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

P. Manoury

Janvier 2017

2 Le langage APS1: expressions fonctionnelles

Le langage *APS1* est simplement l'extesion du noyau *APS0* avec des *expressions fonctionnelles*.

Le langage *APS0* ne fournit pas l'opérateur de comparaison *«inférieur ou égal»*. Toutefois, pour toutes expressions arithmétique e_1, e_2, on peut calculer la valeur de cette comparaison en combinant les opérateurs primitifs `lt`, `eq` et `or`: `(or (lt` e_1 e_2`) (eq` e_1 e_2`))`.

En particulier, si l'on considère deux symboles de variables, disons, `x` et `y`, l'expression `(or (lt x y) (eq x y))` permet de calculer le résultat de la comparaison *«inférieur ou égal»* pour toute valeur que l'on voudra donner à `x` et à `y`; au moyen d'une affectation, par exemple.

Ainsi, la valeur de l'expression `(or (lt x y) (eq x y))` où les valeurs de `x` et de `y` sont indéterminées est appelée *fonction* de `x` et de `y`. On rend explicite cette fonction par *abstraction* de `x` et `y` comme *paramètres*. On forme ainsi une *expression fonctionnelle*.

Syntaxe: On écrit: `[x:int, y:int](or (lt x y) (eq x y))`.

Typage: La fonction ainsi définie a deux arguments de type `int` et donne une valeur de type `bool`. C'est une fonction de type `int * int -> bool`.

Sémantique: L'expression fonctionnelle `[x:int, y:int](or (lt x y) (eq x y))` a pour valeur une ...fonction. Du point de vue machine, une fonction est constitué de l'expression du *corps de la fonction* (ici, l'expression `(or (lt x y) (eq x y))`) et d'un environnement «à trou» où viendront prendre place la valeurs des paramètres. On appelle cette structure une *fermeture*.

Les fonctions sont destinées à être *appliquées* à des arguments.

Syntaxe: L'application est notée: `([x:int, y:int](or (lt x y) (eq x y))` $\texttt{e}_1$ $\texttt{e}_2$`)`, où $\texttt{e}_1$ et $\texttt{e}_2$ sont des expressions.

Typage: Si les expressions $\texttt{e}_1$ et $\texttt{e}_2$ sont de type `int`, alors l'application `([x:int, y:int](or (lt x y) (eq x y))` $\texttt{e}_1$ $\texttt{e}_2$`)` est de type `bool` (puisque la fonction appliquée est de type `int * int -> bool`).

Sémantique L'application `([x:int, y:int](or (lt x y) (eq x y))` $\texttt{e}_1$ $\texttt{e}_2$`)` a pour valeur celle de l'expression `(or (lt x y) (eq x y))` dans un environnement où `x` est lié à la valeur de $\texttt{e}_1$ et `y`, à celle de $\texttt{e}_2$.

2.1 Syntaxe

Lexique On ajoute au lexique de *APS0* les symboles réservés `:`, `*` et `->`.

Grammaire On étend la grammaire de *APS0* de la manière suivante:

EXPR	::=	...
	|	`[` ARGS `]` EXPR
	|	`(` EXPRS `)`
EXPRS	::=	EXPR
	|	EXPR EXPRS
ARGS	::=	ARG
	|	ARG `,` ARGS
ARG	::=	ident `:` TYPE
TYPE	::=	...
	|	`(` TYPES `->` TYPE `)`
TYPES	::=	TYPE
	|	TYPE `*` TYPES

Remarques Cette seule extension permet des *déclarer* des fonctions. En effet, le programme

```
[
  VAR x bool;
  CONST le (int * int -> bool)
           [x:int, y:int](or (lt x y) (eq x y));
  IF (le 0 1)
    [ SET x true; ]
    [ SET x false; ]
]
```

est syntaxiquement correct. Notez que ce programme

```
[
  VAR x bool;
  VAR le (int * int -> bool);
  SET le [x:int, y:int](or (lt x y) (eq x y));
  IF (le 0 1)
    [ SET x true; ]
    [ SET x false; ]
]
```

est aussi syntaxiquement correct.

2.2 Typage

Il y a deux règles à ajouter à celles de *APS0*: la règle de typage des expressions fonctionnelles et la règle de typage de l'application.

