Value restriction

Přiřadit něčemu generický typ můžeme jenom, když:
♣ nemá žádné side−effekty
♣ produkuje to immutable hotnotu

Nicméně pro kompilátor je těžké až nemožné přesně zjistit, kdy jsou obě podmínky pro generický typ splněné. Proto se používá následující pravidlo: ke generalizaci typu dojde, pokud se jedná o syntaktickou hodnotu. Syntaktickou proto, ze je to poznat ze zapisu (ze syntakticke struktury). Pravidla jsou přibližně taková:
♣ literály jako 3, "tři", ...
♣ konstruktory aplikované na syntaktické hodnoty
♣ funkce. tj. `fun i -> i`
Všechny tyto konstrukce se zřejmě vyhodnotí bez side−effektů a jsou neměnitelné.

Speciálně na tomto seznamu není volání funkce, protože F# neví, jestli by tato vyprodukovala nebo nevyprodukovala syntaktickou hodnotu. Někdy opravdu může (`List.rev []`), někdy nemusí (*`ref []`*).

F# funguje tak, že se rozhodne, zda typ může nebo nemůže zobecnit. Pokud ano, je to typ `'a` a vše je v pořádku. Pokud ne, vytvoří typ `'_a`, který znamená "tenhle typ se nemůže zobecnit, musí být konkrétní". Pokud se mu ho podaří zkonkrétnit do konce kompilace modulu, je všechno v pořádku, jinak dojde k chybě "Value restriction".

Tedy zatímco samotný výraz
`let x = ref []`
se nezkompiluje, dvojice výrazů
`let x = ref []`
`x := 5 :: !x`
Už je v pořádku −− druhý řádek určil, že `x : int list ref`. Kdybychom přidali třetí řádek
`x := "šest" :: !x`, tak kompilace opět neuspěje, protože "šest" není *`int`*.

Jak se tedy vyhnout problémům s value restriction:
♣ Pokud vytváříme konstrukci, která nemá mít generický typ, musí být tento typ jednoznačně odvoditelný z následujících operací. Pokud není, musíme dodat typovou signaturu (*`ref [] : int list ref`*).
♣ Pokud vytváříme konstrukci, která má mít generický typ, tak
- dodáme explicitní volání argumentů, např. `let m = List.map id` změníme na `let m xs = List.map id xs`.
- dodáme explicitní typový parametr, např místo `let v = ref []` použijeme `let v<'a> : 'a list ref = ref []`. Pozor, tento kód nedělá to, co byste si na první pohled mysleli. Ve skutečnosti je nyní `v` funkce, která až dostane typ, vrátí odkaz na prázdný seznam. Ovšem pokaždé vrátí nový, takže `v<int> != 5; printf "%A" v<int>` vypíše `[]`.

Ohledně těch explicitních typových argumentů, uvažujme funkci `let empty = id []`. Tato hodnota vrací prázdný seznam, takže by mohla mít obecný typ, ale nevyhovuje pravidlům o syntaktické hodnotě. Pomocí `let empty<'a> : 'a list = id []` ji můžete zkompilovat a používat v kontextech, kde je typ známý (například `5 :: empty`). Ale samotné `empty` stále není syntaktická hodnota, takže nemůžeme napsat `let f = empty`. Můžeme ale použít dva atributy, které toto chování ovlivňují:
♣ `GeneralizableValue`: Pokud zadefinujeme `empty` jako
```
[<GeneralizableValue>]
let empty<'a> : 'a list = id []
```
tak `empty : 'a list` už je syntaktická hodnota a všechno funguje.
♣ `RequiresExplicitTypeArguments`: Opak `GeneralizableValue`. Pokud zadefinujeme
```
[<RequiresExplicitTypeArguments>]
let empty<'a> : 'a list = id []
```
tak `empty` nelze použít bez explicitního typového argumentu (čili jeho typ se vůbec negeneralizuje.

