

UPMC  
Master informatique 2 – STL  
NI503 – CONCEPTION DE LANGAGES  
Notes IV

Décembre 2011

## 1 Rupture et continuation

### 1.1 Rupture brutale

Il est facile d'arrêter complètement un programme, avec une instruction `EXIT`, par exemple.

```
CMD ::= ...
      | EXIT
C[[EXIT]] $\rho$   $\mu$  =  $\mu$ 
```

Il est moins immédiat d'obtenir un effet de rupture localisé, comme, par exemple, le `break` qui interrompt l'itération d'une boucle mais reprend le calcul après celle-ci. Pour cela, il faut pouvoir maîtriser le point de reprise, la *continuation* du calcul ; c'est-à-dire l'endroit (adresse) de reprise du calcul, ainsi qu'éventuellement, un contexte associé.

### Continuations

Représentation fonctionnelle de la pile d'évaluation.

**Accumulateur et récursion terminale** on sait que l'on peut donner une version récursive terminale de la factorielle.

```
FUNREC fact n r =
  (if (n=0) r (fact (n-1) (n*r)))
;
FUN fac = = (fact n 1)
;
```

Mais cette transformation n'est pas toujours possible, par exemple, en imaginant que nous disposions des constructeurs de listes `CONS` et `NIL`, la fonction

```
FUNREC nlist n =
  (if (n=0) NIL (CONS n (nlist (n-1))))
```

construit la liste des entiers de `n` à 1.

Avec un accumulateur, on serait tenté de donner la version suivante

```

FUNREC nlistt n r =
  (if (n=0) r (nlistt (n-1) (CONS n r)))
;
FUN nlist n = (nlistt n NIL)
;

```

mais ça n'est pas la même fonction. Cette dernière construit la liste des entiers de 1 à n.

**Continuation : accumulateur fonctionnel** L'idée est de «retarder» l'application de CONS en utilisant comme accumulateur une *fonction de continuation* plutôt qu'une valeur immédiate. Il faut naturellement que nous disposions pour cela d'une possibilité d'écrire des *expressions fonctionnelles*.

```

FUNREC nlistk n k =
  (if (n=0)
    (k NIL)
    (nlistk (n-1) (lambda (r) (k (CONS n r)))))
;
FUN nlist n = (nlistk n (lambda (r) r))
;

```

**Rupture** Autre exemple : la multiplication des éléments d'une liste d'entiers – on suppose que l'on a les accesseurs CAR et CDR. L'idée est d'*oublier* la continuation courante lorsque l'on tombe sur un entier nul dans la liste : le résultat global est 0.

```

FUNREC mlist ns k =
  (if (ns=NIL)
    (k 1)
    (if ((CAR ns) = 0)
      0
      (mlist (CDR ns) (lambda (r) (k ((CAR ns) * r))))))
;

```

## 1.2 Sémantique à continuation : instructions

Utiliser les continuations au niveau des fonctions sémantiques, ici, les fonctions  $\mathbf{Cs}[\ ]$  et  $\mathbf{C}[\ ]$ , et montrer comment elles permettent de spécifier les constructions de contrôle de flot des langages de programmation.

**Domaines sémantique** Le domaine des continuations, au niveau des commandes

$$Cont = Mem \rightarrow Mem$$

$$K_0 = \lambda m.m$$

Les procédures, sont des commandes, leur domaine est maintenant :

$$\begin{aligned}
FProc_0 &= Mem \rightarrow Cont \rightarrow Mem \\
FProc_{n+1} &= IR \rightarrow FProc_n \\
FProc &= \{FProc_i \mid i \in \mathbb{N}\}
\end{aligned}$$

**Signatures** des fonctions sémantiques avec continuation :

- P** :  $Prog \rightarrow Mem$
- Cs** :  $Cmds \rightarrow Env \rightarrow Mem \rightarrow Cont \rightarrow Mem$
- C** :  $Cmd \rightarrow Env \rightarrow Mem \rightarrow Cont \rightarrow Mem$
- D** :  $Dec \rightarrow Env \rightarrow Mem \rightarrow Env \times Mem$
- E** :  $Expr \rightarrow Env \rightarrow Mem \rightarrow \mathbb{R}$

