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

P. Manoury

Janvier 2022

Contents

3 *APS1a*: valeur ou référence

L'extension *APS1a* de *APS1* est motivivée par deux observations concernant la discipline de types de *APS1*:

- Il existe des programmes *APS1* syntaxiquement corrects, typables mais qui provoquent une erreur d'exécution.
- Le typage de *APS1* interdit la définition de procédures susceptibles d'affecter ses arguments.

3.1 Constantes *vs* variables

L'objectif du typage est de garantir une certaine sûreté d'exécution. Par exemple, dans une application `(e e1 e2)`, le typage garantit que la valeur de `e` est une fonction et que lui sont appliqués exactement le bon nombre d'arguement. Ainsi, les conditions édictées par la règle sémantique (APP) sont satisfaites.

Toutefois, *APS1* souffre du défaut qu'il y existe des programmes syntaxiquement corrects, correctement typés mais qui provoquent une erreur d'exécution: ils ne sont pas évaluables. Par exemple:

```
[
  CONST x int 0;
  SET x 42
]
```

En effet,

1. par la règle (CONST) $\varepsilon, \varepsilon \vdash_{\text{DEF}} \texttt{CONST x int 0} \rightsquigarrow [\texttt{x} = inZ(0)]$
2. reste à évaluer `SET x 42` dans ce contexte. On applique la règle (SET), c'est la seule possible:
   - on a $[\texttt{x} = inZ(0)](\texttt{x}) = inZ(0)$
   - **mais** la règle (SET) attend un $inA(a)$.

L'évaluation échoue sur la tentave d'affectation d'une constante. C'est une bonne chose dans la mesure où, si l'on déclare une *constante*, c'est que l'on souhaite qu'elle le reste.

En revanche, on peut regretter que le typage n'ait pas su détecter cette anomalie. On peut remédier à ce défaut en enrichissant l'information de typage de manière à savoir si un identificateur est une constante ou une variable. C'est-à-dire, en indiquant si une type est assigné à un objet modifiable (variable) ou non (constante).

On s'inspire pour cela du langage Ocaml où le type des valeurs modifiables est explicite: `(int ref)`, `(bool ref)`, etc. Mais, dans *APS1a*, cette information n'est pas demandée au programmeur, elle est introduite et manipulé uniquement par les règles de typage (et, donc, le système de vérification de type).

Pour réaliser cela, on modifie les règles (VAR) est (SET):

(VAR) si $t \in \{\texttt{int}, \texttt{bool}\}$
alors $\Gamma \vdash_{\text{DEF}} (\texttt{VAR}\ x\ t) : \Gamma[x : (\texttt{ref}\ t)]$

(SET) si $\Gamma(x) = (\texttt{ref}\ t)$ et si $\Gamma \vdash_{\text{EXPR}} e : t$
alors $\Gamma \vdash_{\text{STAT}} (\texttt{SET}\ x\ e) : \texttt{void}$

Notre programme fautif n'est plus typable avec ces deux nouvelles règles.

Il faut cependant prendre garde à ce que désormais, le type d'un identificateur peut être un type «constant» ou un type «variable» (`(ref bool)`, `(ref int)`, etc.). Pour pouvoir typer correctement l'utilisation de symboles de variables dans les expressions, il faut modifier la règle (ID). Lorsqu'un symbole de variable est utilisé en «lecture», par exemple, dans l'expression associé à l'instruction d'affichage `ECHO`, il faut confondre le type `int` et le type `(ref int)`. Cela donne deux règles pour le typage des identificatuers:

(IDR) si $x \in$ `ident` et $\Gamma(x) =$ (`ref` t)
alors $\Gamma \vdash_{\text{EXPR}} x : t$

(IDV) si $x \in$ `ident`,
si $\Gamma(x) = t$ avec $t \neq$ (`ref` t')
alors $\Gamma \vdash_{\text{EXPR}} x : t$

3.2 Valeurs *vs* références

Le typage de *APS1* répond strictement à une sémantique d'appel de fonction ou de procédure *par valeur*. À ce titre, elle interdit une définition telle que

```
PROC inc [x:int] [ SET x (add x 1) ];
```

En effet, pour vérifier la coorection du typage de cette définition, la règle (PROC) demande de vérifier que [`x` : *int*] $\vdash_{\text{STAT}}$ (`SET x (add x 1)`) : *void*. Or, notre nouvelle règle (SET) fait échouer cette vérification: le type du `x` «à gauche» de l'affectation est `int` alors que la nouvelle règle (SET) réclame un `(ref int)`.

