Magasabbrendű funkcionális programozás

Hanák Péter

hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

OM Kutatás-Fejlesztési Helyettes Államtitkárság

Hanák Dávid

dhanak@inf.bme.hu

Számítástudományi és Információelméleti Tanszék

Csala Viktor

viktor@csala.net

AAM Technologies Kft.

BEVEZETÉS

Bevezetés

Az előadás felépítése

- 1. alkalom: Az OCaml nyelv alapjai, funkcionális programozás OCaml-ben
- 2. alkalom: Az OCaml modul kezelése; Objektum orientált programozás OCaml-ben

Források

- A Caml website-ja: http://caml.inria.fr
- A Caml történetének összefoglalása a http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html
- Az előadás alapját a Developing Applications with Objective Caml könyv képezi, ami mind francia, mind angol nyelven letölthető a site-ról.
- A Caml levelezőlistája: caml-list@inria.fr (előtte fel kell iratkozni a caml-list-request@inria.fr címen).

Miért jó Caml?

- Nyelvi elemeit tekintve nem jobb és nem rosszabb, mint bármely Standard ML megvalósítás (vannak jobban és vannak rosszabban használható elemek).
- Nagyon hatékony, a jelenlegi megvalósítás összemérhető a C nyelv teljesítményével (a mérés eredménye: http://www.bagley.org/~doug/shootout/).
- Nagyon komoly fejlesztő gárda és felhasználói közösség áll mögötte.
- Üzleti célokra is korlátozások nélkül felhasználható (számos jelentős alkalmazást írtak már OCaml nyelven, ezek egy részét a website is felsorolja).

AZ OCAML NYELV ALAPJAI

A Caml nyelv rövid története

- ML-t (meta-language) Robin Milner tervezte Edinburgh-ban az LCF bizonyítás levezető rendszer leíró nyelveként.
- 1980-81-ben az INRIA Formel projektje érdeklődni kezdett a nyelv iránt.
- 1984 körül a projekt több lökést is kapott:
  - Luca Cardelli megalkotta a Functional Abstract Machine-t, ami az ML fordítását sokkal hatékonyabbá tette.
  - Robin Milner elkerülendő a nyelv Lispéhez hasonló széttagolódását definiálta a Standard ML-t, amit később kiegészítettek Dave McQueen által tervezett modul rendszerrel.
  - Pierre-Louis Curien kidolgozta kategórikus kombinátorok elméletét és megmutatta a λ-kalkulus ezzel kapcsolatos párhuzamait. Ez inspirálta a Categorical Abstract Machine megalkotását.
- $\Rightarrow$ Kialakult a Categorical Abstract Machine Language, vagyis a Caml.

A nyelv legalapvetőbb elemei

- Három különböző deklaráció:

érték deklaráció	:	**let**
kivétel deklaráció	:	**exception**
típus deklaráció	:	**type**

- A nyelv interaktív interpreterrel (is) rendelkezik (toplevel), így a deklarációk helyben a rendszerrel folytatott párbeszéd során is kiadhatók.
- Példa:

```
# let rec fact = function n -> if n < 2 then 1 else n * fact(n-1);;
val fact : int -> int = <fun>
# fact 8;;
- : int = 40320
```

Alaptípusok

- Számok:
  - int (pl. 3)
  - float (pl. 3.2)
  - A két típus nem kompatíbilis, konvertálni kell őket közös használat esetén (float_of_int és int_of_float függvények)
  - Műveletek: +, -, … (int) és +., -., … (float)
- Karakterek (pl. ’A’)
- Karakterfüzérek (pl. ”String”)
- Boolean (**true** és **false**)
- Unit (())
- Direktszorzatok és többesek
  (pl. (1, **true**) vagy (1, 2.1, ”String”, **true**))
- Listák (pl. [] vagy 1::2::3::[] vagy [1; 2; 3])

Feltételes kifejezés

- **if** $expr_1$ **then** $expr_2$ **else** $expr_3$
- Az else ág elhagyható, de ebben az esetben implicit **else** ()-t feltétlez a fordító (vagy az interpreter), ami természetszerűen típus ütközést okozhat.
- A feltételes kifejezés teljes értékű kifejezésnek számít, aminek típusa és értéke van. Emiatt az $expr_2$ és az $expr_3$ típusának ugyanannak kell lennie (ez nem minden esetben van így, ld. később).
- Pl. az (**if** 3 = 5 **then** 8 **else** 10) + 5 érvényes kifejezés, aminek az értéke 15.

