Neprocedurální programování.

Funkcionální programování. Haskell Př. 7–12

Jan Hric

30. března 2025

Obsah

Outline

Funkcionální programování

- Programování s pomocí funkcí
- program: definice funkcí, jako přepisovacích pravidel
- výpočet: opakované nahrazení volání funkce tělem fce
  - β-pravidlo: $(\lambda x.M)N \rightarrow_\beta M[x := N]$ (”beta”, ”lambda”)
- v Hs (čistý jazyk): ”matematické” funkce, bez vedlejších efektů
- datové struktury: λ-termy (λ-abstrakce, aplikace) + vest. a uživ.
- argumenty i výsledky funkcí můžou být funkce i složené dat. struktury
- výsledek: vyhodnocený tvar, normální forma (pokud existuje)
  - vyhodnocený tvar je jednoznačný (i díky teorii)
  - vyhodnocování je deterministické (na rozdíl od Prologu)
- dnešní jazyky integrují části FP (nějak), např. λ-funkce
- z hlediska logiky: rovnostní teorie (pouze predikát rovnosti)

Příklad, Robinsonova aritmetika (část)

```
data Nat = Z | S Nat     -- def. typu
x + Z     = x          -- x+0 = x; Z jako Zero
x + (S y) = S (x+y)    -- x+S(y)=S(x+y)
--
x+y = if y==Z then x else S(x+ unS y)
  where unS (S y) = y
```

- Haskell, definice funkce $+$ (pro numerály)
- case-sensitive (Konstruktory termu vs. funkce), operátory
- složené výrazy (tj. termy, ale jinak se píšou), i jako návratová hodnota fce.
- „=“ : levou stranu přepisujeme pravou stranou
- `S, Z`: ”vyhodnocené” funkce - (datové) konstruktory
- `(S Z)+(S(S Z))` -> `S((S Z)+(S Z))` -> `S(S((S Z)+Z))` -> `S(S(S Z))`

Historie FP a příklad použití

- λ-kalkulus: 30. léta (teorie vyčíslitelnosti)
- Lisp 1959: s-výrazy
- Scheme 1975
- ML 1985: typovaný jazyk
- Haskell 1989
- Haskell 2010
- ...a další: Erlang (pro realtime apl.), Scala (kompiluje do JVM), Clojure (JVM), OCaML, F♯

- FFTW: knihovna FFT in the West - cena za numerickou matematiku 1999. V čem je napsána: no, (samozřejmě) v C/C++. Ale generovaném pomocí OCaML.

Haskell

- https://www.haskell.org ; ekosystém
- kompilátor GHC: Glasgow Haskell Compiler (GHCi, WinGHCi IDE)
- HUGs (WinHugs)

Literatura

- Graham Hutton, Programming in Haskell, Cambridge University Press 2007, Cambridge, Velká Británie
- Richard Bird, Thinking Functionally with Haskell, Cambridge University Press 2014, Cambridge, Velká Británie

- Bryan O'Sullivan, Don Stewart, John Goerzen, Real World Haskell, O'Reilly Media 2008, `http://book.realworldhaskell.org/`
- `http://learnyouahaskell.com/chapters`, online

Charakteristiky Haskellu

- Silný statický typový systém (parametrický polymorfizmus, typové odvozování)
- Rekurzivní funkce, funkce vyšších řádů (f. jako param. i výsl.)
- Uživatelsky definované typy, rekurzivní, parametrické (stromy ...)
- ”Stručné” seznamy (list comprehensions)
- Líné vyhodnocování
- Čistý funkc. jazyk, referenční transparentnost, neměnné (immutable) dat.strukt. - proměnné označují hodnoty
  - pro odvozování a dokazování vlastností programů
- Monády, i pro V/V
- Operátory, přetěžování (pomocí typových tříd), ...
- ... moduly, balíčky, paralelizmus, kompilace