Si l'on veut autoriser la possibilité d'un *passage de paramètre par référence*, il faut modifier

1. et la syntaxe de définition des procédures;
2. et la syntaxe des appels de procédures.

Pour pouvoir modifier la valeur d'un paramètre dans une procédure, il faut, lors de l'appel de la procédure, qu'ait été passé non pas une valeur immédiate, mais une *adresse*. On ne souhaite toutefois pas modifier totalement la sémantique du passage des paramètres. On souhaite que le passage par valeur reste l'option par défaut. En conséquence

1. On introduit dans la syntaxe de déclaration des paramètres d'une procédure le mot clef `var` pour indiquer que celui-ci sera passé par référence.
2. On introduit dans la synstaxe d'appel des procédures le mot clef `adr` pour indiquer que c'est l'adresse du paramètre (qui sera nécessairement un symbole de variable) que l'on veut passer à la procédure, et non sa valeur.

Syntaxe On ajoute donc les deux mots clef `var` (en minuscule) et `adr`

Pour ce qui est de la grammaire, on modifie la syntaxe des définitions et d'appel en introduisant un non-terminal spécifique pour la déclaration des paramètres formels des procédures ainsi qu'un autre spécifique aux paramètres d'appel:

DEF	::=	...
	|	`PROC ident [` ARGSP `]` BLOCK
	|	`PROC REC ident [` ARGSP `]` BLOCK
ARGSP ::=	::=	ARGP
	|	ARGP `,` ARGSP
ARGP	::=	`ident :` TYPE
	|	`var ident :` TYPE
STAT	::=	...
	|	`CALL ident` EXPRSP
EXPRSP	::=	EXPRP
	|	EXPRP EXPRSP
EXPRP	::=	EXPR
	|	`(adr ident)`

Typage Pour autoriser l'affectation d'un paramètre, son type doit être de la forme `(ref t)`. Les paramètres susceptibles d'"être affectés devant être explicitement déclarés comme tels avec la modalité `var`, on modifie les régles de typages des défintions de procédures.

Étant donné une liste de déclaration de paramètres $[p_1 : t_1, \ldots, p_n : t_n]$ on se donne une fonction A qui en extrait la listes des identificateurs des paramètres associés à leur type marqué ou non de `ref` selon qu'ils ont été déclarés avec la modalité `var` ou non. Plus formellement: $A([p_1 : t_1, \ldots, p_n : t_n]) = [x_1 : t'_1, \ldots, x_n : t'_n]$ avec $x_i, t'_i = \begin{cases} x_i : t_i & \texttt{si } p_i = x_i \\ x_i : (\texttt{ref } t_i) & \texttt{si } p_i = \texttt{var } x_i \end{cases}$

Les régles de définitions des procédures deviennent:

(PROC) si $A([p_1 : t_1, \ldots, p_n : t_n]) = [x_1 : t'_1, \ldots, x_n : t'_n]$
si $\Gamma[x_1 : t'_1; \ldots; x_n : t'_n] \vdash_{\text{BLOCK}} bk : \texttt{void}$
alors $\Gamma \vdash_{\text{DEF}} (\texttt{PROC } x \texttt{ [}p_1\texttt{:}t_1, \ldots, p_n\texttt{:}t_n\texttt{]}bk) : \Gamma[x : t'_1 \texttt{ * } \ldots \texttt{ * } t'_n \texttt{ -> void}]$

(PROCREC) si $A([p_1 : t_1, \ldots, p_n : t_n]) = [x_1 : t'_1, \ldots, x_n : t'_n]$
si $\Gamma[x_1 : t'_1; \ldots; x_n : t'_n; x : t'_1 \texttt{ * } \ldots \texttt{ * } t'_n \texttt{ -> void}] \vdash_{\text{BLOCK}} bk : \texttt{void}$
alors $\Gamma \vdash_{\text{DEF}} (\texttt{PROC REC } x \texttt{ [}p_1\texttt{:}t_1, \ldots, p_n\texttt{:}t_n\texttt{]}bk) : \Gamma[x : t'_1 \texttt{ * } \ldots \texttt{ * } t'_n \texttt{ -> void}]$

Concernant l'appel des procédure, on modifie la règle (CALL) en introduisant une relation spécifique au typage des paramètres d'appel.

