Übungen zu Organisation und Architektur von Rechnern
SS 2002
Serie 3 — Musterlösung

Aufgabe 9 (10 Punkte)

Der Scheme-Code für das Trace-Tool (`semcd-trace.scm`):

```
1  (require-library "functio.ss")   ;; for function 'filter'
2
3  ;; Creates a list containing the numbers of the intervall ['from', 'to'] in ascending order.
4  (define (from-to from to)
5          (if (<= from to)
6              (cons from
7                    (from-to (+ from 1) to))
8              ()))
9
10 ;; Returns the last element of the given list.
11 (define (last lst)
12         (list-ref lst (- (length lst) 1)))
13
14 ;; Zips two lists of equal length together into a single list of pairs.
15 (define (zip lst1 lst2)
16         (match (list lst1 lst2)
17                [((head1 tail1 ...) (head2 tail2 ...))   (cons (list head1 head2) (zip tail1 tail2))]
18                [(()                ())                  ()]))
19
20 ;; Applies the transformation function 'op' to 'state' until a final state is reached or 'max-cnt'
21 ;; transformation steps have been done. Returns a list of states.
22 (define (iter-op op state final-state? max-cnt)
23         (if (< max-cnt 0)
24             ()
25             (cons state
26                   (if (final-state? state)
27                       ()
28                       (iter-op op (op state) final-state? (- max-cnt 1))))))
29
30 ;; Applies 'semcd' until a final state is reached or all states corresponding with the numbers given
31 ;; in the list 'cnt-list' have been generated.
32 ;; The parameter 'semcd' represents a function that simulates a SEMCD machine.
33 ;; The parameter 'print-semcd' represents a function that converts a SEMCD state into a string.
34 ;; The parameter 'cnt-lst' must evaluate to a list of numbers in ascending order.
35 ;; Returns a list of pairs with each pair consisting of a state number and a state (which has been
36 ;; applied to 'print-semcd').
37 (define (trace-semcd semcd print-semcd cnt-lst) (lambda (stacks)
38         (letrec ((max-cnt1  (last cnt-lst))                               ;; Get greatest state number
39                  (state-lst (iter-op semcd stacks string? max-cnt1))      ;; Calculate all states
40                  (max-cnt2  (- (length state-lst) 1))                     ;; Count the states
41                  (cnt-lst2  (if (> max-cnt1 max-cnt2)                     ;; Patch 'cnt-lst':
42                                 (append (filter (lambda (x) (< x (- max-cnt2 1)))
43                                                 cnt-lst)                  ;; Remove superfluous state numbers
44                                         (list (- max-cnt2 1)              ;; Append number of final state ...
45                                               max-cnt2))                  ;; ... and message instead
46                                 cnt-lst)))
47                 (zip cnt-lst2                                                 ;; Add state numbers
48                      (map (lambda(x) (if (string? x) x (print-semcd x)))      ;; Convert states into strings
49                           (map (lambda (x) (list-ref state-lst x))            ;; Collect the requested states
50                                cnt-lst2))))))
51
52 ;; Applies 'trace-semcd' to an expression instead of a state.
53 (define (trace-semcd-expr semcd print-semcd cnt-list) (lambda (expr)
54         ((trace-semcd semcd print-semcd cnt-list)
55                       (list () () () (list expr) ()))))               ;; Build initial state
```

Die etwas umständliche Konstruktion mit dem zusätzlichen `lambda` in den Zeilen 37 und 53 ist dadurch motiviert, daß es möglich sein soll, die Trace-Funktionen auch ohne den letzten Parameter aufzurufen (siehe `semcd-rec-exams.scm`). Da partielle Funktionsanwendungen in Scheme nicht erlaubt sind, muß der letzte Parameter von Hand separiert werden. (Diese Vorgehensweise wird Curryfizierung genannt.)

Aufgabe 10 (20 Punkte)

Aufbauend auf der Basis-SEMCD-Maschine muß die Funktion `print-expr` um Ausgaberegeln für die Strukturen `Prf`, `Bool` und `Cond` erweitert werden. Außerdem müssen in der Funktion `eval` einige Muster, entsprechend den in der Vorlesung angegebenen Transformationsregeln, hinzugefügt werden. Der Scheme-Code ist im Abschnitt für Aufgabe 11 angegeben.

Abweichend von den Vorgaben auf dem Aufgabenzettel verwendet die Musterlösung in der Struktur `Prf` ein zusätzliches Feld `cnt`, das die Stelligkeit der primitiven Funktion angibt. Dadurch lassen sich ein- oder mehrstellige Funktionen einfach nachrüsten, es muß lediglich die Funktion `apply-prf` um entsprechende Einträge erweitert werden.

