CONTINUATION PASSING STYLE

Bevezetés

A „folytatás”

A *folytatás* egy számítás befejezésének explicit reprezentációja. Használatának előnyei:

- a számítási sorrendet explicit módon megadja
- könnyű jobbrekurzív függvényeket írni
- a költséges, nem feltétlenül szükséges számítási ágakat le lehet nyesni
- modellezhető vele kivételkezelés
- visszalépések kezelésére is alkalmas

A stílus a 70-es években született.

A folytatás egy függvényargumentum. A függvény, ha kiszámította az eredményét, annak közvetlen visszaadása helyett *meghívja a folytatását, átadva neki az eredményt.*

Faktoriális mintapélda

Egyenes rekurzióval:

```
ffac             :: Integral a => a -> a
ffac 0           = 1
ffac n | n > 0 = n * ffac (n-1)
```

Gyűjtőargumentummal:

```
afac               :: Integral a => a -> a -> a
afac 0 f           = f
afac n f | n > 0 = afac (n-1) (n*f)
```

Folytatással:

```
kfac               :: Integral a => a -> (a -> a) -> a
kfac 0 k           = k 1
kfac n k | n > 0 = kfac (n-1) $ k . (n*)
```

A `\v -> k(n*v)`-vel ekvivalens `k . (n*)` a rekurzív hívás folytatása, amely

- megszorozza az argumentumát `n`-nel,
- majd a számítás eredményét továbbadja a kívülről kapott folytatásának, `k`-nak.

```
kfac 10 id == 3628800
```

Fibonacci mintapélda

Egyenes rekurzióval:

```
ffib   :: Integral a => a -> a
ffib 0 = 1
ffib 1 = 1
ffib n = ffib (n-1) + ffib (n-2)
```

Gyűjtőargumentummal:

```
afib       :: Integral a => a -> a -> a -> a
afib 1 _ b = b
afib n a b = afib (n-1) b (a+b)
```

Folytatással:

```
kfib     :: Integral a => a -> ((a,a) -> a) -> a
kfib 1 k = k (1,1)
kfib n k = kfib (n-1) $ \(a,b) -> k (b,a+b)
```

`kfib` meghívása:

```
kfib 50 fst == 12586269025
```

„Különbségi” listák

```
-- negpoz ls = az ‘ls’ számlista részben rendezett változata, amelyben a
-- negatív számok elöl, a pozitív számok hátul állnak, az eredeti
-- sorrendjük megőrzésével.

-- list comprehension + append
negpoz1 :: (Num a, Ord a) => [a] -> [a]
negpoz1 ls = [ n | n <- ls, n < 0 ] ++
             [ 0 | 0 <- ls ] ++
             [ p | p <- ls, p > 0 ]

-- continuation passing style - avoiding append with a trick very much like
-- difference lists in Prolog

negpoz2 :: (Num a, Ord a) => [a] -> [a]
negpoz2 ls = nzp where
                 (nzp,zp,p) = np ls $ \nzp0 zp0 p0 -> (nzp0,zp0,p0)
                 np [] k = k zp p []
                 np (xs) k | x  < 0 = np xs $ \nzp0 zp0 p0 -> k (x:nzp0) zp0 p0
                           | x == 0 = np xs $ \nzp0 zp0 p0 -> k nzp0 (0:zp0) p0
                           | x  > 0 = np xs $ \nzp0 zp0 p0 -> k nzp0 zp0 (x0)
```

EGY FUNKCIONÁLIS NYELV INTERPRETÁLÁSA

Háttér - 1

```
(* A strict functional language with integers, first-order functions,
   and exceptions * sestoft@dina.kvl.dk 2001-03-14, 2003-03-13

   Exceptions in the functional object language are modelled without
   using exceptions in the meta-language.  Instead the interpreter
   uses continuations to represent the rest of the computation.
   The interpreters are written in continuation-passing style, and
   implement tail calls in constant space.

   This file contains two interpreters:

     * coEval1 allows exceptions to be thrown, but not handled; this
       is implemented using a single continuation (which is used by
       ordinary computations but ignored when an exception is thrown)

     * coEval2 allows exceptions to be thrown and handled; this is
       implemented using two continuations: the success continuation
       is used by ordinary computations, and the error continuation
       is used when an exception is thrown.
*)
app load ["Env"];
open Env;
```