Základy práce

- REPL: Read-Eval-Print Loop, interpretační prostředí
- - lze i kompilovat (do .exe)
- program ve skriptu, obvyklá přípona .hs
- - příprava skriptu v textovém editoru
- `-- jednořádkové komentáře`
- `{- vnořené víceřádkové komentáře -}`
- `Prelude.hs` - standardně natahovaný soubor, obsahuje předdefinované funkce

Prostředí

- prompt `>`
- `:quit` : ukončí prostředí
- `:?` : help
- `:load "myfile.hs"` : načtení souboru
- `:type map` : vypíše typ výrazu, čtyřtečka `::`
  výraz `::` *má typ*
  ```
  map :: (a -> b) -> [a] -> [b]
  ```
- `:set +t` : nastavení options, např. výpis typu

Hodnoty a typy

Příklady zápisu hodnot a jejich typů

- `5 :: Int` – celá čísla
- `100000000000 :: Integer` – dlouhá čísla
- `3.0 :: Float` nebo `3.0 :: Double`
- `'a' :: Char '\t', '\n', '\\', '\'',` a `"\""`
- `True :: Bool, False :: Bool` – v prelude
- `[1,2,3] :: [Int]`, totéž `1:(2:(3:[])) :: [Int]`
- `"abc":: String` – řetězce, String = [Char]
  - *špatně: `[2,'b'] :: [?]` – kompilátor odmítne
- `(2,'b') :: (Int,Char)` – dvojice, n-tice, i `() :: ()`
- `succ :: Int -> Int` – typ funkce, nepovinný při def. fce
- `succ n = n+1` – definice funkce

Typy

- Každý výraz má typ; nemusíme uvádět (typy funkcí), systém si typy (většinou) odvodí.
- `> ((read "5")::  Float) + 3`
- Konkrétní typy začínají velkým písmenem, typové prom. malým.
- Nekonzistentní typy: typová chyba, při překladu (tj. při `:load`)
- Haskell nemá implicitní přetypování, nutno explicitně
  `fromInteger :: Num a => Integer -> a`
- Motivace pro zavedení typů:
  - dokumentace
  - ochrana proti některým druhům chyb
  - vložení/vygenerování pomocného kódu
  - Ale: typy (v Hs) neodchytí speciální sémantiku hodnot:
    `1/0, head []`

Příklady (i se seznamy)

```
-- elem :: (..) => a -> [a] -> Bool  -- typ, funkce
-- je <arg1> prvek seznamu <arg2>?
elem x []     = False
elem x (y:ys) = x==y || elem x ys  -- vs. Prolog

rev :: [a] -> [a]
rev xs       = rev1 xs []   --  akumulátor: 2. arg.
rev1 xs acc = if null xs then acc
                else rev1 (tail xs) (head xs :acc)

expR x e = --  rychlé umocňování x^e
  if e == 0 then 1 else
  if e == 1 then x else
  if even e then expR (x*x) (div e 2)
    else x*expR x (e - 1)
```

Funkce

- Definice v souboru, tj. skriptu; nutno natáhnout `:load/:reload`
- Funkce se aplikuje na *argumenty* a vrací *výsledek* (případně složený); aplikace je ”selektor”
- Vyhodnocování je deterministické: LS první vhodné klauzule (při pattern matchingu), LS první pravdivé stráže
- Jména (alfanumerických) funkcí a proměnných začínají malým písmenem (datové konstruktory velkým)
- Definice funkce je ”laciná”
  $\rightarrow$ mnoho krátkých funkcí, které skládáme a specializujeme
  $\rightarrow$ funkce píšeme tak, aby šly skládat a specializovat
- Currifikovaná forma (curried, podle Haskell B. Curry)
  - ”argumenty se aplikují po jednom” - ale budeme mluvit o funkcích více proměnných
  - funkce dostane jeden argument a vrací funkci ve zbylých argumentech, $f(x, y) = (f(x))y = f_x(y)$