Funkce typeof

Funkce `typeof<'a> : System.Type` vrací reprezentaci daného typu, a to ve třídě `System.Type`. Má zmíněný atribut `RequiresExplicitTypeArguments`, takže je ji možné použít jenom se špičatými závorkami a konkrétním typem.

Reasociace pomocí funkcí

U `List.append` jsme řešili problém, že špatné uzávorkování může zhoršit složitost na kvadratickou. Jedna možnost, jak z toho ven, je reasociace pomocí funkcí.

Místo `append : list<'a> -> list<'a> -> list<'a>` vytvořme funkci
`prepend : list<'a> -> listF<'a> -> listF<'a>`
kde `type listF<'a> = list<'a> -> list<'a>`

Tedy `listF` není seznam, ale funkce, která vrací výsledek přilepený za daný seznam. Potom
```
let prepend xs ys = List.append xs << ys
```

Stejně tak foldy -- `fold` je v podstatě reasociovaný `foldBack` a naopak.

Tedy implementujeme-li `foldBack` pomocí `fold`, chceme udělat podobný trik jako u `append`: výsledek `fold`u bude funkce, která dostane počáteční stav a provede operaci `f` ve správném pořadí na prvky a daný stav.

```
let foldBack f xs s = List.fold (fun stav x z -> f x z |> stav) id xs s
let fold f s xs = List.foldBack (fun x stav z -> f z x |> stav) xs id s
```

Výjimky

Již od začátku byly v ML výjimky, vysoce výkonné. V F# jsou samozřejmě namapovány na .NET výjimky.

Každá funkce může vyhodit výjimku. Všechny výjimky stejně jako v .NETu dědí od `System.Exception`. Novou výjimku můžeme vytvořit jako:
```
exception OutOfBounds
```
nebo pokud chceme, aby nesla nejaka data, jako
```
type Index = int
exception OutOfBounds of Index
```

Vzniklá výjimka dědí od `System.Exception` a vyvolat ji můžeme pomocí
```
raise OutOfBounds
raise (OutOfBounds 5)
```
Funkce **`raise`** má typ `raise :: System.Exception -> 'a` Pozor na value restriction :-)

Zachytávat výjimky jde pomocí **`try ... with ...`**, syntaxe je:
```
try
  výraz
with
  | pattern1 -> handler1
  ...
  | patternN -> handerN
```
za **`with`** je klasický pattern matching, který se spouští s vyvolanou výjimkou, pokud byla vyvolána.

Při pattern matchování se dělá rozdíl mezi F# výjimkou a .NET výjimkou. Při pattern matchingu můžeme použít:
- `identifikátor` -- namatchuje libovolnou výjimku
- `jméno F# výjimky` -- namatchuje F# výjimku
- `:? .NET-výjimka` -- namatchuje .NET výjimku nebo jejího potomka (`:?` je test na předka)

Příklady:
```
let db_connect name =
  try
    sql_connect name
  with
    | e -> eprintf "Exception when connecting to database: %A" e

exception OutOfBounds of Index
let access_array a i =
  try ...
  with OutOfBounds i -> eprintf "The index %d is out of bounds!" e
let divide a b =
  try Some (a / b)
  with :? System.DivideByZeroException -> eprintf "Division by zero!\n"; None
let divide2 a b =
  try Some (a / b)
  with :? System.DivideByZeroException as e -> printfn "Exc %s" (e.Message);None
```

V handleru také můžete použít `reraise()`, které dál propaguje vyvolanou výjimku.

Také existuje konstrukce **`try ... finally ...`**:
```
try expression
finally cleanup
```
Výraz `cleanup : unit` je zde zavolán vždy, až již `expression` vyvolá nebo nevyvolá výjimku.

