Slozitost appendu, reasociace

Append má běžně složitost O(délka prvního seznamu). To by bylo v pořádku, kdyby se asocioval vždy doprava:
`x_1 @ (x_2 @ (x_3 @ ... @ x_N))...)`

Někdy to ale neovlivňujeme my, například při definici `flatten`. Pak se může stát, že vznikne
`((...(x_1 @ x_2) @ x_3 @ ... x_1`
a tento kód má kvadratickou složitost. Vyhnout se tom dá například použitím akumulátoru.

Jiná možnost jak se tomu vyhnout je všimnout si, že je `append` asociován na špatnou stranu a přeasociovat ho před použitím. Tak to dělá `Seq.append`.

Ruční iterace pres seq

Někdy funkce v modulu `Seq` nestačí a je potřeba sekvenci projít ručně. Samotné `IEnumeratory` mají hodně imperativní rozhraní. Pokud chceme psát raději funkcionálně, můžeme si zadefinovat například následující:

```
type Iterator<'a> = System.Collections.Generic.IEnumerator<'a>
let iterator (s:seq<'a>)      = s.GetEnumerator()
let value    (e:Iterator<'a>) = e.Current
let advance  (e:Iterator<'a>) = e.MoveNext() |> ignore; e
```

Toto rozhraní funguje dobře pro nekonečné sekvence, jenom musíme ručně volat `advance` před čtením prvního prvku. Můžeme vytvořit pokročilejší rozhraní:

```
type State = NotStarted | Inside | Ended
type Iterator<'a> = It of State ref * System.Collections.Generic.IEnumerator<'a>
let iterator (s:seq<'a>)          = It (ref NotStarted, s.GetEnumerator())
let init     (It (state, e) as it) =
  if !state = NotStarted then state := if e.MoveNext() then Inside else Ended
let advance  (It (state, e) as it) = match !state with
   | Ended -> it
   | _     -> if not (e.MoveNext()) then state := Ended
              it
let value    (It (state, e) as it) = init it; match !state with
   | Ended -> None
   | _     -> Some e.Current
```

Pak můžeme vytvořit sekvenci, která přeskočí každý druhý prvek, jako

```
let next it = match value it with
   | None   -> None
   | Some v -> Some (v, it |> advance |> advance)
let objedno s = Seq.unfold next (s |> iterator)
```

Mutable

Pomocí **`let mutable`** si můžeme vytvořit klasickou proměnnou:

```
let mutable x = 0
```

Zapisovat do ní můžeme pomocí operátoru <−, tj. `sum` můžeme napsat jako

```
let sum xs =
  let mutable s = 0
  for x in xs do s <- s + x
  s
```

Pozor na to, že **`let mutable`** je proměnná alokovaná na zásobníku, takže

```
let sum xs =
  let mutable s = 0
  Seq.iter (fun x -> s <- s + x) xs
  s
```

Nejde zkompilovat! Protože se tedy s **`mutable`** nedá často pracovat pěkně funkcionálně, budeme potřebovat ještě **`for`** a **`while`**.

♣ **`for`** je klasický for−cyklus, který se v F#pu vyskytuje ve dvou podobách:

- Ta jednodušší je **`for`** `proměnná = low` **`to`** `high` **`do`** `expr`,
  kde `low` a `high` jsou hodnoty typu *`int`* a `expr : `*`unit`*.
  ```
  for i = 1 to 10 do printf "%A\n" i
  ```
  Také existuje varianta **`for`** `proměnná = high` **`downto`** `low` **`do`** `expr`.

- Druhá varianta **for**–cyklu umí procházet cokoliv, co implementuje `IEnumerable<'a>`, tj. sekvence, seznamy, pole, a další. Syntaxe je `for proměnná in seznam do expr`, přičemž `expr : unit`.
  ```
  for x in {1..10} do printf "%A\n" x
  ```
  Podobné konstrukce se často dělají pomocí funkce `iter`, tj. `{1..10} |> Seq.iter (printf "%A\n")`.

