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
- 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 `(k . (n*))` a rekurzív hívás folytatása, amely

- megszorozza az argumentumát `n`-nel,
- majd a számítás eredményét továbbadja az 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 (\ n z p -> (n,z,p))
                 np []     k          = k zp p []
                 np (x:xs) k | x < 0  = np xs $ \n0 z p -> k (x:n0) z p
                             | x == 0 = np xs $ \n z0 p -> k n (0:z0) p
                             | x > 0  = np xs $ \n z p0 -> k n z (x:p0)
```

EGY IMPERATÍV NYELV INTERPRETÁLÁSA

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
                  | "==" => bool2int (i1 = i2)
                  | "<"  => bool2int (i1 < i2)
                  | _    => raise Fail "unknown primitive"
            end
      | Prim _ => raise Fail "unknown primitive"
```

Folytatásos interpretálás kivételkezelés nélkül

```
(* * 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 int2bool(eval e1 sto) 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
      | 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)) (loop (i+1))
                    else
                        cont sto
            in loop start sto end
      | While(e, body) =>
            let fun loop sto =
                    if int2bool(eval e sto) 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"
in

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

Folytatásos interpretálás kivételkezeléssel

```
(* * 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))
      | If(e1, stmt1, stmt2) =>
            if int2bool(eval e1 sto) 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 (loop sr) 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)) (loop (i+1)) econt
                    else
                        cont sto
            in loop start sto end
      | While(e, stmt) =>
            let fun loop sto =
                    if int2bool(eval e sto) then
                        coExec2 stmt sto loop 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)];
```

EGY FUNKCIONÁLIS NYELV INTERPRETÁLÁSA

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 list
  | 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 = (string, value) env;

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

Folytatásos interpretálás kivételkezelés nélkül

```
(* 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 (bool2int b)
      | Var x  => (case lookup env x of
                       Int i => cont i
                     | _     => Failure "coEval1 Var")
      | 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(bool2int (i1 = i2))
                | "<" => cont(bool2int (i1 < i2))
                | _   => Failure "unknown primitive"))
      | Prim(ope, _) => Failure "primitive arity"
      | Letfun(f, x, fbody, ebody) =>
            let val env1 = bind1 env (f, RClo(f, x, fbody, env))
            in coEval1 ebody env1 cont end
      | Call(f, earg) =>
            (case lookup env f of
                 fclosure as RClo (f, x, fbody, env1) =>
                     coEval1 earg env
                            (fn argv =>
                             let val env2 = bind1 env1 (f, fclosure)
```

```
                                 val env3 = bind1 env2 (x, Int argv)
                             in coEval1 fbody env3 cont end)
               | _ => Failure "coEval1 Call")
      | If(e1, e2, e3) =>
            coEval1 e1 env
                    (fn b => if int2bool b 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)
end
```

Folytatásos interpretálás kivételkezeléssel

```
(* 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 (bool2int b)
      | 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(bool2int (i1 = i2))
                | "<" => cont(bool2int (i1 < i2))
                | _   => Failure "unknown primitive") econt) econt
      | Prim(ope, _) => Failure "primitive arity"
      | Letfun(f, x, fbody, ebody) =>
            let val env1 = bind1 env (f, RClo(f, x, fbody, env))
            in coEval2 ebody env1 cont econt end
      | Call(f, earg) =>
            (case lookup env f of
                 fclosure as RClo (f, x, fbody, env1) =>
                     coEval2 earg env
```

```
                     (fn argv =>
                      let val env2 = bind1 env1 (f, fclosure)
                          val env3 = bind1 env2 (x, Int argv)
                      in coEval2 fbody env3 cont econt end) econt
               | _ => Failure "coEval2 Call")
      | If(e1, e2, e3) =>
            coEval2 e1 env (fn b =>
                              if int2bool b then coEval2 e2 env cont econt
                              else coEval2 e3 env cont econt) econt
      | Raise exn => econt exn
      | 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))
end
```

Példaprogramok – 1

```
(* 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)]);

(* 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)]);
```

Példaprogramok – 2

```
(* 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);

(* 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));
```

Példaprogramok – 3

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

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