A co líné vyhodnocování

`type 'a lazy_t = ...` V F#: `type 'a Lazy = ...`

Vytváří se pomocí `let l = lazy (5 + 5)` `let l = lazy (5 + 5)`

Vyhodnotí se pomocí `let h = Lazy.force l` `let h = l.Force ()`

`let force (x:Lazy<'a>) = x.Force ()`

Představa je taková, že se **lazy** `exp` nahrazuje za něco jako **fun** `() -> exp`

```
type 'a streamCell = Nil | Cons of 'a * 'a stream
and  'a stream = 'a streamCell lazy_t
let (@:@) a b = Cons (a,b)

let rec ones = lazy (1 @:@ ones)
let rec fromN n = lazy (n @:@ fromN (n+1))
let rec take n xs =
  if n = 0 then []
  else match force xs with
      Nil -> []
    | Cons (x, xs) -> x :: take (n-1) xs

let matchs seq nil cons = match force seq with
                            Nil -> nil
                          | Cons (x, xs) -> cons x xs
let rec take2 n xs =
  if n = 0 then []
  else matchs xs [] (fun x xs -> x :: take (n-1) xs)
```

Vlastní varianta lazy vyhodnocování

```
type 'a thunk_value = Thunk of (unit -> 'a) | Value of 'a
type 'a thunk = 'a thunk_value ref

let t_create thunk = ref (Thunk thunk)
let t_force thunk =
  match !thunk with
      Value value -> value
    | Thunk func -> let value = func ()
                    in thunk := Value value;
                       value
```

Výjimky

`1 / 0;;` -> `Exception: Division_by_zero.`

`raise : exn -> 'a`

Takže jde `1 + raise Not_found * 21`

Definované jsou výjimky `Not_found : exn`

`List.assoc "Milan" [("Petr", 1); ("Jana", 2)]` -> `Exception: Not_found.`

`Invalid_argument : string -> exn, Failure : string -> exn`

`[|5;6|].(2)` -> `Exception: Invalid_argument "index out of bounds".`

`int_of_string "1f"` -> `Exception: Failure "int_of_string".`

`Match_failure : string * int * int -> exn`

`let f x = match x with Some y -> y in`

`f None` -> `Exception: Match_failure ("", 68, -2).`

`Assert_failure : string * int * int -> exn`

`assert false` -> `Exception: Assert_failure ("", 69, -45).`

`End_of_file : exn, Division_by_zero : exn`

`Out_of_memory : exn, Stack_overflow : exn`

```
exception Fail of string
let head = function [] -> raise (Fail "head: the list is empty") | ...
let head_default def l = try head l with
  Fail _ -> def
```

Použití -- šetření pamětí

```
let rec remove x = function
    y :: l when x = y -> l
  | y :: l -> y :: remove x l
  | [] -> []

exception Unchanged
let remove x l =
  let rec remove' x = function
      y :: l when x = y -> l
    | y :: l -> y :: remove' x l
    | [] -> raise Unchanged
  in try remove' x l with Unchanged -> l
```

Použití -- breaky

```
let cat in_channel out_channel =
  try
    while true do
      output_char out_channel (input_char in_channel)
    done
  with End_of_file -> ()
```

Použití -- piggle

Výjimky mají 'dynamický' typ, takže můžeme mít seznam s dynamickými položkami

```
exception String of string
exception Int of int
exception Float of float
let incint l = l |> List.map (fun x -> match x with Int i -> Int (i+1) | _ -> x)
incint [ String "hello"; Int 2; Float 4. ]

  |-------|\
  |       | \
  |       | /
  |-------|/
   /\   /\
  /  \ /  \
```

Kompilační jednotky aneb moduly poprvé

```
set.ml
let empty = []
let add x l = x :: l
let mem x l = List.mem x l
let badfunc x l = ...

set.mli
type 'a set
val empty : 'a set
val add : 'a - 'a set -> 'a set
val mem : 'a -> 'a set -> bool
```

zde může být = 'a *list*

F# říká
– nelze skrýt type–synonym
– z union type je vidět vše nebo nic
– typ v fsi nemůže být méně polymorfní než ten v fs

```
open Set
```

Moduly v OCamlu

```
module Set = struct
  let empty = []
  let add x l = x :: l
  let mem x l = List.mem x l
end
```

```
module type SetSig = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> 'a set
  val mem : 'a -> 'a set -> bool
end
```

```
module Set : SetSig = struct
  type 'a set = 'a list
  let empty = []
  let add x l = x :: l
  let mem x l = List.mem x l
end
```

Modul může obsahovat definice typů, výjimek, letů, openy, includy, definice signatur a vnořených modulů.
Signatura může obsahovat definice typů, výjimek, valy, openy, includy, vnořené signatury.

Moduly nejsou first–class, dobrná výjimka je

```
let module M = module_expression in expression
```

Include pro novou funkcionalitu

```
module type ChooseSetSig = sig
  include SetSig
  val choose : 'a set -> 'a option
end
```

```
module ChooseSet : ChooseSetSig = struct
  include Set
  let choose = function x :: _ -> Some x
                      | []     -> None
end
```

Tohle nefunguje!

Nejjednodušší oprava jako

```
module SetInternal = struct
  ...
end

module Set : SetSig = SetIngernal
module ChooseSet : ChooseSetSig = struct include SetInternal ... end
```

Include pro změnu funkcionality

```
module type Set2Sig = sig
  ...
  val add : 'a set -> 'a -> 'a set
end

module Set2 : Set2Sig = struct
  include Set
  let add l x = Set.add x l
end
```

Ale teď jsou Set.set a Set2.set jiné typy, takže

```
Set2.add Set.empty 1;;      -> This expression has type 'a Set.set
                               but is used with type 'a Set2.set!
module Set2 : Set2Sig with type 'a set = 'a Set.set = struct
  ...
end
```