### Équations

$$\begin{aligned}
\mathbf{P}[[[cs]]] &= \mathbf{Cs}[[cs]]\emptyset M_0 K_0 \\
\mathbf{Cs}[[ ]]\rho \mu \kappa &= (\kappa \mu) \\
\mathbf{Cs}[[d; cs]]\rho \mu \kappa &= \mathbf{Cs}[[cs]]\rho' \mu' \kappa \\
&\quad \text{avec } \rho', \mu' = \mathbf{D}[[d]]\rho \mu \\
\mathbf{Cs}[[c; cs]]\rho \mu \kappa &= \mathbf{C}[[c]]\rho \mu (\lambda \mu'. \mathbf{Cs}[[cs]]\rho \mu' \kappa) \\
\mathbf{D}[[\text{PROC } f \ x = p]]\rho \mu &= \rho', \mu \\
&\quad \text{avec } \rho' = \rho[f := \text{inP}(\lambda v \lambda \mu \lambda \kappa. (\mathbf{P}[[p]])(\rho[x := \text{inR}(v)] \mu \kappa))] \\
\mathbf{D}[[\text{PROCREC } f \ x = p]]\rho \mu &= \rho', \mu \\
&\quad \text{avec } \rho' = \rho[f := \text{inP}(!w. \lambda v \lambda \mu \lambda \kappa. (\mathbf{P}[[p]]\text{es}(\rho[x := \text{inR}(v); f := \text{inP}(w)] \mu \kappa))] \\
\mathbf{D}[[\text{FUN } f \ x = e]]\rho \mu &= \rho', \mu \\
&\quad \text{avec } \rho' = \rho[f := \text{inF}(\lambda v \lambda \mu. (\mathbf{E}[[e]])(\rho[x := \text{inR}(v)] \mu))] \\
\mathbf{D}[[\text{FUNREC } f \ x = e]]\rho \mu &= \rho', \mu \\
&\quad \text{avec } \rho' = \rho[f := \text{inF}(!w. \lambda v \lambda \mu. (\mathbf{E}[[e]])(\rho[x := \text{inR}(v); f := \text{inF}(w)] \mu))] \\
\\
\mathbf{D}[[\text{VAR } x]]\rho \mu &= \rho', \mu' \\
&\quad \text{avec } \rho' = \rho[x := a] \text{ et } a, \mu' = (\text{newM } \mu) \\
\mathbf{C}[[\text{MOVE } e]]\rho \mu \kappa &= (\kappa (\text{setM } (\text{setM } \mu \ a_2 \ x + k \sin(\alpha)) \ a_3 \ y + k \cos(\alpha))) \\
&\quad \text{avec } \alpha = (\text{getM } \mu \ a_1), x = (\text{getM } \mu \ a_2), y = (\text{getM } \mu \ a_3) \\
&\quad \text{et } k = \mathbf{E}[[e]]\rho \mu \\
\mathbf{C}[[\text{TURN } e]]\rho \mu \kappa &= (\kappa (\text{setM } \mu \ a_1 \ (\alpha + d \frac{\pi}{180}))) \\
&\quad \text{avec } \alpha = (\text{getM } \mu \ a_1) \text{ et } d = \mathbf{E}[[e]]\rho \mu \\
\mathbf{C}[[\text{CALL } f \ e]]\rho \mu \kappa &= \text{case } (\rho \ f) : \\
&\quad \text{inP}(p) \rightarrow (p (\mathbf{E}[[e]]\rho \mu) \mu \kappa) \\
&\quad | \_ \rightarrow \perp \\
\\
\mathbf{C}[[x := e]]\rho \mu \kappa &= \text{case } (\rho \ x) : \\
&\quad \text{inA}(a) \rightarrow (\kappa (\text{setM } \mu \ a \ (\mathbf{E}[[e]]\rho \mu))) \\
&\quad | \_ \rightarrow \perp \\
\mathbf{C}[[\text{IF } e \ cs1 \ cs2]]\rho \mu \kappa &= \text{case } (\mathbf{E}[[e]]\rho \mu) : \\
&\quad 0 \rightarrow \mathbf{Cs}[[cs2]]\rho \mu \kappa \\
&\quad | v \rightarrow \mathbf{Cs}[[cs1]]\rho \mu \kappa \\
&\quad | \_ \rightarrow \perp \\
\mathbf{C}[[\text{WHILE } e \ cs]]\rho \mu \kappa &= (!k. \lambda m. \\
&\quad \text{case } (\mathbf{E}[[e]]\rho \mu) : \\
&\quad 0 \rightarrow (\kappa \ m) \\
&\quad | v \rightarrow \mathbf{Cs}[[cs]]\rho \ m \ k \\
&\quad | \_ \rightarrow \perp \\
&\mu)
\end{aligned}$$