Érték deklaráció

- Egyszerű: **let** *name* = *expr* ;;
- Egyidejű: **let** $name_1 = expr_1$
  and $name_2 = expr_2$
  and …**;;**
- Lokális: **let** *name* = $expr_1$
  in $expr_2$;;
- Lokális, egyidejű: **let** $name_1 = expr_1$
  and $name_2 = expr_2$
  and …
  and $name_n = expr_n$
  in $expr_{n+1}$;;

Függvény kifejezések - 1

- Alapeset: **function** *p* -> *expr* ;;
- Alternatív szintaxis: **fun** p_1 ... p_n -> *expr* ;;
  (ez ekvivalens a kifejezéssel) **function** p_1 -> … -> **function** p_n -> *expr* ;;
- Closure:

```
let m = 3 ;;
let f = function x -> x + m ;;
let m = 5 ;;
f 2 ;; ⇒ az eredménye 5.
```

Függvény kifejezések - 2

- Nevesített függvény: **let** *f* = **function** *p* -> *expr* ;;
  vagy **let** *g* = **fun** p_1 ... p_n -> *expr* ;;
- Alternatív szintaxis: **let** *h* p_1 ... p_n = *expr* ;;
- Infix függvények: (op)
  pl. **let** *succ* = (+) 1 ;;
  succ 4 ;;⇒ 5;
  és **let** (++) *c1 c2* =
  (*fst c1*) + (*fst c2*),
  (*snd c1*) + (*snd c2*)
- Magasabbrendű függvények:

let *h* = **function** *f* ->
function *y* -> (*f y*) *y* ;;

Rekurzív deklarációk

- Rekurzív deklaráció: **let rec** *name* = *expr* ;;
  vagy **let rec** $name_1 = expr_1$
  and $name_2 = expr_2$
  and … ;;
  vagy **let rec** $name_1 = expr_1$
  and …
  and $name_n = expr_n$
  in $expr_{n+1}$;;
- Függvényeknél az *expr* lehet **function** vagy **fun**, illetve használható a paraméter megadásos (**let rec** $f\ p_1 \ldots p_n = expr$) forma is.

Rekurzív (nem függvény) értékek

- Rekurzív deklarációval létrehozhatók végtelen ismétlődő adatszerkezetek is, pl:
  # **let rec** *l* = 1::*l* ;;
  val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
  1; 1; 1; 1; 1; …]
- Ezekkel a kifejezésekkel nagyon óvatosan kell bánni (jellemzően emiatt ritkán is használják), mert a rekurzív függvények az ilyen értékekre meghívva memória túlcsordulást eredményezhetnek (és a beépített függvények nagy része, pl. a *List.size* függvény rekurzív).

Polimorfizmus és típus kényszerek

- A Caml teljes mértékben támogatja a polimorfizmust, minden kifejezéshez megkeresi a legáltalánosabb típust (principal type, ld. a λ-kalkulus előadást, illetve a Hindley-Milner algoritmust).
- Ennek megfelelően minden paraméterezhető kifejezés meghívható minden, a típus mintájának megfelelő paraméterrel.
- A nyelv rendelkezik a Church féle típuskényszer megadásának lehetőségével (de egy adott kifejezésben ugyanahhoz a névhez nem rendelhető több különböző típus). A típuskényszer szintaxisa:
  (*x* : type).
- Pl. **let** *make_pair_int* (*x* : int) (*y* : int) = *x*, *y* ;;