Folgende Punkte sind bezüglich der Funktion `eval` zu beachten:

- Eine primitive Funktion ist ihr eigener Wert, d. h. falls sie nicht auf Argumente angewendet wird, muß sie mit Hilfe der Atom-Regel (Regel 8) vom C- auf den S-Stack transportiert werden. Beispiel:

$$\overline{@}^{(2)}\ 0\ 0\ \boxed{\overline{@}^{(2)}\ +\ \lambda^{(2)}\ x\ x} \quad\rightsquigarrow\quad \overline{@}^{(2)}\ 0\ 0\ \boxed{+} \quad\rightsquigarrow\quad 0 \quad .$$

- Es ist dafür zu sorgen, daß eine Fehlermeldung generiert wird, falls eine primitive Funktion oder ein Konditional auf eine falsche Anzahl von Argumenten angewendet wird.
- Es sind geeignete Maßnahmen zu treffen, falls die Anwendung einer primitiven Funktion nicht ausgewertet werden kann, etwa weil die Argumente vom falschen Typ sind. Der Vorgehensweise von DrScheme folgend wäre es naheliegend, eine Fehlermeldung zu generieren. Es wäre aber auch denkbar, die nicht auswertbare Applikation auf dem S-Stack zu reproduzieren. Für die Musterlösung wurde die erstgenannte Alternative gewählt.
- Für den Fall, daß ein Konditional nicht ausgewertet werden kann, da die Bedingung sich nicht zu einer Bool'schen Konstanten auswertet oder fehlt, gelten die Ausführungen des vorhergehenden Absatzes entsprechend.

  Dabei ist jedoch zu beachten, daß ein `Cond` zunächst in eine Closure verpackt werden muß, bevor es auf den S-Stack geschoben werden kann, da es u. U. relativ-freie Variablen enthält!

  Um die Closure-Bildung zu umgehen, darf angenommen werden, daß ein nicht direkt auswertbares Konditional keinen legalen Wert darstellt. (Das ist in Scheme übrigens auch der Fall!) Es kann dann darauf verzichtet werden, Konditionale jemals auf den S-Stack zu transportieren. Applikationen von Konditionalen können, wie auch in der Vorlesung angegeben, stattdessen direkt auf dem C-Stack transformiert werden. Falls das `Cond` nicht ausgewertet werden kann, muß eine Fehlermeldung generiert werden. In der Musterlösung wurde dieses vereinfachte Transformationsschema verwendet.

Hinweis: Das auf dem Aufgabenzettel angegebene Beispiel berechnet die Fakultät von 3. Ein Vergleich mit der Version aus Aufgabe 11 verdeutlicht, daß der Term

$$\overline{@}^{(2)}\ \lambda^{(2)}\ \texttt{fac}\ \mathit{body}\ \texttt{Y}$$

eine rekursive Funktion `fac` definiert. Rekursion läßt sich also unter alleiniger Verwendung von Abstraktionen und Applikationen realisieren! Der die Rekursion vorantreibende Term `Y` wird in der Literatur Fixpunkt-Kombinator oder auch Y-Kombinator genannt.

Aufgabe 11 (10 Punkte)

Für den primitiven Rekursionsoperator sind folgende Transformationsregeln geeignet:

$$\begin{array}{lcl}
\Big(\, S\ ,\ E\ ,\ \mathit{nil}\ ,\ \alpha^{(2)}fe : C\ ,\ D \,\Big) & \to & \Big(\, [E\ \alpha^{(2)}fe] : S\ ,\ E\ ,\ \mathit{nil}\ ,\ C\ ,\ D \,\Big) \\
\Big(\, S\ ,\ E\ ,\ \overline{@}^{(n,i)} : M\ ,\ \alpha^{(2)}fe : C\ ,\ D \,\Big) & \to & \Big(\, [E\ \alpha^{(2)}fe] : S\ ,\ E\ ,\ \overline{@}^{(n,i-1)} : M\ ,\ C\ ,\ D \,\Big) \\
\Big(\, [E'\ \alpha^{(2)}fe] : S\ ,\ E\ ,\ \mathit{nil}\ ,\ C\ ,\ D \,\Big) & \to & \Big(\, S\ ,\ \Big\langle f\ [E'\ \alpha^{(2)}fe] \Big\rangle : E'\ ,\ \mathit{nil}\ ,\ e\ ,\ (E,C,D) \,\Big) \\
\Big(\, [E'\ \alpha^{(2)}fe] : S\ ,\ E\ ,\ \overline{@}^{(n,i)} : M\ ,\ C\ ,\ D \,\Big) & \to & \Big(\, S\ ,\ \Big\langle f\ [E'\ \alpha^{(2)}fe] \Big\rangle : E'\ ,\ \overline{@}^{(n,i+1)} : M\ ,\ e\ ,\ (E,C,D) \,\Big)
\end{array}$$