Forrás: `http://www.it-c.dk/courses/PFOO/F2005/plan.html`, Item 8

Háttér - 2

```
(* Env.sig: environments that map strings to data of type 'data *)
type 'data env
val empty    : 'data env
val lookup   : 'data env -> string -> 'data
val bind1    : 'data env -> string * 'data -> 'data env
val bind     : 'data env -> (string * 'data) list -> 'data env
val bindZip  : 'data env -> string list * 'data list -> 'data env
val fromList : (string * 'data) list -> 'data env

(* Env.sml -- environments implemented as association lists *)
type 'data env = (string * 'data) list
val empty = []

fun lookup []            x = raise Subscript
  | lookup ((y, v)::yr) x = if x=y then v else lookup yr x

fun bind1 env kv = kv :: env

fun bind env kvs = kvs @ env

fun bindZip env (ks, vs) = ListPair.zip(ks, vs) @ env

fun fromList kvs = kvs
```

A program reprezentációja

```
datatype exn =
    Exn of string

datatype expr =
    CstI of int
  | CstB of bool
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
  | If of expr * expr * expr
  | Letfun of string * string * expr * expr     (* (f, x, fbody, ebody) *)
  | Call of string * expr
  | Raise of exn
  | Handle of expr * exn * expr                 (* e1 handle exn => e2  *)

datatype value =
    Int of int
  | RClo of string * string * expr * vfenv      (* (f, x, fbody, bodyenv) *)
withtype vfenv = value env;

datatype answer =
    Success of int
  | Failure of string
```

Folytatásos interpretálás kivételkezelés nélkül - 1

```
(* This interpreter coEval1 takes the following arguments:

     * An expression e to evaluate.
     * An environment env in which to evalute it.
     * A success continuation cont which accepts as argument the value
       of the expression.

   It returns an answer: Success i or Failure s.  When the evaluation
   of e succeeds, it applies the success continuation to its result,
   and when e raises an exception (Exn s), it returns Failure s.
   Since there is no error continuation, there is no provision for
   handling raised exceptions.
*)

local
fun coEval1 (e : expr) (env : vfenv) (cont : int -> answer) : answer =
    case e of
        CstI i => cont i
      | CstB b => cont (if b then 1 else 0)
      | Var x  => (case lookup env x of
                         Int i => cont i
                       | _     => Failure "coEval1 Var")
```

Folytatásos interpretálás kivételkezelés nélkül - 2

```
| Let(x, erhs, ebody) =>
      coEval1 erhs env (fn xval =>
                            let val env1 = bind1 env (x, Int xval)
                            in coEval1 ebody env1 cont end)
| Prim(ope, e1, e2) =>
      coEval1 e1 env
      (fn i1 =>
       coEval1 e2 env
       (fn i2 =>
        case ope of
            "*" => cont(i1 * i2)
          | "+" => cont(i1 + i2)
          | "-" => cont(i1 - i2)
          | "=" => cont(if i1 = i2 then 1 else 0)
          | "<" => cont(if i1 < i2 then 1 else 0)
          | _   => Failure "unknown primitive"))
| Letfun(f, x, fbody, ebody) =>
      let val env1 = bind1 env (f, RClo(f, x, fbody, env))
      in coEval1 ebody env1 cont end
```

Folytatásos interpretálás kivételkezelés nélkül - 3

```
      | Call(f, earg) =>
            let val fclosure = lookup env f
            in  case fclosure of
                    RClo (f, x, fbody, fenv) =>
                        coEval1 earg env
                                (fn argv =>
                                 let
                                     val env2 = bind1 fenv (f, fclosure)
                                     val env3 = bind1 env2 (x, Int argv)
                                 in coEval1 fbody env3 cont end)
                  | _ => Failure "eval Call: not a function"
            end
      | If(e1, e2, e3) =>
            coEval1 e1 env
                    (fn b => if b<>0 then coEval1 e2 env cont
                             else coEval1 e3 env cont)
      | Raise (Exn s) => Failure s
      | Handle(e1, exn, e2) =>
             Failure "Not implemented"
in
    fun eval1 e env = coEval1 e env (fn v => Success v)
    fun run1 e = eval1 e Env.empty
end
```

