UPMC

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

Basile STARYNKEVITCH, http://starynkevitch.net/Basile/ *
travaillant au CEA, LIST sur **gcc-melt.org**
reprenant le cours de
Pascal MANOURY, http://www.pps.univ-paris-diderot.fr/~eleph/

2013-2014

courriel : **basile@starynkevitch.net** *et* **basile.starynkevitch@cea.fr**
forum : conception-langage-upmc2013@googlegroups.com

1 les continuations, notamment en *Scheme*

Rappel : *Scheme* est [une famille de] langage[s] de programmation, d'apparence Lisp-ienne, à typage dynamique -avec des valeurs fonctionnelles, donc des fermetures [1] Il est défini par le tout récent (juillet-novembre 2013) smallR^7RS (Revised 7th Report on Scheme) qui a été précédé notamment par R^6RS et R^5RS. Il y a un "small R^7RS", un "large R^7RS", et les $SRFI$ (Scheme Requests For Implementation)...

1.1 continuations

Une **continuation** représente *la suite* (dynamique, c.à.d *durant l'exécution*) ou *le futur du calcul* (à un certain instant précis). Comme la plupart des implantations de Scheme ont un *top-level read-eval-print-loop* (REPL), un calcul en Scheme a [presque] toujours une continuation non triviale, puisqu'il va afficher le résultat de l'expression lue en cours d'évaluation puis lire l'expression suivante. Une continuation est donc une représentation abstraite de l'état de contrôle d'un programme en cours d'exécution (techniquement, d'un processus). Pratiquement, la continuation "contient" le compteur ordinal, la pile, et en principe [2] le tas !

*Toutes les notes de cours sont disponible sur mon site web.

1. D'ailleurs, *Scheme* ressemble -malgré une syntaxe très différente- plus qu'on ne le croit à des langages comme JavaScript ou même Python.

2. Pratiquement, quand la continuation est réifiée par `call/cc` en une fermeture, on n'a pas besoin de dupliquer le tas -car il est commun à cette fermeture est à l'appelant de `call/cc`-, mais il faut "copier" la pile et le compteur ordinal.

La “procédure” **`call/cc`** (ou `call-with-current-continuation`) est toujours appelée avec un seul argument [3] qui doit être une valeur fonctionnelle ϕ [4] ou fermeture. Cette fonction ϕ reçoit comme seul argument la *continuation courante*, qui est réifiée comme une fonction κ. Ainsi on codera souvent

$$\overbrace{\texttt{(call/cc }\underbrace{\texttt{(lambda (k) ...)}}_{\phi}\texttt{)}}^{\epsilon}$$

où `k` sera liée à la continuation courante κ, donc “à la *suite du calcul* de l'expression ϵ toute entière”. Si durant l'application de ϕ à κ, c'est à dire dans le corps de ϕ la variable `k` (valant κ), on applique cette continuation κ à une valeur ν, l'exécution revient *directement* à ϵ qui renvoie ainsi ν. Pour citer R^7RS (avec mes notations) :

> The procedure `call/cc` packages the current continuation (see the rationale below) as an “escape procedure” κ and passes it as an argument to ϕ. The escape procedure is a Scheme procedure that, if it is later called, will abandon whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created.
>
> The escape procedure κ that is passed to ϕ has unlimited extent just like any other procedure in Scheme. It can be stored in variables or data structures and can be called as many times as desired. However, [...] it never returns to its caller.
>
> *Rationale :*
>
> A common use of `call/cc` is for structured, non-local exits from loops or procedure bodies, but in fact `call/cc` is useful for implementing a wide variety of advanced control structures. [...]
>
> Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at the REPL, for example, then the continuation might take the result, print it on the screen, prompt for the next input, evaluate it, and so on forever. [...] Normally these ubiquitous continuations are hidden behind the scenes and programmers do not think much about them. On rare occasions, however, a programmer needs to deal with continuations explicitly. The `call/cc` procedure allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Notez que la continuation fournie par `call/cc` pourrait être appellée plusieurs fois !

1.2 exemples avec `call/cc`

Exemples [5] :

Voir aussi la wikipage *call-with-current-continuation*

Exemple 1 :

3. L'argument passé à `call/cc` peut bien évidemment être syntaxiquement n'importe quelle expression, qui est évaluée à l'exécution -au moment où `call/cc` est appliqué- en une valeur qui doit être fonctionnelle.
4. Ainsi `(call/cc 1)` est erronné.
5. Avec `drracket` 5.3.4, langage “Assez gros Scheme”

```
(begin
  (printf "avant~n")
  (let ( (x (call/cc
              (lambda (k)
                 (printf "foo~n")
                 (k 33)
                 (printf "bide~n"))))
       )
    (printf "x=~a~n" x))
  (printf "apres~n"))
```

$\Rightarrow$

l_1, l_2 : `avant`
l_3, l_4, l_5 : `foo`,
l_6, l_8, l_9 : `x` $= 33$,
l_{10} : `apres`

Comprenez pourquoi l_7 n'est *pas* évalué donc `bide` n'est jamais affiché !