F# navíc nabízí několik standardních funkcí pro vyhazování výjimek:
`failwith : (msg:`*`string`*`) -> 'a` Výjimka `FailureException`.
`failwithf : (msg:`*`format`*`) -> args -> 'a` Výjimka `FailureException` s formátováním.
`invalidArg : (param:`*`string`*`)->(msg:`*`string`*`) -> 'a` Výjimka `System.ArgumentException`

IDisposable

Objekty v .NETu, které alokují zdroje systému, je uvolňují pomocí rozhraní `IDisposable`. F# obsahuje dvě možnosti, jak s nimi zjednodušit práci:

♣ **use**
funguje jako **let**, ale při odchodu proměnné ze scope (či vyvolání výjimky) zavolá `Dispose`, není-li objekt **null**
```
let writetofile filename obj =
   use file1 = File.CreateText(filename)
   file1.WriteLine(obj.ToString())
```
♣ **using**
funkce typu `using : 'a -> ('a->'b) -> 'b when 'a :> System.IDisposable`, která dostane objekt a funkci, ketrá ho zpracuje. Ať vyvolá výjimku nebo ne, objekt `'a` je poté `Dispose`-ován.
```
using (System.IO.File.CreateText("test.txt")) (printToFile "XYZ")
```

Rekurzivní definice a datové typy

Někdy bychom chtěli definovat rekurzivní datové typy. To jde například takto:
```
type Tree<'a> = Nil
              | Node of 'a * Forest<'a>
 and Forest<'a> = List<Tree<'a>>
```

Stejně tak můžeme chtít několik rekurzivních funkcí, která volá jedna druhou. To také lze pomocí:
```
let rec flatten_tree acc = function
        | Nil -> acc
        | Node (x, f) -> x :: flatten_forest acc f
    and flatten_forest acc = function
        | [] -> acc
        | f :: fs -> flatten_tree (flatten_forest acc fs) f
```

Líné seznamy, LazyList, z F# PowerPack

Pomocí líného vyhodnocování lze vytvořit i líné seznamy. Nicméně implementace `LazyList` z PowerPacku nepopužívá **lazy**, místo toho používá explicitní *`unit`* argumenty:
```
type LazyList<'T> =
    { mutable status : LazyCellStatus< 'T > }
    member x.Value = match x.status with
        | LazyCellStatus.Value v -> v
        | _ -> lock x (fun () ->
               match x.status with
               | LazyCellStatus.Delayed f ->
                   x.status <- Exception UndefinedException;
                   try
                       let res = f ()
                       x.status <- LazyCellStatus.Value res;
                       res
                   with e ->
                       x.status <- LazyCellStatus.Exception(e);
                       reraise()
               | LazyCellStatus.Value v -> v
               | LazyCellStatus.Exception e -> raise e)
and
    [<NoEquality; NoComparison>]
    LazyCellStatus<'T> = | Delayed of (unit -> LazyListCell<'T> )
                         | Value of LazyListCell<'T>
                         | Exception of System.Exception
and
    [<NoEquality; NoComparison>]
    LazyListCell<'T> = | CellCons of 'T * LazyList<'T>
                       | CellEmpty
```

Funguje jako normální *`list`*, ale seznam může být líný. Kromě klasických funkcí jsou v modulu `LazyList`:
`cons : 'T -> LazyList<'T> -> LazyList<'T>` klasický `(::)`
`consDelayed : 'T -> (unit->LazyList<'T>)->LazyList<'T>` jako `(::)`, ale `tail` je líně vyh.
`repeat : 'T -> LazyList<'T>` nekonečný seznam obsahující jeden prvek

`delayed : (`*`unit`*`->LazyList<'T>) -> LazyList<'T>` vytvoří seznam vyhodnocovaný líně

Navíc můžete v pattern matchingu použít `Nil` nebo `Cons` **`of`** `'T * LazyList<'T>`, tj. jako:

```
let rec sum acc = function | Nil -> acc
                           | Cons (x, xs) -> sum (acc + x) xs
```