(ABS) Si $G; x_1 : t_1; \ldots; x_n : t_n \vdash_{\text{EXPR}} e : t$ alors $G \vdash_{\text{EXPR}} [x_1 : t_1, \ldots, x_n : t_n]e : t$

(APP) Si $G \vdash_{\text{EXPR}} e_1 : t_1$ et $\ldots$ et $G \vdash_{\text{EXPR}} e_n : t_n$ et si $G \vdash_{\text{EXPR}} e : t_1 * \ldots * t_n \texttt{->} t$ alors $G \vdash_{\text{EXPR}} (e\ e_1 \ldots e_n) : t$

2.3 Sémantique opérationnelle

Les règles sémantiques pour les expressions leur associent une *valeur*. La valeur d'une expression fonctionnelle (une abstraction) est une *fermeture* constituée d'une expression (le corps de la fonction), d'un environnement et de la liste des paramètres de la fonction. On ajoute donc à celles de *APS0* une règle sémantique pour décrire la valeur d'une abstraction et une seconde pour donner la valeur le l'application d'une abstraction.

Abstraction Soit e une expression (corps d'une fonction), r un environnement et $x_1, \ldots, x_n$ des identificatuers (paramètres d'une fonction), on note $\langle e, r+[x_1, \ldots, x_n]\rangle$ la fermeture constituée de ces trois éléments.

(ABS) $r, m \vdash_{\text{EXPR}} \texttt{[}x_1 : t_1, \ldots, x_n : t_n\texttt{]}e \rightsquigarrow \langle e, r + [x_1, \ldots, x_n]\rangle$

Application Une application `(`$e\ e_1 \ldots e_n$`)` n'a de valeur que si e a pour valeur une fermeture.

(APP) Si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \langle e', r' + [x_1, \ldots, x_n]\rangle$, si $r, m \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1$ et $\ldots$et $r, m \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1$ et si $r'[x_1 = v_1] \ldots [x_n = v_n], m \vdash_{\text{EXPR}} e' \rightsquigarrow v$ alors $r, m \vdash_{\text{EXPR}}$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow v$.

Liaison statique et liaison dynamique

La définition que nous donnons des fermetures fait un choix de conception de langage pour `APS1`: celui de la *liaison statique* pour la définition de fonctions. En effet, si l'on évalue les déclarations suivante `CONST a int 1; CONST f (int -> int) [x:int](add x a)`, dans un contexte d'exécution r, m, on obtient dans un premier temps, l'environnement $r[\texttt{a} = \#1]$; puis $r[\texttt{a} = \#1][\texttt{f} = \langle \texttt{(add x a)}, [\texttt{a} = \#1] + [\texttt{x}]$.

Pour simplifier, on écrira $[x_1 = w_1; x_2 = w_2; \ldots; x_n = w_n]$ à la place de $[x_1 = w_1][x_2 = w_2] \ldots [x_n = w_n]$.

Considérons maintenant le programme suivant

```
[
  CONST a int 1;
  CONST f (int -> int) [x:int](add x a);
  VAR r int;
  CONST a int 42;
  SET x (f 5)
]
```

Après l'évaluation des quatre déclaration, l'environnement obtenu est

$$[\texttt{a} = \#1; \texttt{f} = \langle \texttt{(add x a)}, [\texttt{a} = \#1] + [\texttt{x}]\rangle; \texttt{x} = @a_1; \texttt{a} = \#42]$$

La mémoire est $[a_1 = \mathit{new}]$.

Pour évaluer l'affectation `SET x (f 5)`, on évalue l'application `(f 5)` dans cet environnement. La valeur de `f` y est la fermeture $\langle \texttt{(add x a)}, [\texttt{a} = \#1] + [\texttt{x}]\rangle$, la valeur de `5` est 5, donc la valeur de l'application est la valeur de l'expression `(add x a)` (premier terme de la fermeture) dans l'environnement $[\texttt{a} = \#1; \texttt{x} = \#5]$, construit à partir du second terme de la fermeture et de la valeur de l'argument `5`. Ce qui donne la valeur 6. La liaison est ici statique car la valeur de la constante `a` utilisée dans le corps de `f` a été figée dans la fermeture au moment de la déclaration de `f`. La seconde valeur de `a` n'interviens pas dans le calcul de `(f 5)` bien que celui-ci intervienne après la seconde déclaration de `a`.

Si l'on modifie la définition des fermeture en excluant la capture de l'environnement présent au moment de l'évaluation des abstraction, on obtient un langage à *liaison dynamique*. Une fermeture est alors réduite au corps de la fonction et la liste de ses paramètres formels. On écrit $\langle e, +[x_1, \ldots, x_n]\rangle$. La règle de l'application y est légèrement différente:

si $r, m \vdash_{\text{EXPR}} e \rightsquigarrow \langle e', +[x_1, \ldots, x_n]\rangle$, si $r, m \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1$ et $\ldots$et $r, m \vdash_{\text{EXPR}} e_N \rightsquigarrow v_n$ et si $r[x_1 = v_1, \ldots, x_n = v_n], m \vdash_{\text{EXPR}} e' \rightsquigarrow v$ alors $r, m \vdash_{\text{EXPR}}$ `(`$e\ e_1 \ldots e_n$`)` $\rightsquigarrow v$.