Les deux exemples suivants viennent de
`http://community.schemewiki.org/?call-with-current-continuation`

Exemple 2 :

```
    ;;; Return the first element in LST for which WANTED?
    ;;; returns a true value.
(define (search wanted? lst)
  (call/cc
    (lambda (return)
      (for-each (lambda (element)
                  (if (wanted? element)
                      (return element)))
                lst)
                #f)))
```

On peut appeler plusieurs fois la continuation courante :

Exemple 3 :

```
(define return #f)
(+ 1 (call/cc
        (lambda (cont)
          (set! return cont)
    1)))
(return 22)
(return 33)
```

qui affiche après l_5 : 2, l_6 : 23, l_7 : 34

1.3 réalisation dans *Guile 2*

GNU Guile, dans sa version actuelle, a sa propre machine virtuelle applicative à codes-octets (bytecode). La VM de Guile est bien documentée.

```
scheme@(guile-user)> (define (foo x) (+ 1 (call/cc (lambda (k) (list (k x) 'a)))))
scheme@(guile-user)> ,x foo          ;; on désassemble foo
Disassembly of #<procedure foo (x)>:

   0    (assert-nargs-ee/locals 1)      ;; 1 arg, 0 locals
   2    (object-ref 1)                  ;; #<procedure 19ed100 at <current input>:1:0 (k)>
   4    (local-ref 0)                   ;; `x'
   6    (make-closure 0 1)
   9    (call/cc)
  10    (add1)
  11    (return)

----------------------------------------
Disassembly of #<procedure 19ed100 at <current input>:1:0 (k)>:

   0    (assert-nargs-ee/locals 1)      ;; 1 arg, 0 locals
   2    (new-frame)
   3    (local-ref 0)                   ;; `k'
   5    (free-ref 0)                    ;; (closure variable)
   7    (call 1)
   9    (object-ref 1)                  ;; a
  11    (list 0 2)                      ;; 2 elements
  14    (return)
```

Quelques détails sur le codage de la machine virtuelle :
Extrait du fichier `guile-2.0/libguile/vm.c`

```
SCM
scm_i_vm_capture_stack (SCM *stack_base, SCM *fp, SCM *sp, scm_t_uint8 *ra,
                        scm_t_uint8 *mvra, scm_t_uint32 flags)
{
  struct scm_vm_cont *p;

  p = scm_gc_malloc (sizeof (*p), "capture_vm_cont");
  p->stack_size = sp - stack_base + 1;
  p->stack_base = scm_gc_malloc (p->stack_size * sizeof (SCM),
                                 "capture_vm_cont");
  p->ra = ra;
  p->mvra = mvra;
  p->sp = sp;
  p->fp = fp;
  memcpy (p->stack_base, stack_base, (sp + 1 - stack_base) * sizeof (SCM));
  p->reloc = p->stack_base - stack_base;
  p->flags = flags;
  return scm_cell (scm_tc7_vm_cont, (scm_t_bits)p);
}
static void
vm_return_to_continuation (SCM vm, SCM cont, size_t n, SCM *argv)
{
  struct scm_vm *vp;
  struct scm_vm_cont *cp;
  SCM *argv_copy;

  argv_copy = alloca (n * sizeof(SCM));
  memcpy (argv_copy, argv, n * sizeof(SCM));
```

```
29
30    vp = SCM_VM_DATA (vm);
31    cp = SCM_VM_CONT_DATA (cont);
32
33    if (n == 0 && !cp->mvra)
34      scm_misc_error (NULL, "Too few values returned to continuation",
35                      SCM_EOL);
36
37    if (vp->stack_size < cp->stack_size + n + 1)
38      scm_misc_error ("vm-engine", "not enough space to reinstate continuation",
39                      scm_list_2 (vm, cont));
40    vp->sp = cp->sp;
41    vp->fp = cp->fp;
42    memcpy (vp->stack_base, cp->stack_base, cp->stack_size * sizeof (SCM));
43    if (n == 1 || !cp->mvra)
44      {
45        vp->ip = cp->ra;
46        vp->sp++;
47        *vp->sp = argv_copy[0];
48      }
49    else
50      {
51        size_t i;
52        for (i = 0; i < n; i++)
53          {
54            vp->sp++;
55            *vp->sp = argv_copy[i];
56          }
57        vp->sp++;
58        *vp->sp = scm_from_size_t (n);
59        vp->ip = cp->mvra;
60      }
61  }
62
63  SCM
64  scm_i_vm_capture_continuation (SCM vm)
65  {
66    struct scm_vm *vp = SCM_VM_DATA (vm);
67    return scm_i_vm_capture_stack (vp->stack_base, vp->fp, vp->sp, vp->ip, NULL, 0);
68  }
```