Funkce 2

- `f :: Int -> Char -> Bool`
- funkční typ `->` je asociativní doprava:
  `f::Int->(Char->Bool)`
- necurrifikovaná funkce `f'::(Int,Char)->Bool` volaná na dvojici, která nedovoluje částečnou aplikaci
- `> f 3 'a'` ; volání funkce, také `f 3`
- volání/aplikace je asociativní doleva: `(f 3) 'a'`, místo necurrifikované `f' (3,'a')`
- typování částečných aplikací sedí: `((f 3)::(Char->Bool)) ('a'::Char)`
- konkrétní výskyt fce `f` v programu může mít stejně, míň nebo víc arg. vůči definici (jediná def. `f` s určitou četností/typem, na rozdíl od Prologu)
  → syntakticky, tj. závorkami, určíme aktuální počet arg.

Funkce 3

- Složené argumenty funkce jsou v závorkách.
- typové odvozování unifikací: z `f::b->c` a `x::b` odvoď `f x::c`
- typicky, výsledek funkce, tj. hodnota, se hned použije jako argument, případně *pojmenuje* lokálním jménem (tj. nepřiřazuje se, nemáme příkazy);
  proměnné reprezentují hodnoty (i složené), ne místa v paměti

.

Stručně vestavěné funkce

... stručně a zjednodušeně

- `Int`: typ celých čísel, `Integer`: dlouhá čísla
- běžné aritmetické funkce:
- `+, *, -, div, mod :: Int -> Int -> Int` – zj.
- `abs, negate :: Int -> Int`
- formálně: `(+) :: Num a => a -> a -> a`
- typ `Bool`, výsledky podmínek, stráží
- `== /= > >= <= < :: (..) => a -> a -> Bool` – zj.
- `(&&), (||) :: Bool -> Bool -> Bool` – binární and a or
- `not :: Bool -> Bool`

Syntax - layout

- V .hs souborech definice globálních funkcí začínají v 1. sloupci
- 2D layout, offside pravidlo: definice ze stejné skupiny začínají ve stejném sloupci
  - co začíná víc vpravo, patří do stejné definice jako minulý řádek
  - co začíná víc vlevo, ukončuje skupinu definic
  - aplikuje se na lokální definice (let, where), stráže, case
  - umožňuje vynechání oddelovačů ”;” a závorek ”{}”
- Víc definic na jednom řádku:
  `let {x=1;y=2} in ...`

Definice funkcí: if, stráže

- `even x = mod x 2 == 0` – jednořádková
- `abs1 x = if x>=0 then x else -x` – if-then-else
  - podmínka je výraz typu `Bool`
  - i.t.e. má vždy `else` větev: co vracíte, když selže podmínka
- stráže (svislítko): výpočet ve FP je deterministický, bere se první výraz za rovnítkem, u kterého uspěje stráž `:: Bool`; `otherwise` ve významu `True`

```
nsd n m
  | m == 0     = n
  | n >= m     = nsd m (mod n m)
  | otherwise = nsd m n
abs2 x = (if x>=0 then id else \y-> -y) x
id = \x -> x -- id x = x {; anonymní funkce}
```

Porovnávání se vzorem 1

- angl.: *pattern matching*, obvyklý způsob def. fce
- Vyhodnocovaný výraz ve FP je bez proměnných (na rozdíl od Prologu), ale ne nutně vyhodnocený. V def. funkce na místě formálních parametrů může být nejen proměnná, ale i term, který vyjadřuje implicitní podmínku na tvar dat (po nezbytném vyhodnocení). Př. `ordered (x1:x2:xs) =...`
- Aktuální parametry se musí dostatečně vyhodnotit, aby šla podmínka rozhodnout. Vyhodnotí se pouze potřebné části struktury (díky línému vyh.), u složených typů. Jednoduché typy se vyhodnotí na konstantu.

```
length1 :: [a] -> Int
length1 []     = 0
length1 (x:xs) = 1+length1 xs
```

- `length1` je parametricky polymorfní (`[a]->...`), přijímá seznamy s lib. prvky `:: a` ; zpracovává pouze strukturu seznamu, nikoli prvků (!typické pro FP)