La différence est que l'expression qui constitue le corps de la fonction est évalué dans un contexte où l'environnement est construit à partir de celui qui est présent au moment de l'évaluation de l'application; et non plus celui qui aurait été figé deans la fermeture. Avec cette manière de faire, la valeur de l'application `(f 5)` de notre programme est 47, c'est-à-dire, la valeur de $5 + 42$.

2.4 Sémantique dénotationnelle

Pour la sémantique dénotationnelle, un élément important de l'extension de *APS0* à *APS1* est l'apparition d'une nouvelle espèce de valeur: les fermetures.

Avec le langage de fonction de la sémantique dénotationnelle, on peut décrire un peu plus précisément ce qu'est une fermeture. Le second terme d'une fermeture que nous avions écrit $r + [x_1, \ldots, x_n]$ peut être explicité comme une *fonction* qui calcule l'environnement d'évaluation des applications à venir.

En effet, la fonction notée $\lambda v_1 \ldots v_n.\rho[x_1 = v_1; \ldots; x_n = v_n]$ appliquée aux valeurs $u_1, \ldots, u_n$ est l'environnement $\rho[x_1 = u_1; \ldots; x_n = u_n]$. Ce qu'il nous faut pour l'évaluation de l'application.

Avec ce point de vue, une fermeture est un couple qui encapsule une expression et une fonction des valeurs vers les environnements.

Ensembles

Union généralisée Pour tout ensemble I et toute expression ensembliste $E(i)$ où $i \in I$, on note $\bigcup_{i \in I} E(i)$ l'ensemble tel que, pour tout z, $z \in \bigcup_{i \in I} E(i)$ si et seulement si il existe $i \in I$ tel que $z \in E(i)$.

Fonction n-aires Pour tout $n \in I\!N$ on définit $X^n \to Y$ par induction sur $n \in I\!N$: $X^0 \to Y = Y$; $X^{n+1} \to Y = X \to X^n \to Y$.

On pose : $X^* \to Y = \bigcup_{i \in I\!N} X^i \to Y$.

Domaines sémantiques

- $F = \textsc{Expr} \times (V^* \to E)$ fermetures
- $V = B \cup N \cup F$ valeurs
- $E = \mathsf{ident} \to (A \oplus V)$ ensvironnements

Notez que les ensembles F, V, et E sont mutuellements définis. Toutefois, cette définition est *bien fondée* en ce sens que l'on peut l'appuyer sur l'ensemble des valeurs de base: $B \cup N$.

Posons $V_0 = B \cup N$; $E_0 = \mathsf{ident} \to (A \oplus V_0)$ et $F_0 = \textsc{Expr} \times (V_0^* \to E_0)$. Puis, par induction sur $n \in I\!N$: $V_{n+1} = V_n \cup F_n$; $E_{n+1} = \mathsf{ident} \to (A \oplus V_n)$ et $F_{n+1} = \textsc{Expr} \times (V_n^* \to E_n)$.

On pose alors $V = \bigcup_{n \in I\!N} V_n$; $E = \bigcup_{n \in I\!N} E_n$ et $F = \bigcup_{n \in I\!N} F_n$.

Fonction sémantique E

La signature de **E** ne change pas en apparence, mais la définition de V, et donc de E ont été modifiées.

On rajoute donc aux équations sémantiques de *APS0* qui définissent **E** les deux équations suivantes:

$$\mathbf{E}[[\texttt{[}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{]}\, e]]\rho\sigma = \langle e, \lambda w_1 \ldots w_n.\rho[x_1 = w_1; \ldots; x_n = w_n]\rangle$$

$$\begin{aligned}\mathbf{E}[[\texttt{(}e\ e_1 \ldots e_n\texttt{)}]]\rho\sigma = {} & \text{let } v_1 = \mathbf{E}[[e_1]]\rho\sigma \text{ in}\\ & \vdots\\ & \text{let } v_n = \mathbf{E}[[e_n]]\rho\sigma \text{ in}\\ & \text{let } \langle e', r\rangle = \mathbf{E}[[e]]\rho\sigma \text{ in}\\ & \text{let } \rho' = (r\ v_1 \ldots v_n) \text{ in}\\ & \quad \mathbf{E}[[e']]\rho'\sigma\end{aligned}$$

3 Expressions procédurales

Sur le modèle des expressions fonctionnelles, on peut abstraire des *paramètres* d'un bloc pour créer de nouvelles instructions.