Minta illesztés - 1

- Alap kifejezés:

match *expr* **with**
| p_1 -> $expr_1$
| p_2 -> $expr_2$
| …

- Ugyanabban a mintában elvben nem szerepelhet egynél többször ugyanaz a változó (ez általában fordítási hibát okoz).
- A helyettesítő (wildcard) minta: _ (ez részminta helyesetesítésére is használható).
- Minták kombinálhatóak: p_1 | … | p_n
- Karakter intervallumok: ’c_1’.. ’c_n’, ami ekvivalens a ’c_1’| … | ’c_n’ mintával.
- A mintaillesztéshez név rendelhető: (*p* **as** *name*)

Minta illesztés - 2

- A mintákhoz segédfeltételek rendelhetők, amiket sikeres illesztés esetén értékel ki a rendszer és ha a kiértékelés eredmény false, akkor folytatja a minták illesztését:

  match *expr* **with**
  …
  | *p* **when** *cond* -> *expr*
  …

- A listák illesztéséhez használható mind a [], mind a *h*::*t* konstruktor.
- A függvények paramétereinek illesztése a következő konstrukcióban is történhet:

  function
  | p_1 -> $expr_1$
  | p_2 -> $expr_2$
  | …

  Természetesen ez a **match** használatával is történhet. A **fun** konstrukcióban a mintaillesztés nem alkalmazható, több paraméter együttes illesztését csak többesek alkalmazásával tehetjük meg.

Típus deklaráció - 1

- Alapeset: **type** *name* = *typedef* ;;
- Együttes deklarálás: **type** $name_1 = typedef_1$ **and** $name_2 = typedef_2$ **and** …
- Paraméteres típus: **type** *'a name* = *typedef* ;;
  illetve **type** ($'a_1$ … $'a_n$) *name* = *typedef*;;
- A típusdefiníció (typedef) lehet a meglévő típusokból összeállított kifejezés is.
- Rekordok: **type** *name* = { $name_1 : t_1$; …, $name_n : t_n$ } ;;

  A rekordok létrehozása: {$name_1 = expr_1$; …; $name_n = expr_n$}

  illetve egy másik rekordon alapuló rekord (csak néhány mező értéke tér el):

  {*name* **with** $name_1 = expr_1$; …; $name_n = expr_n$}

Típus deklaráció - 2

- Unió típus: **type** *name* = …
  | $Name_i$ **…**
  | $Name_j$ **of** t_j ...
  | $Name_k$ **of** t_1 * ... * t_n …;;
- Konstans konstruktor: **type** *name* = $Name_1$ | … | $Name_n$;;
- Konstruktorok argumentumokkal:

  type *name* =
  | $Name_1$ *of* $typeexpr_1$
  | $Name_2$ *of* $typeexpr_2$
  | …
- A típusdeklarációk rekurzívak is lehetnek.

Kivételek

- Kivételek deklarációja: **exception** *Name* ;;
  vagy **exception** *Name* **of** *typeexpr* ;;
  Az utóbbi esetben a típuskifejezés nem tartalmazhat típusváltozókat (ennek okát lásd a következő dián).
- Kivételek kikényszerítése: **raise** *Name* ;;
  vagy **raise** (*Name expr*) ;;
  vagy **raise** (*Name* $expr_1$ ... $expr_n$) ;;
  vagy név nélkül **failwith** *karakterfüzér* ;;
- Kivételek kezelése: **try** *expr* **with**
  | p_1 -> $expr_1$
  | p_2 -> $expr_2$
  | …

Függvények visszatérési értéke

- A függvények kimenetének típusa jellemzően függ az argumentumainak típusaitól.
- Néha ez nem igaz:

```
let f x = [] ;; típusa 'a -> 'b list
let f x = f x ;; típusa 'a -> 'b
let f x = failwith "any" ;; típusa 'a -> 'b
let f x = List.hd [] ;; típusa 'a -> 'b
```

- Ha a kivételekben megengedett lenne a típusváltozó:

```
exception Exn of 'a ;; (ez nem értelmezhető igazából)
let f = function
        |    0 -> raise (Exn false)
        |    n -> n + 1 ;;
let g n = try f n with Exn x -> x + 1 ;;
```

ekkor *g* 0 kiértékelése egy int és egy boolean összeadását eredményezné.

