UPMC
Master informatique 2 – STL
NI503 – Conception de langages
Notes V

2012

1 Rupture et continuation

1.1 Rupture brutale

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

$$\begin{array}{lcl} \textsc{Cmd} & ::= & \ldots \\ & | & \texttt{EXIT} \end{array}$$

$$\mathbf{C}[[\texttt{EXIT}]]\rho\,\mu \;=\; \mu$$

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 disposition des constructiers 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 et d'*oublier* la continuation courrante 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 **Cs**[[]] et **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

$$Cont = Mem \to Mem$$

$$K_0 = \lambda m.m$$

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

$$\begin{array}{lcl} FProc_0 & = & Mem \to Cont \to Mem \\ FProc_{n+1} & = & I\!R \to FProc_n \\ FProc & = & \{FProc_i \mid i \in I\!N\} \end{array}$$

Signatures des fonctions sémantiques avec continuaton :

$\mathbf{P}$: $Prog \to Mem$
$\mathbf{Cs}$: $Cmds \to Env \to Mem \to Cont \to Mem$
$\mathbf{C}$: $Cmd \to Env \to Mem \to Cont \to Mem$
$\mathbf{D}$: $Dec \to Env \to Mem \to Env \times Mem$
$\mathbf{E}$: $Expr \to Env \to Mem \to I\!R$

Équations

$$\mathbf{P}[[\texttt{[}cs\texttt{]}]] = \mathbf{Cs}[[cs]]\emptyset\ M_0\ K_0$$

$$\mathbf{Cs}[[\ ]]\rho\ \mu\ \kappa = (\kappa\ \mu)$$

$$\mathbf{Cs}[[d\texttt{;}cs]]\rho\ \mu\ \kappa = \mathbf{Cs}[[cs]]\rho'\ \mu'\ \kappa \quad \text{avec } \rho', \mu' = \mathbf{D}[[d]]\rho\ \mu$$

$$\mathbf{Cs}[[c\texttt{;}cs]]\rho\ \mu\ \kappa = \mathbf{C}[[c]]\rho\ \mu\ (\lambda\mu'.\mathbf{Cs}[[cs]]\rho\ \mu'\ \kappa)$$

$$\mathbf{D}[[\texttt{PROC}\ f\ x\ \texttt{=}\ p]]\rho\ \mu = \rho', \mu \quad \text{avec } \rho' = \rho[f := inP(\lambda v\lambda\mu\lambda\kappa.(\mathbf{P}[[p]](\rho[x := inR(v)])\ \mu\ \kappa))]$$

$$\mathbf{D}[[\texttt{PROCREC}\ f\ x\ \texttt{=}\ p]]\rho\ \mu = \rho', \mu \quad \text{avec } \rho' = \rho[f := inP(!w.\lambda v\lambda\mu\lambda\kappa.(\mathbf{P}[[p]]es(\rho[x := inR(v); f := inP(w)])\mu\ \kappa))]$$

$$\mathbf{D}[[\texttt{FUN}\ f\ x\ \texttt{=}\ e]]\rho\ \mu = \rho', \mu \quad \text{avec } \rho' = \rho[f := inF(\lambda v\lambda\mu.(\mathbf{E}[[e]](\rho[x := inR(v)])\ \mu))]$$

$$\mathbf{D}[[\texttt{FUNREC}\ f\ x\ \texttt{=}\ e]]\rho\ \mu = \rho', \mu \quad \text{avec } \rho' = \rho[f := inF(!w.\lambda v\lambda\mu.(\mathbf{E}[[e]](\rho[x := inR(v); f := inF(w)])\ \mu))]$$

$$\mathbf{D}[[\texttt{VAR}\ x]]\rho\ \mu = \rho', \mu' \quad \text{avec } \rho' = \rho[x := a] \text{ et } a, \mu' = (newM\ \mu)$$

$$\mathbf{C}[[\texttt{MOVE}\ e]]\rho\ \mu\ \kappa = (\kappa\ (setM\ (setM\ \mu\ a_2\ x + k\cos(\alpha))\ a_3\ y + k\sin(\alpha)))$$
avec $\alpha = (getM\ \mu\ a_1), x = (getM\ \mu\ a_2), y = (getM\ \mu\ a_3)$
et $k = \mathbf{E}[[e]]\rho\ \mu$

$$\mathbf{C}[[\texttt{TURN}\ e]]\rho\ \mu\ \kappa = (\kappa\ (setM\ \mu\ a_1\ (\alpha + d\tfrac{\pi}{180})))$$
avec $\alpha = (getM\ \mu\ a_1)$ et $d = \mathbf{E}[[e]]\rho\ \mu$

$$\mathbf{C}[[\texttt{CALL}\ f\ e]]\rho\ \mu\ \kappa = \mathsf{case}\ (\rho\ f):\ inP(p) \to (p\ (\mathbf{E}[[e]]\rho\ \mu)\ \mu\ \kappa)\ |\ \_ \to \bot$$

$$\mathbf{C}[[x\ \texttt{:=}\ e]]\rho\ \mu\ \kappa = \mathsf{case}\ (\rho\ x):\ inA(a) \to (\kappa\ (setM\ \mu\ a\ (\mathbf{E}[[e]]\rho\ \mu)))\ |\ \_ \to \bot$$