$$\begin{aligned}
\mathbf{E}[[x]]\rho \mu &= \text{case } (\rho x) : \\
&\quad \text{inR}(v) \rightarrow v \\
&\quad | \text{inA}(a) \rightarrow (\text{getM } \mu a) \\
&\quad | \_ \rightarrow \perp \\
\mathbf{E}[[e_1+e_2]]\rho \mu &= \text{case } (\mathbf{E}[[e_1]]\rho \mu, \mathbf{E}[[e_2]]\rho \mu) : \\
&\quad \text{inR}(v_1), \text{inR}(v_2) \rightarrow v_1 + v_2 \\
&\quad | \_ \rightarrow \perp \\
\cdots & \\
\mathbf{E}[[f e]]\rho \mu &= \text{case } (\rho f) : \\
&\quad \text{inF}(t) \rightarrow (t (\mathbf{E}[[e]]\rho \mu)) \\
&\quad | \_ \rightarrow \perp \\
\mathbf{E}[[\text{if } e e_1 e_2]]\rho \mu &= \text{case } (\mathbf{E}[[e]]\rho \mu) : \\
&\quad 0 \rightarrow \mathbf{E}[[e_2]]\rho \mu \\
&\quad | v \rightarrow \mathbf{E}[[e_1]]\rho \mu \\
&\quad | \_ \rightarrow \perp
\end{aligned}$$

## La «continuation courante»

C'est une structure de contrôle apportée par le langage Scheme. On note aussi `call/cc`. Intuitivement, en Scheme, on utilise cette structure en écrivant l'application (`call/cc (lambda (k) e)`). L'évaluation de cette application particulière est l'évaluation de l'expression `e` où `k` a pour valeur la «continuation courante». En Scheme, on invoque la continuation sous forme de l'application de son nom à une expression : `(k e')`

Avec notre sémantique à continuations, il est facile de connaître la continuation courante d'une instruction. On peut donc facilement réaliser un `CALL/CC` au niveau d'une instruction ou d'un bloc d'instructions. On invoquera la continuation avec la nouvelle instruction `THROW`.

```

CMD ::= ...
      | CALL/CC ident [ CMDS ]
      | THROW ident

```

Le premier argument (`ident`) de `CALL/CC` sera utilisé comme le nom local d'un *exit* : `CALL/CC` à un effet de *lieur* sur ce nom. Pour lier la continuation courante à ce nom, il faut la *réifier*, c'est-à-dire, en faire une valeur. Sémantiquement, cela est très facile :

$$Val = \dots \oplus Cont$$

$$\begin{aligned}
\mathbf{C}[[\text{CALL/CC } k \text{ cs}]]\rho \mu \kappa &= \mathbf{Cs}[[cs]]\rho' \mu \kappa \\
&\quad \text{avec } \rho' = \rho[k := \text{inC}(\kappa)] \\
\mathbf{C}[[\text{THROW } k]]\rho \mu \kappa &= \text{case } (\rho k) : \\
&\quad | \text{inC}(\kappa') \rightarrow (\kappa' \mu) \\
&\quad | \_ \rightarrow \perp
\end{aligned}$$

**Faire un break** Pour simuler un `break`, on choisit un identificateur particulier, par exemple `*break*` (il ne fait pas partie du langage).

On peut alors *macro-générer* l'instruction `BREAK` par :

$$\mathbf{C}[[\text{BREAK}]]\rho \mu \kappa = \mathbf{C}[[\text{THROW } *break*]]\rho \mu \kappa$$

On peut alors également macro-générer la possibilité d'un `break` à l'intérieur d'une boucle avec `CALL/CC` :

$$\mathbf{C}[[\text{WHILE } e \text{ cs}]]\rho \mu \kappa = \mathbf{C}[[\text{CALL/CC } *break* \text{ [ WHILE } e \text{ cs } ]]]\rho \mu \kappa$$

Ainsi, la continuation courante du `WHILE` est capturée et associée à l'identificateur `*break*`. On a, en quelque sorte marqué le point de sortie du `WHILE` pour pouvoir y «sauter» si un `BREAK` est exécuté.

On peut procéder de même avec toute instruction provoquant l'activation d'un bloc : `IF`, `CALL`, etc.

**Exception simple** Un mécanisme tel que le `try-catch` peut être réalisé avec le `CALL/CC` comme une «macro» :

$$\mathbf{C}[[\text{TRY/CATCH } cs_1 \ x \ cs_2]]\rho \ \mu \ \kappa \ = \ \mathbf{C}[[\text{CALL/CC } x \ cs_1]]\rho \ \mu \ \kappa' \\ \text{avec } \kappa' = \lambda m. \mathbf{Cs}[[cs_2]]\rho \ m \ \kappa$$

**Remarque** cette utilisation simple des continuations est en fait l'équivalent d'un `GOTO` *dynamique* : l'étiquette de saut est posée au moment de l'exécution du `CALL/CC`.