt{[}cs_2\texttt{]}]]\rho\sigma\kappa \quad = \quad \mathbf{E}[[e]]\rho\sigma(\lambda(v, s).(\text{if } v\ (\mathbf{Cs}[[cs_1]]\rho\ s\ \kappa)\ (\mathbf{Cs}[[cs_2]]\rho\ s\ \kappa)))$$

L'évaluation de la boucle `WHILE` e `[`cs`]` est aussi envisagée en terme de continuation du résultat de l'évalaution de e. Mais il faut ici également tenir compte du mécanisme d'itération. Nous avions vu comment l'obtenir avec un point fixe. dans la sémantique à continuation, la suite des calculs est dans la continuation, donc pour itérer, c'est la continuation qu'il faut itérer. La sémantique de la boucle est définie comme un point fixe sur les continuations.

$$\mathbf{S}[[\texttt{WHILE}\ e\ \texttt{[}cs\texttt{]}]]\rho\sigma\kappa \quad = \quad ((!k.\lambda s.\mathbf{E}[[e]]\rho\ s\ (\lambda(v, s').(\text{if } v\ (\mathbf{Cs}[[cs]]\rho\ s'\ k)\ (\kappa\ (v, s'))))\ \sigma)$$

Suite de commandes

$$\mathbf{Cs} : \textsc{Cmds} \to E \to S \to C \to (V \times S)$$

A tout seigneur tout honneur, examinons en premier le cas de la suite réduite à la seule commande `RETURN`. Cette commande ne peut être que la dernière d'une suite. Elle donne une valeur qui est passée à la continuation mémorisée dans l'environnement, oubliant ainsi la continuation courante. C'est ainsi qu'est réalisée la *rupture de séquentialité* de la commande `RETURN`.

$$\begin{array}{lcl} \mathbf{Cs}[[\texttt{RETURN}\ e]]\rho\sigma\kappa & = & \text{let } inC(k) = \rho(\texttt{*ret*}) \text{ in} \\ & & \mathbf{E}[[e]]\rho\sigma(\lambda(v, s).(k\ (v, s))) \end{array}$$

Lorsque la suite est réduire à une seule instruction s, la valeur de la suite est celle de l'instruction

$$\mathbf{Cs}[[s]]\rho\sigma\kappa \quad = \quad \mathbf{S}[[s]]\rho\sigma\kappa$$

La valeur d'une suite de longueur supérieure à un strictement ey qui commence par une instruction s'obtient en évaluant l'instruction avec pour continuation l'évaluation du reste de la suite.

$$\mathbf{Cs}[[s\,;cs]]\rho\sigma\kappa \quad = \quad \mathbf{S}[[s]]\rho\sigma(\lambda(v,s).\mathbf{Cs}[[cs]]\rho s\kappa)$$

Notez que si l'évaluation de s rencontre une commande `RETURN`, la continuation $\lambda(v,s).\mathbf{Cs}[[cs]]\rho s\kappa$ est ignorée.

Pour les déclarations de constantes, on évalue l'expression définissant la constante avec pour continuation, l'évaluation du reste de la suite avec un environnement modifié en fonction du résultat obtenu pour l'expression.

$$\mathbf{Cs}[[\texttt{CONST}\ x\ \tau\ e\texttt{;}\ cs]]\rho\sigma\kappa \quad = \quad \mathbf{E}[[e]]\rho\sigma(\lambda(v,s).\mathbf{Cs}[[cs]](\rho[x = inV(v)])\ s\ \kappa$$

Pour les déclarations de variables, on évalue simplement le reste de la suite avec les modifications apportée à l'environnement et à l'état mémoire.

$$\begin{aligned}\mathbf{Cs}[[\texttt{VAR}\ x\ \tau\texttt{;}\ cs]]\rho\sigma\kappa \quad = \quad & \text{let } a,\sigma' = alloc(\sigma) \text{ in} \\ & \mathbf{Cs}[[cs]](\rho[x = inA(a)])\sigma'\kappa\end{aligned}$$

Enfin, pour les déclarations de fonctions procédurales, on invoque notre nouvelle fonction sémantique **F** avec pour continuation, l'évaluation du reste de la suite avec un environnement mis à jour.

$$\mathbf{Cs}[[\texttt{FUN}\ x\ \tau\ p\texttt{;}\ cs]]\rho\sigma\kappa \quad = \quad \mathbf{F}[[p]]\rho\sigma(\lambda(v,s).\mathbf{Cs}[[cs]](\rho[x = inF(v)])s\kappa)$$

avec p expression procédurale.

La sémantique de la définition de procédure et de leur invocation est laissée en exercice.

Bloc $\mathbf{B} : \textsc{Prog} \to E \to S \to C \to V \times S$

$$\mathbf{B}[[cs]]\rho\sigma\kappa \quad = \quad \mathbf{Cs}[[cs]]\rho\sigma(\lambda(v,s).(\kappa\ (v, s/\rho))$$

Programme Un programme est un bloc dont nous ne voulons pas oublier le résultat. C'est-à-dire dont nous voulons conserver l'effet produit par son évaluation sur l'état mémoire. Cet effet est double: allocation et modification.

L'évaluation d'un programme est l'évaluation de la suite de commandes qui le compose en partant d'un environnement initial ρ_0, d'une mémoire initiale σ_0 et de la continuation initial $\kappa_0 = \lambda x.x$ qui nous permettra de récupérer l'état mémoire final. La valeur du programme `[`cs`]` est

$$\mathbf{Cs}[[cs]]\rho_0\sigma_0\kappa_0$$