Folytatásos interpretálás kivételkezeléssel - 1

```
(* This interpreter coEval2 takes the following arguments:

     * An expression e to evaluate.
     * An environment env in which to evalute it.
     * A success continuation cont which accepts as argument the value
       of the expression.
     * A error continuation econt, which is applied when an exception
       is thrown

   It returns an answer: Success i or Failure s.  When the evaluation
   of e succeeds, it applies the success continuation to its result,
   and when e raises an exception exn, it applies the failure
   continuation to exn.  The failure continuation may choose to handle
   the exception.
*)

local
fun coEval2 (e : expr) (env : vfenv)
            (cont : int -> answer) (econt : exn -> answer) : answer =
    case e of
        CstI i => cont i
      | CstB b => cont (if b then 1 else 0)
```

Folytatásos interpretálás kivételkezeléssel - 2

```
| Var x  => (case lookup env x of
                Int i => cont i
              | _     => Failure "coEval2 Var")
| Let(x, erhs, ebody) =>
      coEval2 erhs env (fn xval =>
                        let val env1 = bind1 env (x, Int xval)
                        in coEval2 ebody env1 cont econt end)
                       econt
| Prim(ope, e1, e2) =>
      coEval2 e1 env
      (fn i1 =>
       coEval2 e2 env
       (fn i2 =>
        case ope of
            "*" => cont(i1 * i2)
          | "+" => cont(i1 + i2)
          | "-" => cont(i1 - i2)
          | "=" => cont(if i1 = i2 then 1 else 0)
          | "<" => cont(if i1 < i2 then 1 else 0)
          | _   => Failure "unknown primitive") econt) econt
| Letfun(f, x, fbody, ebody) =>
      let val env1 = bind1 env (f, RClo(f, x, fbody, env))
      in coEval2 ebody env1 cont econt end
```

Folytatásos interpretálás kivételkezeléssel - 3

```
| Call(f, earg) =>
      let val fclosure = lookup env f
      in
          case fclosure of
              RClo (f, x, fbody, fenv) =>
                  coEval2 earg env
                          (fn argv =>
                           let
                               val env2 = bind1 fenv (f, fclosure)
                               val env3 = bind1 env2 (x, Int argv)
                           in coEval2 fbody env3 cont econt end)
                          econt
            | _ => raise Fail "eval Call: not a function"
      end
| If(e1, e2, e3) =>
      coEval2 e1 env (fn b =>
                  if b<>0 then coEval2 e2 env cont econt
                  else coEval2 e3 env cont econt) econt
| Raise exn => econt exn
```

Folytatásos interpretálás kivételkezeléssel - 4

```
      | Handle(e1, exn, e2) =>
            let fun econt1 exn1 =
                    if exn1 = exn then coEval2 e2 env cont econt
                                  else econt exn1
            in coEval2 e1 env cont econt1 end

in
    (* The top-level error continuation returns the continuation,
       adding the text Uncaught exception *)

    fun eval2 e env =
        coEval2 e env
               (fn v => Success v)
               (fn (Exn s) => Failure ("Uncaught exception: " ^ s))
    fun run2 e = eval2 e Env.empty
end
```

Példaprogramok – 1

```
(* Examples in abstract syntax *)

val ex1 = Letfun("f1", "x", Prim("+", Var "x", CstI 1),
                 Call("f1", CstI 12));

(* Factorial *)

val ex2 = Letfun("fac", "x",
                 If(Prim("=", Var "x", CstI 0),
                    CstI 1,
                    Prim("*", Var "x",
                              Call("fac", Prim("-", Var "x", CstI 1)))),
                 Call("fac", Var "n"));

val fac10 = eval1 ex2 (Env.fromList [("n", Int 10)]);
```

Példaprogramok – 2

```
(* Example: deep recursion to check for constant-space tail recursion *)
val exdeep = Letfun("deep", "x",
                    If(Prim("=", Var "x", CstI 0),
                       CstI 1,
                       Call("deep", Prim("-", Var "x", CstI 1))),
                    Call("deep", Var "n"));

fun rundeep n = eval1 exdeep (Env.fromList [("n", Int n)]);

(* Example: throw an exception inside expression *)

val ex3 = Prim("*", CstI 11, Raise (Exn "outahere"));

(* Example: throw an exception and handle it *)

val ex4 = Handle(Prim("*", CstI 11, Raise (Exn "Outahere")),
                 Exn "Outahere",
                 CstI 999);
```