♣ **while** je opět starý známý, syntaxe je `while podmínka do expr`, kde `podmínka:bool` a `expr:unit`.

```
  let e = s.GetEnumerator()
  while e.MoveNext() do
    printf "%A\n" e.Current
```

Jako **mutable** mohou být označeny i položky třídy nebo pojmenovaných struktur, tj.

```
type bod = { mutable x : float; mutable y : float }
```

Odkazy

Odkazy fungují jako pointery v jiných jazycích. Typ odkazu je `ref<'a>` od odkaz na typ `'a`.

Funkce pro práci s odkazy:

- `ref : 'a -> 'a ref` Vytvoří odkaz obsahující danou hodnotu
- `(!) : 'a ref -> 'a` Vrátí hodnotu, na kterou odkaz ukazuje
- `(:=) : 'a ref -> 'a -> unit` Přiřadí do odkazu jinou hodnotu
- `incr, decr : int ref -> unit` Zvýší či sníži hodnotu v odkazu na *int*

Funkci `sum` jsme tedy mohli napsat jako

```
let sum xs =
  let s = ref 0
  for x in xs do s := !s + x
  !s
```

Odkazy mj. řeší problémy s nemožností předávat **let mutable** do funkce, tj. můžeme napsat

```
let sum xs =
  let s = ref 0
  Seq.iter (fun x -> s := !s + x) xs
  !s
```

Odkazy nejsou nic jiného, než

```
type Ref<'a> = { mutable contents : 'a }
```

Tato definice je viditelná, takže můžete psát
`{ contents = 5 }` místo `ref 5`
`x.contents <- 5` místo `x := 5`

Pole

Pole typu `'a []` je klasické .NETí pole. Je to pole neměnné velikosti, update a čtení položky v konstantním čase, spodní index je nula.

Pole můžete vytvořit jako `[|1; 2; 3|]` nebo jako `[|0..10|]`.

Přístup k prvkům je `x.[4]`, zápis do prvků `x.[4] <- 5`. Všimněte si té tečky, která se jinde většinou nepíše. Můžete také přistoupit k celému úseku pole pomocí `x.[5..10]`. Tento výraz vrátí deset prvků počínaje šestým daného pole, a to jako pole nové.

V modulu `Array` se nachází mnoho funkcí pro práci s poli. Většinu jich budete znát z modulu `List` a `Seq`. Jenom složitosti mohou být trochu jiné, například `Array.append` má složitost O(délka obou polí) místo O(délka prvního seznamu) jako v `List.append`.

Zde následuje seznam všech funkcí:

```
append, average, averageBy, blit, choose, collect, concat, copy, create, empty,
exists, exists2, fill, filter, find, findIndex, fold, fold2, foldBack,
foldBack2, forall, forall2, get, init, isEmpty, iter, iter2, iteri, iteri2,
length, map, map2, mapi, mapi2, max, maxBy, min, minBy, ofList, ofSeq,
partition, permute, pick, reduce, reduceBack, rev, scan, scanBack, set, sort,
sortBy, sortInPlace, sortInPlaceBy, sortInPlaceWith, sortWith, sub, sum, sumBy,
toList, toSeq, tryFind, tryFindIndex, tryPick, unzip, unzip3, zeroCreate, zip,
zip3
```

Pole můžeme vytvořit jako:
- ♣ literál `[|1; 2; 3|]`
- ♣ `Array.empty<'T> : 'T []`
- ♣ `Array.init : int -> (int -> 'T) -> 'T []`
- ♣ `Array.create : int -> 'T -> 'T []`
- ♣ `Array.zeroCreate : int -> 'T []`

Přirozeně délka pole lze zjistit v konstantním čase pomocí `Array.length` nebo `[|1|].Length`.
Metody `foldBack` a `reduce` už nemusí alokovat pomocnou paměť velikosti O(N).