Die ersten beiden Regeln bewegen einen Rekursionsterm vom C-Stack auf den S-Stack. Wichtig ist dabei, daß auf dem S-Stack eine Closure gebaut wird, da der Rekursionsterm relativ-freie Variablen enthalten könnte! Das zweite Beispiel auf dem Aufgabenzettel verdeutlicht dieses Problem: Das Vorkommen der Variablen `m` am Ende der zweiten Zeile ist an das davor stehende $\lambda^{(2)}$`m` gebunden. Bei naiver Transformation ohne Closure würde es jedoch parasitär an das $\lambda^{(2)}$`m` aus der dritten Zeile gebunden und als Ergebnis ergäbe sich nicht 1 sondern 2.

Die letzten beiden Regeln greifen, falls auf dem S-Stack eine Closure mit einem Rekursionsterm gefunden wird. Es soll dann der Rumpf des Rekursionsterms berechnet werden, wobei alle Vorkommen der Rekursionsvariablen durch den Term selbst zu ersetzen sind. Also wird der aktuelle Zustand auf dem Dump gesichert, der Rumpf auf den C-Stack gelegt und das Environment aus der Closure wiederhergestellt. Ferner muß im Environment die gewünschte Bindung zwischen Rekursionsvariablen und Rekursionsterm abgelegt werden. Eine weitere wichtige Modifikation betrifft den obersten Applikator auf dem M-Stack: Da ein Argument wieder von S zurück nach C transportiert worden ist, muß der Zähler inkrementiert werden!

Es mag auf den ersten Blick verblüffen, daß ein Rekursionsterm auf C erst nach S transportiert wird, um ihn gleich darauf wieder nach C zurück zu schieben. Alternativ könnten die Regeln so umgestaltet werden, daß ein Rekursionsterm direkt auf dem C-Stack durch seinen Rumpf ersetzt wird (nebst Modifikation von E und D). Dadurch ließe sich auch das „sinnlose“ runter- und wieder heraufzählen des Applikators vermeiden. Allerdings müßten dann die Regeln 3 und 4 der Basismaschine geändert werden, da sonst ein im Environment aufgefundener Rekursionsterm (*lookup*-Funktion) auf dem S-Stack landen würde!! Die o. a. Regeln fügen sich dagegen ohne Modifikationen in die Basismaschine ein.