Példaprogramok – 3

```
(* Example: throw an exception in a called function *)

val ex5 =
    Letfun("fac", "x",
           If(Prim("<", Var "x", CstI 0),
              Raise (Exn "negative x in fac"),
              If(Prim("<", Var "x", CstI 0),
                 CstI 1,
                 Prim("*", Var "x",
                           Call("fac", Prim("-", Var "x", CstI 1))))),
           Call("fac", CstI ~10));

(* Example: throw an exception but don't handle it *)

val ex6 = Handle(Prim("*", CstI 11, Raise (Exn "Outahere")),
                 Exn "Noway",
                 CstI 999);

val ex7 = Handle(Prim("*", CstI 11, Raise (Exn "Outahere")),
                 Exn "Outahere",
                 CstI 999);
```

EGY IMPERATÍV NYELV INTERPRETÁLÁSA

Háttér - 1

```
(* Naive imperative language with loops and exceptions
   sestoft@dina.kvl.dk 2001-03-17, 2003-03-13

   This file contains two interpreters:

     * coExec1 allows exceptions to be thrown, but not handled; this
       is implemented using a single continuation (which is used by
       ordinary computations but ignored when an exception is thrown)

     * coExec2 allows exceptions to be thrown and handled; this is
       implemented using two continuations: the success continuation
       is used by ordinary computations, and the error continuation is
       used when an exception is thrown.
*)

app load ["Int", "Naivestore"];

open Naivestore;

(* A store is just a mapping from ints to ints; no lvalue/rvalue distinction *)

type sto = int naivesto
```

Forrás: `http://www.it-c.dk/courses/PFOO/F2005/plan.html`, Item 8

Háttér - 2

```
(* Naivestore *)

type 'data naivesto                          (* A map from string to 'data *)

val empty : 'data naivesto
val get   : 'data naivesto -> string -> 'data
val set   : 'data naivesto -> string * 'data -> 'data naivesto

(* Naivestore -- implemented as association lists *)

type 'data naivesto = (string * 'data) list

val empty = []

fun get []           x = raise Subscript
  | get ((y, v)::yr) x = if x=y then v else get yr x

fun set sto (k, v) = (k, v) :: sto
```

A program reprezentációja

```
(* A computation may terminate normally or throw an exception: *)

datatype answer =
    Success
  | Failure of string;

datatype exn =
    Exn of string

datatype expr =
    CstI of int
  | Var of string
  | Prim of string * expr list

datatype stmt =
    Asgn of string * expr
  | If of expr * stmt * stmt
  | Block of stmt list
  | For of string * expr * expr * stmt
  | While of expr * stmt
  | Print of expr
  | Throw of exn
  | TryCatch of stmt * exn * stmt
```

Egyszerű interpretálás kivételkezelés nélkül

```
(* Evaluation of expressions without side effects and exceptions *)

fun eval e (sto : sto) : int =
    case e of
        CstI i => i
      | Var x  => get sto x
      | Prim(ope, [e1, e2]) =>
            let val i1 = eval e1 sto
                val i2 = eval e2 sto
            in
                case ope of
                    "*"  => i1 * i2
                  | "+"  => i1 + i2
                  | "-"  => i1 - i2
                  | "==" => if i1 = i2 then 1 else 0
                  | "<"  => if i1 < i2 then 1 else 0
                  | _    => raise Fail "unknown primitive"
            end
      | Prim _ => raise Fail "unknown primitive"
```

Folytatásos interpretálás kivételkezelés nélkül - 1

```
(* This interpreter coExec1 takes the following arguments:

   * A statement stmt to execute.
   * A naive store mapping names to values.
   * A success continuation cont, for normal termination.  By
     discarding the continuation, it can terminate abnormally (when
     executing a Throw statement), but it cannot handle thrown
     exceptions (because it has no error continuation).
*)
local
fun coExec1 stmt sto (cont : sto -> answer) : answer =
    case stmt of
        Asgn(x, e) =>
            cont (set sto (x, eval e sto))
      | If(e1, stmt1, stmt2) =>
            if eval e1 sto <> 0 then
                coExec1 stmt1 sto cont
            else
                coExec1 stmt2 sto cont
      | Block stmts =>
            let fun loop []        sto = cont sto
                  | loop (s1::sr) sto = coExec1 s1 sto (fn sto => loop sr sto)
            in loop stmts sto end
```