Extrait du fichier `guile-2.0/libguile/vm-i-system.c`

```
1   VM_DEFINE_INSTRUCTION (65, call_cc, "call/cc", 0, 1, 1)
2   {
3     int first;
4     SCM proc, vm_cont, cont;
5     POP (proc);
6     SYNC_ALL ();
7     vm_cont = scm_i_vm_capture_stack (vp->stack_base, fp, sp, ip, NULL, 0);
8     cont = scm_i_make_continuation (&first, vm, vm_cont);
9     if (first)
10      {
11        PUSH (SCM_PACK (0)); /* dynamic link */
12        PUSH (SCM_PACK (0));  /* mvra */
```

```
13        PUSH (SCM_PACK (0));  /* ra */
14        PUSH (proc);
15        PUSH (cont);
16        nargs = 1;
17        goto vm_call;
18      }
19    else
20      {
21        /* Otherwise, the vm continuation was reinstated, and
22           vm_return_to_continuation pushed on one value. We know only one
23           value was returned because we are in value context -- the
24           previous block jumped to vm_call, not vm_mv_call, after all.
25
26           So, pull our regs back down from the vp, and march on to the
27           next instruction. */
28        CACHE_REGISTER ();
29        program = SCM_FRAME_PROGRAM (fp);
30        CACHE_PROGRAM ();
31        RESTORE_CONTINUATION_HOOK ();
32        NEXT;
33      }
34  }
```

Remarquez bien que la pile de la VM guile n'est *pas* la pile du processeur (celle du code C). Voir aussi sous LINUX `setcontext(3)`, `swapcontext(3)`, `sigaltstack(2)` et `setjmp(3)`

2 Transformation CPS (continuation passing style)

Chaque fonction ϕ est transformée en une fonction $\phi^{\&}$ qui prend comme (dernier) argument explicite la continuation `k` de l'appel de la fonction ϕ ; de même les opérations primitives (par exemple $\star$) ont leur forme CPS $\star$`&` qui prend comme argument la continuation donc *ne renvoie pas de valeur* (car celle-ci est donnée à la continuation ! ! !), ainsi :

```
(define (carre x)
   (* x x))
```

$\xrightarrow{cps}$

```
(define (carre& x k)
   (*& x x k))
```

de même (tiré de la wikipage)

```
(define (pyth x y)
  (sqrt (+ (* x x)
           (* y y))))
```

$\xrightarrow{cps}$

```
(define (pyth& x y k)
 (*& x x (lambda (x2)
    (*& y y
    (lambda (y2)
      (+& x2 y2 (lambda (x2py2)
                  (sqrt& x2py2 k)))))))))
```

Les conditionnelles (ou autre structure de contrôle locale) ne sont pas CPS-transformées (c.à.d il n'y a pas de `if`$^{\&}$)

```
(define (factorial n)
 (if (= n 0)
     1
     (* n (factorial (- n 1)))))
```

$\xrightarrow{cps}$

```
(define (factorial& n k)
    (=& n 0
       (lambda (b)
          (if b          ; growing continuation
              (k 1)      ; in the recursive call
              (-& n 1
                 (lambda
                   (nm1)
                   (factorial& nm1
                      (lambda (f)
                         (*& n f k)))))))))
```

Remarque : tous les appels dans la forme CPS sont terminaux-récursifs ; on a “remplacé” la pile d’appel par des fermetures emboitées.