Par exemple, on pourrait définir une instruction de *swap* en abstrayant les varaible `x` et `y` du bloc `[ VAR z int; SET z x; SET x y; SET y z ]` en écrivant simplement `[x:int, y:int][ VAR z int; SET z x; SET x y; SET y z ]`. On pourrait nommer ces instructions et les invoquer.

3.1 Syntaxe

Lexique on ajoute le mot clef `PROC` pour la définition d'une procédure, le mot clef `CALL` pour leur invocation et le mot réservé `void` pour le typage.

Grammaire

DEC	::=	...
	|	`PROC` trmsident `[` ARGS `]` PROG
STAT	::=	...
	|	`CALL` ident EXPRS
TYPE	::=	...
	|	`void`

3.2 Typage

Une procédure est une «fonction» dont le type de retour est void. Le typage d'une déclaration de procédure vient avec celui des suites de commandes:

(PROC) si $G; x_1 : t_1; \ldots; x_n : t_n \vdash_{\text{CMDS}} cs_1 :$ void et si $G; x : t_1\texttt{*}\ldots\texttt{*}t_n\texttt{->void} \vdash_{\text{CMDS}} cs_2 :$ void alors $G \vdash_{\text{CMDS}} (\texttt{PROC}\ x\ \texttt{[}x_1\texttt{:}t_1\texttt{,}\ldots\texttt{,}x_n : t_n\texttt{]}\ \texttt{[}cs_1\texttt{]}\texttt{;}cs_2) :$ void

Et on rajoute dans le typage des instructions:

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

3.3 Sémantique opérationnelle

Pour les procédures, on se donne une nouvelle catégorie de fermetures dont le premier éléments est un bloc (suite de commandes) et non plus une expression. On écrit $\langle cs, r + [x_1, \ldots, x_n]\rangle$.

On ajoute aux règles sémantiques des déclarations

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

La règle pour l'instruction d'appel de procédure est la suivante:

(CALL) si $r, m \vdash_{\text{EXPR}} x \rightsquigarrow \langle cs, r_p + [x_1, \ldots, x_n]\rangle$ et si $r, m \vdash_{\text{EXPR}} e_1 \rightsquigarrow v_1$, ..., et si $r, m \vdash_{\text{EXPR}} e_n \rightsquigarrow v_n$ et si $r_p[x_1 = v_1, \ldots, x_n = v_n], m \vdash_{\text{BLOCK}} \texttt{[}cs\texttt{]} \rightsquigarrow m'$ alors $r, m \vdash_{\text{STAT}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (r, m')$

3.4 Sémantique dénotationnelle

Domaines le domaine des fermetures s'est enrichi, il devient

$$F = (\text{EXPR} \oplus \text{PROG}) \times (V^* \to E)$$

Équations sémantiques il faut ajouter à la sémantique déjà établie une nouvelle équation pour définir la valeur des définitions de procédure (fonction sémantique **D**) ainsi qu'une équation pour définir celle d'un appel de procédure (fonction sémantique **S**).

La signature de ces fonctions reste formellement identique mais la définition de F, donc de V et donc de E ont été modifiées.

$$\begin{aligned}\mathbf{D}[[\texttt{PROC}\ x\ \texttt{[}x_1 : t_1, \ldots, x_2 : t_2\texttt{]}\ \texttt{[}cs\texttt{]}]]\rho\sigma \quad = \quad & \text{let } r = \lambda v_1 \ldots v_n.\rho[x_1 = inV(v_1); \ldots; x_n = inV(v_n)]\rangle \text{ in} \\ & \text{let } f = \langle in\text{PROG}(cs), r\rangle \text{ in} \\ & \quad (\rho[x = f], \sigma)\end{aligned}$$

$$
\begin{array}{lcl}
\mathbf{S}[[\texttt{CALL}\ x\ e_1 \ldots e_n]]\rho\sigma & = & \text{let } v_1 = \mathbf{E}[[e_1]]\rho\sigma \text{ in} \\
 & & \vdots \\
 & & \text{let } v_n = \mathbf{E}[[e_n]]\rho\sigma \text{ in} \\
 & & \text{let } \langle in\textsc{Prog}(cs), r\rangle = \rho(x) \text{ in} \\
 & & \text{let } \rho' = (r\ v_1 \ldots v_n) \text{ in} \\
 & & \quad \mathbf{B}[[cs]]\rho'\sigma
\end{array}
$$