Folytatásos interpretálás kivételkezelés nélkül - 2

```
| For(x, estart, estop, body) =>
      let val start = eval estart sto
          val stop  = eval estop  sto
          fun loop i sto =
              if i <= stop then
                  coExec1 body (set sto (x, i)) (fn sto => loop (i+1) sto)
              else
                  cont sto
      in loop start sto end
| While(e, body) =>
      let fun loop sto =
              if eval e sto <> 0 then
                  coExec1 body sto loop
              else
                  cont sto
      in loop sto end
| Print e =>
      (print (Int.toString (eval e sto)); print "\n"; cont sto)
| Throw (Exn s) =>
      Failure ("Uncaught exception: " ^ s)
| TryCatch _ =>
      Failure "TryCatch is not implemented"
```

Folytatásos interpretálás kivételkezelés nélkül - 3

```
in
    fun run1 stmt : answer =
        coExec1 stmt empty (fn sto => Success)
end;
```

Folytatásos interpretálás kivételkezeléssel - 1

```
(* This interpreter coExec2 takes the following arguments:

   * A statement stmt to execute.
   * A naive store mapping names to values.
   * A success continuation cont, for normal termination.  By
     discarding the continuation, it can terminate abnormally (when
     executing a Throw statement), but it cannot handle thrown
     exceptions (because it has no error continuation).
   * An error continuation econt for abnormal termination.  The error
     continuation receives the exception and the store, and decides
     whether it wants to handle the exception or not.  In the former
     case it executes the handler’s statement body; in the latter case
     it re-raises the exception, by applying the handler’s own error
     continuation.
*)

local

fun coExec2 stmt sto
         (cont : sto -> answer) (econt : exn * sto -> answer) : answer =
    case stmt of
        Asgn(x, e) =>
            cont (set sto (x, eval e sto))
```

Folytatásos interpretálás kivételkezeléssel - 2

```
| If(e1, stmt1, stmt2) =>
      if eval e1 sto <> 0 then
          coExec2 stmt1 sto cont econt
      else
          coExec2 stmt2 sto cont econt
| Block stmts =>
      let fun loop []       sto = cont sto
            | loop (s1::sr) sto =
              coExec2 s1 sto (fn sto => loop sr sto) econt
      in loop stmts sto end
| For(x, estart, estop, stmt) =>
      let val start = eval estart sto
          val stop  = eval estop  sto
          fun loop i sto =
              if i <= stop then
                  coExec2 stmt (set sto (x, i))
                               (fn sto => loop (i+1) sto)
                               econt
              else
                  cont sto
      in loop start sto end
```

Folytatásos interpretálás kivételkezeléssel - 3

```
      | While(e, stmt) =>
            let fun loop sto =
                        if eval e sto <> 0 then
                              coExec2 stmt sto (fn sto => loop sto) econt
                        else
                              cont sto
            in loop sto end
      | Print e =>
            (print (Int.toString (eval e sto)); print "\n"; cont sto)
      | Throw exn =>
            econt(exn, sto)
      | TryCatch(stmt1, exn, stmt2) =>
            let fun econt1 (exn1, sto1) =
                        if exn1 = exn then coExec2 stmt2 sto1 cont econt
                                      else econt (exn1, sto1)
            in coExec2 stmt1 sto cont econt1 end
in
    fun run2 stmt : answer =
        coExec2 stmt empty
                (fn sto => Success)
                (fn (Exn s, sto) => Failure ("Uncaught exception: " ^ s))
end;
```

Példaprogramok – 1

```
(* Abruptly terminating a for loop *)

val ex1 =
    For("i", CstI 0, CstI 10,
        If(Prim("==", [Var "i", CstI 7]),
           Throw (Exn "seven"),
           Print (Var "i")));

(* Abruptly terminating a while loop *)

val ex2 =
    Block[Asgn("i", CstI 0),
          While (CstI 1,
                 Block[Asgn("i", Prim("+", [Var "i", CstI 1])),
                       Print (Var "i"),
                       If(Prim("==", [Var "i", CstI 7]),
                          Throw (Exn "seven"),
                          Block []))]),
          Print (CstI 333333)];
```

Példaprogramok – 2

```
(* Abruptly terminating a while loop, and handling the exception *)

val ex3 =
    Block[Asgn("i", CstI 0),
          TryCatch(Block[While (CstI 1,
                                Block[Asgn("i", Prim("+", [Var "i", CstI 1])),
                                      Print (Var "i"),
                                      If(Prim("==", [Var "i", CstI 7]),
                                         Throw (Exn "seven"),
                                         Block [])]),
                         Print (CstI 111111)],
                   Exn "seven",
                   Print (CstI 222222)),
          Print (CstI 333333)];
```