AZ OCAML MODUL KEZELÉSE

Interface és implementációs file-ok

- Az OCaml legegyszerűbb modulkezelése az interface és implementációs file-okon alapul.
- A két file neve azonos (ez lesz a modul neve is nagy kezdőbetűvel), kivéve a kiterjesztésüket. Az interface file kiterjesztése .mli, az implementációsé .ml (pl. stack.mli és stack.ml, amik a Stack modul-t valósítják meg).
- A két file külön fordítható, összekötésük a linkelés során történik meg, így egy adott interface-nek több implementációja is lehet.
- Az interface file tartalmazza a modul típusainak (**type** kulcsszó), kivételeinek (**exception** kulcsszó) és értékeinek (beleértve a függvényeket is, **val** kulcsszó) szignatúra definícióját.
- Ez a fajta modul kezelés már elavultnak tekinthető, ezért egyre kevésbé használják.

Példa – stack.ml

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with
        hd :: tl -> s.c <- tl; hd
    |   [] -> raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

Példa – stack.mli

```
(* Module [Stack]: last-in first-out stacks *)
(* This module implements stacks (LIFOs), with in-place
   modification. *)
type 'a t
(* The type of stacks containing elements of type ['a]. *)
exception Empty
(* Raised when [pop] is applied to an empty stack. *)
val create: unit -> 'a t
(* Return a new stack, initially empty. *)
val push: 'a -> 'a t -> unit
(* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t -> 'a
(* [pop s] removes and returns the topmost element in stack [s],
   or raises [Empty] if the stack is empty. *)
val clear : 'a t -> unit
(* Discard all elements from a stack. *)
```

Példa – stack.mli

```
val length: 'a t -> int
(* Return the number of elements in a stack. *)
val iter: ('a -> unit) -> 'a t -> unit
(* [iter f s] applies [f] in turn to all elements of [s],
   from the element at the top of the stack to the element at the
   bottom of the stack. The stack itself is unchanged. *)
```

A modul nyelv alapjai – 1

- Szignatúra: **modul type** *NAME* =
  sig
  interface declaration
  end
- Struktúra: **module** *Name* =
  struct
  implementation definition
  end
- Mindkettőnek létezik anonim formája:
  sig *declaration* **end**
  struct *definition* **end**

A modul nyelv alapjai – 2

- Minden struktúrának van alap szignatúrája, amit a rendszer rendel hozzá a legáltalánosabb típus alapján.
- Megadható az explicit szignatúra is:

  module *Name* **:** *signature* =
  structure

  vagy **module** *Name* =
  (*structure* **:** *signature*)
- Explicit szignatúra esetében a rendszer megvizsgálja, hogy az illeszthető-e az alap szignatúrához.
- Hivatkozás modul elemére: $Name_1.name_2$
- Modul láthatóvá tétele (ekkor nem kell a hivatkozásnál a modul név és a pont): **open** *Name*
- Névütközésnél mindig az utoljára megnyitott modulbeli implementáció az érvényes.