Porovnávání se vzorem 2

- P.m. lze použít i pro uživatelsky definované typy.
- P.m. pojmenuje složky aktuálních argumentů, proto (obvykle) nepotřebujeme selektory (resp. rozebírání struktury, např. `head`, `tail`)
- Jméno proměnné ve vzoru je definiční výskyt (může být jen jednou, na rozdíl od Prologu), prom. porovnáváme `(==)`
- Neodmítnutelný vzor `"_"` (podtržítko): uspěje, nevyhodnocuje arg., může být víckrát

```
and1 :: Bool -> Bool -> Bool
and1 False _ = False
and1 _     x = x  -- x  se bude vyhodnocovat až v kontextu
and2 True True = True
and2 _    _    = False
```

- - and1 je líné v 2. arg., and2 vyhodnocuje někdy i 2. arg.

Porovnávání se vzorem 3

vzor `@`: přístup na celek i části

```
-- ordered :: [a] -> Bool
ordered (x:xs@(y:_)) = x<=y && ordered xs
ordered _            = True --  default, záchytná podm.
```

- matching `@` neselže, ale matching podčástí může selhat
- default s _ v příkladu se použije pouze pro `[x]` a `[]`. Použití koncové klauzule na 2. místě nevadí (ve FP, na rozdíl od Prologu) pro LCO/TRO.
- implementačně: použití `@` ušetří novou stavbu struktury v těle funkce (zde konstrukce seznamu pomocí `:`)
- složené vzory musí být v závorkách, jinak se špatně naparsují (Př. sémantické chyby: `length x:xs =...`)

Seznamy

- - jsou vestavěné, jako speciální syntax
- `data [a] = [] | a : [a]` – pseudokód
- Konstruktory (odvozené z typu):
- `[] :: [a]` – polymorfní konstanta
- `(:) :: a -> [a] -> [a]` – datový konstruktor
- syntax: `[1,2,3]` je `1:2:3:[]` asoc. doprava
  tj. `1:(2:(3:[]))`
- *nelze `[1,2:xs]`, nutno `1:2:xs` pro seznamy s tělem `xs`
- hranaté (seznamové) závorky se používají pro další dva konstrukty - příště (stručné seznamy a generátory posl.)

map, filter, reverse

`map` a `filter` jsou funkce vyššího řádu

```
map :: (a->b) -> [a] -> [b]
map _ []     = []  --  strukturální rekurze podle 2. arg.
map f (x:xs) = f x : map f xs
filter (a->Bool)->[a]->[a]
filter p [] = []
filter p (x:xs) = if p x then x:filter p xs
                         else   filter p xs
reverse xs = rev1 xs []  --  rychlé, pomocí akumulátoru
rev1 [] acc     = acc
rev1 (x:xs) acc = rev1 xs (x:acc)
```

`> map (map (\x->c*x)) [[1,2],[3,4]]` – částečná aplikace
`zip xs ys = zipWith (\x y->(x,y)) xs ys` – specializace f.

- DC: `zip :: [a]->[b]->[(a,b)]`
- DC: `zipWith :: (a->b->c)->[a]->[b]->[c]`

Lokální definice: let (a where)

- `let` tvoří výraz, možno vkládat do výrazů
- Definice v `let` (a `where`) jsou vzájemně rekurzivní, můžeme def. hodnoty i (lokální) funkce
- V `let` vlevo lze použít pattern matching (pro dvojice a pod.)

```
let (zprava1,v1) = faze1 x
    (zprava2,v2) = faze2 v1
    (zprava3,v ) = faze3 v2
    zprava = zprava1++zprava2++zprava3
 in (zprava,v)
```

- typické použití `let`: lokální zapamatování hodnoty, když ji nebo její části potřebujeme víckrát použít
- `++` je append, spojení seznamů