Seznam neznámých funkcí v modulu `Array`:

`blit : 'T [] -> int -> 'T [] -> int -> int -> unit`
  Kopíruje část jednoho pole do druhého:
  `Array.blit source sourceIndex target targetIndex count`

`copy : 'T [] -> 'T []`
  Vytvoří kopii daného pole

`fill : 'T [] -> int -> int -> 'T -> unit`
  Vyplní úsek od daného indexu dané délky daným prvkem.

`get : 'T [] -> int -> 'T`
  Přečte prvek na daném indexu

`set : 'T [] -> int -> 'T -> unit`
  Zapíše prvek na daném indexu

`sortInPlace : 'T [] -> unit`
  Jako `sort`, ale provede se přímo v daném poli

`sortInPlaceBy : ('T -> 'Key) -> 'T [] -> unit`
  Jako `sortBy`, ale provede se přímo v daném poli

`sortInPlaceWith : ('T -> 'T -> int) -> 'T [] -> unit`
  Jako `sortWith`, ale provede se přímo v daném poli

`sub : 'T [] -> int -> int -> 'T []`
  Vrátí úsek od daného indexu dané délky jako nové pole, jako `pole.[from..len]`

F# obsahuje i více rozměrné pole, dvou až čtyřrozměrné: `'a [,]`, `'a [,,]`, `'a [,,,]`
Nemůžete už vytvořit literál pro dvourozměrné pole, musíte použít funkce modulů `Array[234]`.
Vícerozměrná pole navíc nemusí mít dolní index nulu. Některé funkce, ostatní viz dokumentace:

```
Array[234].{length1, length2, length3, length4, create, zeroCreate,init,get,set}
Array[23].{iter, iteri, map, mapi}
array2D :: seq<#seq<'T>> -> 'T [,]
```

Nafukovací pole

`ResizeArray<'a>` je typové synonymum pro `System.Collections.Generic.List<'a>`.

To je pole, které má proměnlivou velikost, přidávat hodnoty na konec umí amortizovaně za konstantu a odebírat hodnoty z konce za konstantu. Kromě toho umí indexovat a měnit hodnoty v konstantním čase, se stejnou syntaxí jako `Array`.

Typ ResizeArray<'a> `má několik členských funkcí, ty nejdůležitější jsou:*)`

`member this.Add : 'a -> unit`	Přidá prvek na konec
`member this.AddRange : seq<'a> -> unit`	Přidá na konec danou sekvenci
`member this.Count : int`	Počet prvků
`member this.RemoveAt : int -> unit`	Odebere prvek s daným indexem. Je–li poslední, O(1)

```
let p = new ResizeArray<int>();
for x = 1 to 10 do p.Add x;
```

Pro práci lze použít i modul `ResizeArray`, který obsahuje skoro ty samé funkce jako `Array`. Jenom pozor na to, že tento modul už není v základním F#pu, je jenom v F# PowerPacku.

Value restriction

Všimli jste si podezřelé práce s prázdným seznamem?

```
List.rev []
(* stdin(1,1): error FS0030: Value restriction. The value 'it' has been inferred
to have generic type
  val it : '_a list
Either define 'it' as a simple data term, make it a function with explicit
arguments or if you do not intend for it to be generic, add a type annotation.*)
```

Všimli jste si toho zvláštního typu `list<'_a>`? Takový podtržítkový typ jsme ještě neviděli.

Nebo například
```
let m = List.map            (* OK *)
let m f = List.map f        (* OK *)
let m = List.map id         (* Chyba *)
```

Problém je ve automatické generalizaci typů. F# se snaží používat co nejvíc gerické typy, např. `let id x = x` má typ `'a -> 'a` a `[] : list<'a>`.

Ale ne vždy můžete použít nejvíce generický typ –– kdyby `ref [] : ref<list<'a>>`, tak potom by bylo možné udělat `let x = ref []; x := 5 :: !x; x := "šest" :: !x`, což je určitě špatně.

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

Problém s `ref []` je přesně v tom, že vrací hodnotu, která se může měnit –– musí to tedy být hodnota jednoho konkrétního typu, ne hodnota libovolného typu!

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, protoře "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.