On a trivialement la transformée-CPS de `call/cc`

```
(define call/cc& (c k) (c (lambda (v k') (k v)) k))
```

2.1 Transformation CPS hybride

Tiré de `http://matt.might.net/articles/cps-conversion/` Trois transformations :

— $\mathtt{T}-\mathtt{c} : expr \times aexp \to cexp$

— $\mathtt{T}-\mathtt{k} : expr \times (aexp \to cexp) \to cexp$

— $\mathtt{M} : expr \to aexp$

où $expr$ est le language source, $cexpr$ la CPS-continuation et $aexp$ la continuation en source

```
(define (T-k expr k)
  (match expr
    [`(Lambda . ,_)       (k (M expr))]
    [ (? symbol?)    (k (M expr))]
    [`(,f ,e)
      ; =>
      (define $rv (gensym '$rv))
      (define cont `(lambda (,$rv) ,(k $rv)))
      (T-k f (lambda ($f)
               (T-k e (lambda ($e)
                        `(,$f ,$e ,cont))))))]))

(define (T-c expr c)
  (match expr
    [`(Lambda . ,_)       `(,c ,(M expr))]
    [ (? symbol?)   `(,c ,(M expr))]
    [`(,f ,e)
      ; =>
      (T-k f (lambda ($f)
                (T-k e (lambda ($e)
                         `(,$f ,$e ,c))))))]))

(define (M expr)
  (match expr
    [`(Lambda (,var) ,expr)
      ; =>
      (define $k (gensym '$k))
```

```
28        `(Lambda (,var ,$k) ,(T-c expr $k))]
29      [(? symbol?)  #;=>  expr]))
```

3 Autour de la sémantique *formelle* de Scheme

Voir absolument la sémantique formelle de R^5RS,ou Operational Semantics for R^5RS Scheme (J.MATTHEWS, R.B.FINDLER) ; le chapitre 10 *Aspects of Similix : a Partial Evaluator for a Subset of Scheme*, in Partial Evaluation and Automatic Program Generation (N.JONES, C.GOMARD, P.SEATOF) ; Formal Syntax and Semantics of Programming Languages: a Laboratory Based Approach (K. SLONNEGER, B. KURTZ), Continuations from Generalized Stack Inspection (G.PETTYJOHN, J.CLEMENTS, J.MARSHALL, S.KRISHNAMURTHI, M.FELLEISEN, at ICFP05) ; An Operational Semantics for Scheme (J.MATTHEWS, R.FINDLER ; in JFP2007) ; (How to Write a (Lisp) Interpreter (in Python)) par P.NORVIG ; etc...

Quelques remarques :

— le store associe une location à une valeur (éventuellement augmentée) etc...
— les environnements associent un symbole à une location (on veut le `set!`)
— il existe un environnement initial (dont l'initialisation en machine est ad-hoc, demandant un traitement particulier)
— la norme spécifie (informellement) que l'ordre d'évaluation des arguments n'est pas spécifié ; on peut (le plus simple) le fixer (par exemple de gauche à droite) dans la sémantique ; abstraire avec une permutation (figée) sur cet ordre ; abstraire avec une permutation (quelconque, et variant à chaque appel) de cet ordre.

4 GCC et MELT

Deux logiciels libres qui me tiennent à cœur :

— GCC : une collection de compilateurs GNU COMPILER COLLECTION voir **`gcc.gnu.org`** (plusieurs langages sources : C, C++, Ada, Go, ... ; plusieurs processeurs cibles ; x86, ARM, Sparc, MMIX, ... ; plusieurs systèmes hôtes et cibles (compilateur croisé) : Linux, Hurd, MacOSX, ...)
— MELT : un langage dédié spécifique (domain specific language) pour étendre GCC voir `gcc-melt.org` que je développe

La complexité de GCC (plus de 10 millions de lignes !) :

— un frontal (parmi plusieurs) dépendant du langage source
— un "middle-end" commun ("Tree"-s pour les operateurs, "Gimple"-s pour les instructions souvent à 3 adresses $x \leftarrow y + z$)
— "back-end" (parmi plusieurs) dépendant du processeur cible

En TP et chez vous : utilisez `gcc-4.8` [6] (i.e. 4.8.2) et MELT plugin 1.0.1 ou mieux ; en TP : `/opt/bin/` dans votre `PATH` donc `/opt/bin/gcc-4.8` ; vérifier la version avec `gcc -v` ou `gcc-4.8 -v` puis faire `ls $(gcc-4.8 -print-file-name=plugin)` regarder le code source de GCC, en salle de TP dans `/opt/gcc-4.8.2/` notamment `gcc/tree.def` et `gcc/gimple.def`, et celui de MELT dans `/opt/melt-plugin-1.0.1/` ... Jouer avec `gcc -fdump-tree-all` (dans un répertoire *quasi-vierge*, contenant seulement un petit `*.c`)

Faire une extension MELT qui compte le nombres d'opérations entières (via leur Gimple).

6. Sous Debian ou Ubuntu récents : `apt-get install gcc-4.8 gcc-4.8-plugin-dev`