Modification de la règle (CALL):

(CALL) si $\Gamma(x) = t_1 \texttt{ * } \ldots \texttt{ * } t_n \texttt{ -> } void$
si $\Gamma \vdash_{\text{EXPAR}} e_1 : t_1, \ldots$ et si $\Gamma \vdash_{\text{EXPAR}} e_n : t_n$
alors $\Gamma \vdash_{\text{STAT}} (\texttt{CALL } x\ e_1 \ldots e_n) : \texttt{void}$

Définition de la nouvelle relation de typage: $\vdash_{\text{EXPAR}}$

(REF) si $\Gamma(x) = (\texttt{ref } t)$
alors $\Gamma \vdash_{\text{EXPAR}} \texttt{(adr } x\texttt{)} : (\texttt{ref } t)$

(VAL) si $\Gamma \vdash_{\text{EXPR}} e : t$
alors $\Gamma \vdash_{\text{EXPAR}} e : t$

Sémantique La modification des régles sémantiques des définitions de procédures consiste simplement à récupérer les noms de paramètres pour définir les fermetures procédurales. Pour ce, on se donne la fonction X telle que $X([p_1 : t_1; \ldots; p_n : t_n]) = x_1, \ldots, x_n$ avec $x_i = \begin{cases} x_i & \texttt{si } p_i = x_i \\ x_i & \texttt{si } p_i = \texttt{(var } x_i\texttt{)} \end{cases}$

On définit alors

(PROC) $\rho, \sigma \vdash_{\text{DEF}} (\texttt{PROC } x\ t \texttt{ [}p_1\texttt{:}t_1\texttt{,} \ldots p_n\texttt{:}t_n\texttt{]}\ bk) \rightsquigarrow (\rho[x = inP(bk, (x_1, \ldots, x_n), \rho)], \sigma)$

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

À l'instar de ce qui a été fait pour le typage, la sémantique des appels de procédure est définie à l'aide d'une relation sémantique spécifiques pour l'évaluation des paramètres d'appel: $\vdash_{\text{EXPAR}}$. La «valeur» d'un paramètre sera soit une adresse, soit une valeur proprement dite.

(REF) si $\rho(x) = inA(a)$
alors $\rho, \sigma \vdash_{\text{EXPAR}} (\texttt{adr } x) \rightsquigarrow inA(a)$

(VAL) si $\rho, \sigma \vdash_{\text{EXPR}} e \rightsquigarrow v$
alors $\rho, \sigma \vdash_{\text{EXPAR}} e \rightsquigarrow v$

(CALL) si $\rho(x) = inP(bk, (x_1; \ldots; x_n), \rho')$,
si $\rho, \sigma \vdash_{\text{EXPAR}} e_1 \rightsquigarrow v_1$, ..., si $\rho, \sigma \vdash_{\text{EXPAR}} e_n \rightsquigarrow v_n$
si $\rho'[x_1 = v_1; \ldots; x_n = v_n), \sigma, \omega \vdash_{\text{BLOCK}} bk \rightsquigarrow (\sigma', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (\sigma', \omega')$

(CALLR) si $\rho(x) = inPR(bk, x, (x_1; \ldots; \rho')$,
si $\rho, \sigma \vdash_{\text{EXPAR}} e_1 \rightsquigarrow v_1$, ..., si $\rho, \sigma \vdash_{\text{EXPAR}} e_n \rightsquigarrow v_n$
et si $\rho'[x_1 = v_1; \ldots; x_n = v_n][x = inPR(bk, x, (x_1; \ldots; x_n), \rho')], \sigma, \omega \vdash_{\text{BLOCK}} bk \rightsquigarrow (\sigma', \omega')$
alors $\rho, \sigma, \omega \vdash_{\text{STAT}} (\texttt{CALL}\ x\ e_1 \ldots e_n) \rightsquigarrow (\sigma', \omega')$