Típus implementáció elrejtése

```
# module Int_Star =
    ( struct
        type t = int
        exception Isnul
        let of_int = function 0 -> raise Isnul | n -> n
        let mult = ( * )
      end
    :
      sig
        type t
        exception Isnul
        val of_int : int -> t
        val mult : t -> t -> t
      end
    ) ;;
module Int_Star :
  sig type t exception Isnul val of_int : int -> t
  val mult : t -> t -> t end
```

Értékek elrejtése

```
# module type GENSYM =
    sig
      val reset : unit -> unit
      val next : string -> string
    end ;;

# module Gensym : GENSYM =
    struct
      let c = ref 0
      let reset () = c:=0
      let next s = incr c ; s ^ (string of int -> c)
    end ;;
module Gensym : GENSYM
```

Modul több nézete

```
# module type USER_GENSYM =
    sig
      val next : string -> string
    end ;;
module type USER_GENSYM = sig val next : string -> string end

# module UserGensym = (Gensym : USER_GENSYM) ;;
module UserGensym : USER_GENSYM

# UserGensym.next "U" ;;
- : string = "U2"
# UserGensym.reset () ;;
Characters 0-16:
Unbound value UserGensym.reset
```

Típus megosztás modulok között

- A típusok megosztása történhet
  - típus kényszerek alkalmazásával:

    NAME **with type** $t_1 = t_2$ **and** …

    pl.: M1 = (M:S1 **with type** t = M.t)

    és M2 = (M:S2 **with type** t = M.t)

    akkor M1.t = M2.t (S1 és S2 ugyanannak a struktúrának a különböző nézetei)

    A típuskényszer teljes modulra is felírható:

    NAME **with module** $Name_1 = Name_2$
  - beágyazott modulokkal, ebben az esetben a megosztásban részt vevő modulok struktúrájának definiálása és példányosítása is egy másik modulba beágyazva történik megosztva a befoglaló modul absztrakt típusait (hívatkozás: ModuleName.SubModuleName.Name). A beágyazás tetszőleges mélységű lehet.

Paraméterezhető modulok

- Functor: **functor** (*Name* : *signature*) –> *structure*
- Alternatív szintaxis:
  module $Name_1$ ($Name_2$: *signature*) = *signature*
- A functoroknak tetszőleges számú paramétere lehet:
  functor ($Name_1$: $signature_1$) -> …
  functor ($Name_n$: $signature_n$) ->
  structure
- Ugyanez alternatív szintaxissal:
  module *Name* ($Name_1$: $signature_1$) …
  ($Name_n$: $signature_n$) =
  structure
- Functor használata:
  module *Name* = *functorName* ($structure_1$) …
  ($structure_n$)

Lokális modul definíció

```
# let sort l =
    let module M =
      struct
        type t = int
        let compare x y =
          if x < y then -1 else if x > y then 1 else 0
      end
    in
      let module MSet = Set.Make(M)
      in MSet.elements (List.fold right MSet.add l MSet.empty) ;;
val sort : int list -> int list = <fun>

# sort [ 5 ; 3 ; 8 ; 7 ; 2 ; 6 ; 1 ; 4 ] ; ;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

OBJEKTUM ORIENTÁLT PROGRAMOZÁS OCAML-BEN

Osztály deklaráció

```
# class point (x_init, y_init) =
    object
      val mutable x = x_init
      val mutable y = y_init
      method get_x = x
      method get_y = y
      method moveto (a,b) = x <- a ; y <- b
      method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
      method to_string () =
        "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
      method distance () = sqrt (float(x * x + y * y))
    end ;;
```

Osztályok használata

- Osztály egyedének létrehozása: **new** *name* $expr_1$ … $expr_n$

  # **let** *p1* = **new** *point* (0,0); ;
  val p1 : point = <obj>
  # **let** *p2* = **new** *point* (3,4); ;
  val p2 : point = <obj>
  # **let** *coord* = (3,0); ;
  val coord : int * int = 3, 0
  # **let** *p3* = **new** *point coord*; ;
  val p3 : point = <obj>

- Üzenet küldése: obj_1*#name* p_1 … p_n

  # *p1#get x* ;;
  - : int = 0
  # *p2#get y* ;;
  - : int = 4

Aggregáció