Der Scheme-Code für die erweiterte SEMCD-Maschine inklusive der Änderungen aus Aufgabe 10 (`semcd-rec.scm`):

```
(load "semcd-basic.scm")

(define-structure (Rec  name expr))
(define-structure (Prf  cnt name))
(define-structure (Cond expr1 expr2))

;; Applies the given primitive function to its arguments
;; For the time being only binary functions are implemented
(define (apply-prf cnt name args)
        (match (list name args)
               ;; Arithmetic
               [("+"   (($ Num  e1) ($ Num  e2)))   (make-Num  (+  e1 e2))]
               [("-"   (($ Num  e1) ($ Num  e2)))   (make-Num  (-  e1 e2))]
               [("*"   (($ Num  e1) ($ Num  e2)))   (make-Num  (*  e1 e2))]
               [("/"   (($ Num  e1) ($ Num  e2)))   (make-Num  (/  e1 e2))]
               ;; Comparison
               [("<"   (($ Num  e1) ($ Num  e2)))   (make-Bool (<  e1 e2))]
               [("<="  (($ Num  e1) ($ Num  e2)))   (make-Bool (<= e1 e2))]
               [("="   (($ Num  e1) ($ Num  e2)))   (make-Bool (=  e1 e2))]
               [(">="  (($ Num  e1) ($ Num  e2)))   (make-Bool (>= e1 e2))]
               [(">"   (($ Num  e1) ($ Num  e2)))   (make-Bool (>  e1 e2))]
               ;; Logic
               [("and" (($ Bool e1) ($ Bool e2)))   (make-Bool (and e1 e2))]
               [("or"  (($ Bool e1) ($ Bool e2)))   (make-Bool (or  e1 e2))]
               ;; All patterns failed -> Create error message
               [_                                   "Illegal use of primitive function found!"]))

;; Decrements the counter of the top-most ApM-constructor on the given M-stack by 'cnt'
(define (dec-top cnt m)
        (match m [()                          m]
                 [(($ ApM cnt1 cnt2) m ...)   (cons (make-ApM cnt1
                                                              (- cnt2 cnt))
                                                    m)]))

;; SEMCD machine:
;; Returns the new stack constellation iff a transformation can be performed,
;; returns a string iff an error occurs or the final state is reached already,
;; returns the unmodified argument otherwise.
(define (semcd stacks)
        (match stacks ;;------------------------------------------------------------------------
                      ;; Recursion on S
                      [((($ Clos e2 ($ Rec name expr)) s ...)
```

```
  e m c d)                            (list s
                                            (cons-bind name
                                                       (make-Clos e2
                                                                  (make-Rec name expr))
                                                       e2)
                                            (dec-top -1 m)   ;; increment top-most applicator
                                            (list expr)
                                            (list e c d))]
;;-------------------------------------------------------------------------
;; Primitive function on S and application on M
[((($ Prf cnt1 name) s ...)
  e
  (($ ApM cnt2 0) m ...)
  c d)                                (let ((cnt (- cnt2 1)))
                                           (if (= cnt1 cnt)
                                               (let ((res (apply-prf cnt1
                                                                     name
                                                                     (take s cnt1))))
                                                    (if (string? res)   ;; Error message?
                                                        (string-append "semcd: " res)
                                                        (list (cons res
                                                                    (drop s cnt1))
                                                              e
                                                              (dec-top 1 m)
                                                              c d)))
                                               (string-append "semcd: "
                                                              "Illegal application (Prf) found!"
                                                              " (expected #args: "
                                                              (number->string cnt1)
                                                              ", found #args: "
                                                              (number->string cnt)
                                                              ")")))]
;;-------------------------------------------------------------------------
;; Boolean on S, conditional on C and application on M
[((($ Bool val) s ...)
  e
  (($ ApM cnt2 1) m ...)
  (($ Cond expr1 expr2) c ...)
  d)                                  (if (= cnt2 2)
                                          (list s e m
                                                (cons (if val expr1 expr2)
                                                      c)
                                                d)
                                          (string-append "semcd: "
                                                         "Illegal application (Cond) found!"
                                                         " (expected #args: 1, found #args: "
                                                         (number->string (- cnt2 1))
                                                         ")"))]
;;-------------------------------------------------------------------------
;; Conditional on C but no boolean on S or no application on M
[(s e m
  (($ Cond expr1 expr2) c ...)
  d)                                  (match m [(($ ApM cnt2 1) m ...)
                                                "semcd: Argument of conditional is no bool!"]
                                               [_
                                                "semcd: Conditional without application found!"])]
;;-------------------------------------------------------------------------
;; Recursion on C
[(s e m
  (($ Rec name expr) c ...)
  d)                                  (list (cons (make-Clos e
                                                             (make-Rec name
                                                                       expr))
                                                  s)
                                            e
                                            (dec-top 1 m)
                                            c d)]
... same code as in basic version ...))
```

Der Scheme-Code für die erweiterte Print-Routine (`semcd-rec-print.scm`):

```
(load "semcd-basic-print.scm")

;; Prints an expression
(define (print-expr expr)
        (match expr [($ Rec name expr)      (string-append "Rec "
                                                           name
                                                           " "
                                                           (print-expr expr))]
                    [($ Prf 2 name)         name]                                   ;; Skip "{2}"
                    [($ Prf cnt name)       (string-append name
                                                           "{"
                                                           (number->string cnt)
                                                           "}")]
                    [($ Cond expr1 expr2)   (string-append "If "
                                                           (print-expr expr1)
                                                           " "
                                                           (print-expr expr2))]
                    ... same code as in basic version ...))
```