$$\mathbf{C}[[\texttt{IF}\ e\ cs1\ cs2]]\rho\ \mu\ \kappa = \mathsf{case}\ (\mathbf{E}[[e]]\rho\ \mu):\ 0 \to \mathbf{Cs}[[cs2]]\rho\ \mu\ \kappa\ |\ v \to \mathbf{Cs}[[cs1]]\rho\ \mu\ \kappa\ |\ \_ \to \bot$$

$$\mathbf{C}[[\texttt{WHILE}\ e\ cs]]\rho\ \mu\ \kappa = (!k.\lambda m.\ \mathsf{case}\ (\mathbf{E}[[e]]\rho\ \mu):\ 0 \to (\kappa\ m)\ |\ v \to \mathbf{Cs}[[cs]]\rho\ m\ k\ |\ \_ \to \bot\ \mu)$$

$$
\begin{array}{lcl}
\mathbf{E}[[x]]\rho\ \mu & = & \mathsf{case}\ (\rho\ x): \\
 & & \quad inR(v) \to v \\
 & & \mid inA(a) \to (getM\ \mu\ a) \\
 & & \mid \_ \to \bot \\
\mathbf{E}[[e_1\texttt{+}e_2]]\rho\ \mu & = & \mathsf{case}\ (\mathbf{E}[[e_1]]\rho\ \mu, \mathbf{E}[[e_2]]\rho\ \mu): \\
 & & \quad inR(v1), inR(v2) \to v_1 + v_2 \\
 & & \mid \_ \to \bot \\
\ldots & & \\
\mathbf{E}[[(f\ e)]]\rho\ \mu & = & \mathsf{case}\ (\rho\ f): \\
 & & \quad inF(t) \to (t\ (\mathbf{E}[[e]]\rho\ \mu)) \\
 & & \mid \_ \to \bot \\
\mathbf{E}[[(\texttt{if}\ \ e\ e_1\ e_2)]]\rho\ \mu & = & \mathsf{case}\ (\mathbf{E}[[e]]\rho\ \mu): \\
 & & \quad 0 \to \mathbf{E}[[e_2]]\rho\ \mu \\
 & & \mid v \to \mathbf{E}[[e_1]]\rho\ \mu \\
 & & \mid \_ \to \bot
\end{array}
$$

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`.

$$
\begin{array}{lcl}
\textsc{Cmd} & ::= & \ldots \\
 & \mid & \texttt{CALL/CC}\ \mathsf{ident}\ \texttt{[}\ \textsc{Cmds}\ \texttt{]} \\
 & \mid & \texttt{THROW}\ \mathsf{ident}
\end{array}
$$

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 :

$$Val = \ldots \oplus Cont$$

$$
\begin{array}{lcl}
\mathbf{C}[[\texttt{CALL/CC}\ \ k\ cs]]\rho\ \mu\ \kappa & = & \mathbf{Cs}[[cs]]\rho'\ \mu\ \kappa \\
 & & \quad \text{avec } \rho' = \rho[k := inC(\kappa)] \\
\mathbf{C}[[\texttt{THROW}\ \ k]]\rho\ \mu\ \kappa & = & \mathsf{case}\ (\rho\ k): \\
 & & \mid inC(\kappa') \to (\kappa'\ \mu) \\
 & & \mid \_ \to \bot
\end{array}
$$

Faire un `break` Pour siluker 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 :

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

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

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

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 instrution provoquant l'activation d'un bloc : `IF`, `CALL`, etc.

Exception simple Un mécanisme tel que le `try-catch` peu être réalisé de la manière suivante :

$$\mathbf{C}[[\texttt{TRY/CATCH}\ cs_1\ x\ cs_2]]\rho\ \mu\ \kappa \quad = \quad \mathbf{C}[[cs_1]]\rho'\ \mu\ \kappa \quad \text{avec } \rho' = \rho[x \mapsto inC(\lambda m.\mathbf{Cs}[[cs_2]]\rho\ m\ \kappa)]$$

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

1.3 Mélange expressions/instructions

Il peut être intéressant que l'invocation d'une continuation puisse transporter une valeur, comme le font, par exemple les exceptions. Il n'est pas possible de réaliser cela dans notre sémantique où nous avons strictement séparés les deux mondes.

Les ≪vrais≫ langages supportent en général plus ou moins de mélange. Par exemple l'affectation

en C est aussi une expression : `x = (y = 42)` est licite. Son évaluation affecte 42 à `y` puis à `x`, étant entendu que les deux variable ont été déclarées avec un type compatible ;

en Scheme on peut également écrire que quelque chose comme `(set! x (set! y 42))`, mais ici, seul `y` a la valeur 42, `x` a une valeur indétermninée ;

en OCaml l'affectation a une valeur, mais une valeur particulière : l'unique valeur `()` du type `unit`. L'affectation `x := (y := 42)` n'est acceptable que si `x` a le bon type : `unit ref` (et `y: int ref`).

En C, l'affectation est considéré comme une *expression* dont la valeur est celle de son membre droit ; d'où le néfaste `if (x=0) ...`. Ceci n'est plus vrai en JAVA. En Scheme ou ML, il n'y a pas de différence de catégorie grammaticale entre instructions et expressions : tout est expression (la distinction est faite entre expressions et déclarations) et a donc une valeur.

En C, en Scheme ou en ML, il n'y a pas de différence entre procédure et fonction. Le mélange le plus universellement admit est donc celui qui consiste à définir une fonction en utilisant un calcul procédural, impératif. Lorsque le langage n'est pas, par défaut un langage d'expression, il faut fournir une *instruction* particulière pour signaler une valeur comme valeur de retour d'une fonction écrite par des moyens procéduraux : `return`.