```
# class picture n =
    object
      val mutable ind = 0
      val tab = Array.create n (new point(0,0))
      method add p =
        try tab.(ind)<-p ; ind <- ind + 1
        with Invalid argument("Array.set")
          -> failwith ("picture.add:ind =" ^ (string_of_int ind))
      method remove () = if (ind > 0) then ind <-ind-1
      method to string () =
        let s = ref "["
        in for i=0 to ind-1 do
          s:= !s ^ " " ^ tab.(i)#to string() done ;
          (!s) ^ "]"
    end ;;
```

Öröklés

```
# class rectangle ((p1, p2) :'a) =
    object
      constraint 'a = point * point
      inherit printable ()
      inherit geometric_object ()
      val llc = p1
      val urc = p2
      method to string () =
        "["^llc#to_string()^","^urc#to_string()^"]"
      method compute_area() =
        float (abs(urc#get_x - llc#get_x) *
               abs(urc#get_y - llc#get_y))
      method compute_peri() =
        float ((abs(urc#get_x - llc#get_x) +
               abs(urc#get_y - llc#get_y)) * 2)
    end ;;
```

Előredefiniált referenciák (self és super)

```
# class colored_point (x,y) c =
    object (self)
      inherit point (x,y) as super
      val c = c
      method get_color = c
      method to_string () =
        super#to_string() ^ " [" ^ self#get_color ^ "] "
    end ;;
```

Késleltetett kötés

```
# class colored_point_1 coord c =
    object
      inherit colored_point coord c
      val true_colors = ["red"; "green"; "blue"; "yellow"]
      method get_color =
        if List.mem c true_colors then c else "UNKNOWN"
    end ;;

# let p1 = new colored_point (1,1) "blue as an orange" ;;
val p1 : colored_point = <obj>
# p1#to_string() ;;
- : string = "( 1, 1) [blue as an orange] "
# let p2 = new colored_point_1 (1,1) "blue as an orange" ;;
val p2 : colored_point_1 = <obj>
# p2#to_string() ;;
- : string = "( 1, 1) [UNKNOWN] "
```

Inicializálás

```
# class verbose_point p =
    object (self)
      inherit point p
      initializer
        let xm = string_of_int x and ym = string_of_int y
        in Printf.printf ">> Creation of a point at (%s %s)\n"
                         xm ym ;
           Printf.printf " , at distance %f from the origin\n"
                         (self#distance() ) ;
    end ;;

# new verbose_point (1,1);;
>> Creation of a point at (1 1)
, at distance 1.414214 from the origin
- : verbose_point = <obj>
```

Privát metódusok

```
# class point m1 (x0,y0) =
    object (self)
      inherit point (x0,y0) as super
      val mutable old x = x0
      val mutable old y = y0
      method private mem_pos () = old x <- x ; old y <- y
      method undo () = x <- old x; y <- old y
      method moveto (x1, y1) = self#mem pos () ;
                               super#moveto (x1, y1)
      method rmoveto (dx, dy) = self#mem pos () ;
                                super#rmoveto (dx, dy)
    end ;;
```

Absztrakt osztályok és metódusok

```
# class virtual printable () =
    object (self)
      method virtual to_string : unit -> string
      method print () = print_string (self#to_string())
    end ;;
class virtual printable :
  unit ->
  object
    method print : unit -> unit
    method virtual to_string : unit -> string
  end
```

Paraméterezhető osztályok

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ; ;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method
                snd : 'b end
```

Altípusok

```
# let pc = new colored point (4,5) "white";;
val pc : colored_point = <obj>
# let p1 = (pc : colored point :> point);;
val p1 : point = <obj>
# let p2 = (pc :> point);;
val p2 : point = <obj>
```

Fontos: az altípusok és az öröklés két különböző koncepció az OCamlben, egy típus lehet altípusa egy másiknak, anélkül, hogy köztük öröklési reláció lenne.

Funkcionális műveletek osztályokon

```
# class f_point p =
    object
      inherit point p
      method f_rmoveto_x (dx) = {< x = x + dx >}
      method f_rmoveto_y (dy) = {< y = y + dy >}
    end ;;
```