Der Scheme-Code für die Testumgebung (`semcd-rec-exams.scm`):

```
(load "semcd-rec.scm")
(load "semcd-rec-print.scm")
(load "semcd-trace.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Examples

;;----------------------------------------------------------------------------
;; Body of faculty function (used in exercises 10 and 11)
(define fac-body (make-Lam 2 (list (make-Var "n")
                    (make-Ap 2 (list (make-Ap 3 (list (make-Num 1) (make-Var "n")
                                                      (make-Prf 2 "<=")))
                                     (make-Cond (make-Num 1)
                                                (make-Ap 3 (list (make-Var "n")
                                                                 (make-Ap 2 (list (make-Ap 3 (list (make-Num 1)
                                                                                                  (make-Var "n")
                                                                                                  (make-Prf 2 "-")))
                                                                                  (make-Var "fac")))
                                                                 (make-Prf 2 "*")))))))))

;;----------------------------------------------------------------------------
;; Exercise 10
(define a        (make-Lam 2 (list (make-Var "x")
                    (make-Ap 2 (list (make-Lam 2 (list (make-Var "z")
                                                       (make-Ap 2 (list (make-Var "z")
                                                                        (make-Ap 2 (list (make-Var "x")
                                                                                         (make-Var "x")))))))
                                     (make-Var "y"))))))
(define y        (make-Lam 2 (list (make-Var "y")
                                   (make-Ap 2 (list a a)))))
(define fac-y    (make-Ap 2 (list (make-Lam 2 (list (make-Var "fac") fac-body))
                                  y)))
(define exer10   (make-Ap 2 (list (make-Num 3) fac-y)))

;;----------------------------------------------------------------------------
;; Exercise 11, 1st example: faculty function
(define fac-rec  (make-Rec "fac" fac-body))
(define exer11-1 (make-Ap 2 (list (make-Num 3) fac-rec)))                       ;; Try (fac 100) :-)

;;----------------------------------------------------------------------------
;; Exercise 11, 2nd example -> Should evaluate to 1
(define exer11-2 (make-Ap 2 (list (make-Bool #f)
                    (make-Ap 2 (list (make-Num 1)
                       (make-Lam 2 (list (make-Var "m")
                          (make-Rec "f" (make-Cond (make-Var "m")
                             (make-Ap 2 (list (make-Num 2)
                                              (make-Lam 2 (list (make-Var "m")
                                                                (make-Ap 2 (list (make-Bool #t)
                                                                                 (make-Var "f")))))))))))))))
```

```scheme

;;---------------------------------------------------------------------------
;; Tests whether isolated Prfs are handled correctly -> Should evaluate to 0
(define test3    (make-Ap 3 (list (make-Num 0) (make-Num 0)
                                  (make-Ap 2 (list (make-Prf 2 "+")
                                                   (make-Lam 2 (list (make-Var "x")
                                                                     (make-Var "x"))))))))

;;---------------------------------------------------------------------------
;; Tests whether wrongly applied Prfs are handled correctly -> Should evaluate to an error
(define test4-1  (make-Ap 2 (list (make-Num 1)
                                  (make-Prf 2 "+"))))
(define test4-2  (make-Ap 4 (list (make-Num 1) (make-Num 2) (make-Num 3)
                                  (make-Prf 2 "+"))))

;;---------------------------------------------------------------------------
;; Tests whether unevaluatable applications of Prfs are handled correctly
(define test5    (make-Ap 3 (list (make-Num 1)
                                  (make-Bool #t)
                                  (make-Prf 2 "+"))))

;;---------------------------------------------------------------------------
;; Tests whether Cond expressions are evaluated lazily -> Should evaluate to 2
;; Note: (omega omega) does not terminate!!
(define omega    (make-Lam 2 (list (make-Var "x")
                                   (make-Ap 2 (list (make-Var "x") (make-Var "x"))))))
(define test6    (make-Ap 2 (list (make-Bool #t)
                                  (make-Cond (make-Num 2) (make-Ap 2 (list omega omega))))))

;;---------------------------------------------------------------------------
;; Tests whether isolated Conds are handled correctly -> Should evualate to an error or 1
(define test7    (make-Ap 2 (list (make-Num 1)
                    (make-Lam 2 (list (make-Var "x")
                       (make-Ap 2 (list (make-Cond (make-Var "x") (make-Var "x"))
                          (make-Lam 2 (list (make-Var "y")
                             (make-Ap 2 (list (make-Num 2)
                                (make-Lam 2 (list (make-Var "x")
                                                  (make-Var "y"))))))))))))))

;;---------------------------------------------------------------------------
;; Tests whether wrongly applied Conds are handled correctly -> Should evaluate to an error
(define test8    (make-Ap 3 (list (make-Bool #t)
                                  (make-Bool #t)
                                  (make-Cond (make-Num 1) (make-Num 2)))))

;;---------------------------------------------------------------------------
;; Tests whether unevaluatable applications of Conds are handled correctly
(define test9    (make-Ap 2 (list (make-Num 0)
                                  (make-Cond (make-Num 1) (make-Num 2)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Testing

(map (trace-semcd-expr semcd print-semcd-s (from-to 0 5000))   ;; Show all states
     (list exer11-1 exer11-2
           test3 test4-1 test4-2 test5 test6 test7 test8 test9))

(map (trace-semcd-expr semcd print-semcd-s (list 5000))        ;; Show final state only
     (list exer10))

(stepper2 10 180                                               ;; Use teachpack 'semcd-gui.ss'
          ((trace-semcd-expr semcd print-semcd-l (from-to 0 5000))
                             exer11-1))
```