Konstrukt where

- Speciální syntax, netvoří výraz; lze použít jen na vnější úrovni definice funkcí
- Definice ve where jsou vzájemně rekurzivní, můžeme def. hodnoty i funkce
- `qs/2`: quicksort, s parametrickým komparátorem `cmp/2` (!typické pro FP `:-)`)
- `f`: `where` přes několik stráží, nelze pomocí `let`

```
qs _cmp [] = []
qs  cmp (x:xs) = qs cmp l ++ (x:qs cmp u)
  where
    (l,u) = split cmp x xs
f x y
  | y >z = ...
  | y==z = ...
  | y <z = ...
    where z = x*x
```

Lexikální konvence 1

Alfanumerické identifikátory:

- posloupnosti písmen, číslic, ′ (apostrofu), _ (podtržítka)
- jména funkcí a proměnných: začínají malým písmenem nebo podtržítkem
- (velkým písmenem začínají konstruktory: True, Bool)
- vyhrazeno: klíčová slova: case of where let in if then else data type infix infixl infixr primitive class instance module default ...
- funkce se zapisují v prefixním zápisu: `mod 5 2`
- alfanumerický identifikátor jako (binární infixní) operátor, uzavření v ‘(zpětný apostrof): `5 ` mod ` 2`

Lexikální konvence 2

Symbolové identifikátory:

- jeden nebo víc znaků: `: ! * + - . < = > @ ^ | / \ &`
- speciální význam: `: ~`
- symbolové konstruktory začínají znakem `:`
- standardní zápis: jako infixní operátor: `3+4`
- pro zápis v prefixním nebo neoperátorovém kontextu: v závorkách: `(+) 3 4, map (+) [5,6], (+) :: ...`
- sekce (*op*), (*op* arg), (arg *op*) jako částečně aplikované funkce, pro oba druhy identifikátorů:

```
(+) = \x y -> x+y
(1/) =  \y -> 1/y
(`div`2) = \x -> x`div`2
```

Literate Haskell

Je to varianta zdrojové syntaxe
Obvyklá přípona je .lhs
Za platný kód jsou považovány pouze řádky začínající znakem `'>'`,
všechno ostatní je komentář
Okolo řádků kódu musí být prázdné řádky
Příklad funkce `(++)`, tj. append

```
> (++) :: [a] -> [a] -> [a]
> []     ++ ys = ys
> (x:xs) ++ ys = x:(xs++ys)
```

Obvyklé použití: soubor jiného určení (blog v HTML, TeX) je také platný Literate Haskell (po změně přípony na .lhs)

Standardní funkce z prelude 1

```
head :: [a] -> a
head (x:_) = x -- pro [] chyba
tail :: [a]->[a]
tail (_:xs) = xs -- pro [] chyba
null :: [a] -> Bool
null xs = xs==[] --
```
nepoužívat `if :-(` , ale bool. spojky
```
id :: a->a
elem :: Eq a => a -> [a] -> Bool --
```
relace jako charakteristická fce
```
elem x [] = False
elem x (y:ys) = x==y || elem x ys
(!!):: [a] -> Int -> a -- n-tý od 0
fst :: (a,b) -> a
snd :: (a,b) -> b
(,) :: a -> b -> (a,b) -- spec. syntax
```

Standardní funkce 2

- take n xs - vrátí prvních n prvků z xs nebo všechny, když je jich málo
- chceme fci rozumně dodefinovat (pro `length xs < n`)
- ukázka použití selektorů :-((`head`, `tail`) a stráží :-)

```
take :: Int -> [a] -> [a]
take n xs =
  | n<=0 || xs==[] = []
  | otherwise = head xs: take (n-1) (tail xs)
```

- drop n xs - stejný typ, zahodí prvních n prvků a vrátí zbytek (i `[]`)
- takeWhile p xs - vrátí nejdelší úvodní úsek, kde pro prvky platí podmínka p; porovnejte typy take a takeWhile

```
takeWhile :: (a->Bool)->[a]->[a]
takeWhile p [] = []
takeWhile p (x:xs) =
  if p x then x:takeWhile p xs
         else []
```

Standardní funkce 3

- map, filter, (bude: foldr, unfold)
- zipWith f xs ys - paralelní zpracování 2 seznamů xs a ys fcí f

- `zipWith/3` je polymorfní, protože prvky vstupních seznamů zpracovává pouze funkcionální parametr `f/2`. (!typické pro FP)

```
zipWith :: (a->b->c)->[a]->[b]->[c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _      _      = []  -- jeden ze seznamů skončil

zip :: [a] -> [b] -> [(a,b)]
zip = zipWith (,)
-- totéž:
zip xs ys = zipWith (,) xs ys  -- stejné posl. arg. lze vypustit
zip xs ys = zipWith (\x y->(x,y)) xs ys
```

- zip - jednořádková definice získaná specializací, !typické pro FP

Generátory posloupností

- speciální syntax pro generování aritmetických posloupností, dvě tečky
- převádí se na volání funkcí z typové třídy Enum → jde použít i pro uživatelské typy
- jiné posloupnosti než aritmetické získáme transformací (např. map)
- první dva členy určují ”krok”, generuje i nekonečné seznamy

- `[1..5] ~> [1,2,3,4,5]`
- `[1,3..10] ~> [1,3,5,7,9]`
- `[1..] ~> [1,2,3,4,5 ...`
- `[1,3..] ~> [1,3,5,7,9,11 ...`
- `[5,4..1] ~> [5,4,3,2,1]`

Stručné seznamy 1

- list comprehensions
- motivace z teorie množin: $\{x^2 - y^2 | x, y \in \{1..9\} \wedge x > y\}$
- vybíráme prvky ze seznamů a pro každý úspěšný zkombinovaný výběr generujeme 1 prvek výstupního seznamu
- před svislítkem: hodnota, která se vygeneruje na výstup
- za svislítkem, oddělené čárkami:

1. generátory `x<-xs`, vyhodnocované zleva
2. testy/stráže `::` `Bool`,
3. `let mez = 100`

```
[f x|x<-xs] -- map f xs, generátor x<-xs
[x|x<-xs, p x] -- filter p xs, test p x
[(x,kv)|x<-xs, let kv=x*x] -- let
```

Stručné seznamy 2: příklady

```
kart :: [a] -> [b] -> [(a,b)]
kart xs ys = [(x,y)|x<-xs, y<-ys]
concat :: [[a]] -> [a]
concat xss = [x|xs<-xss, x<-xs]
length xs = sum [1| _<-xs] --  nepoužijeme hodnotu
klice pary = [klic|(klic,hodn)<-pary] -- pattern matching
delitele n = [d| d<-[1..n], n`mod`d==0]
prvocislo n = delitele n == [1,n]
horniTrojuh n=[(i,j)|i<-[1..n],j<-[i..n]] -- použijeme i

> kart [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

Q: kolik dělitelů se vygeneruje, aby se zjistilo, že n není prvočíslo?

Příklad: quicksort

```
qs :: Ord a => [a] -> [a]
qs []     = []
qs (x:xs) = qs [y|y<-xs,y<x]++[x]++qs [y|y<-xs,y>=x]

qs1 :: (a->a->Bool)->[a]->[a]
qs1 cmp xs = qs xs
  where qs [] = []
        qs (x:xs) = qs [y|y<-xs, y`cmp`x]++[x]++
                    qs [y|y<-xs, not(y`cmp`x)]
> qs1 (<) [9,7..0]
```

- qs1 parametrizovaný uspořádáním cmp, obvyklé v FP: funkce píšeme co nejobecněji (funkc. param. jsou nejobecnější způsob předání arg.)
  → qs1 je polymorfní, univerzální (třídí i reverzně, podle 1./2. složky ...)
- relace reprezentována jako charakteristická fce `:: ... -> Bool`
- ”schovaný” parametr cmp v rekurzi

Seznamy výsledků

- Programátorský idiom, nemáme backtracking (na rozdíl od Prologu), pracujeme se všemi výsledky v seznamu a vydáváme jeden seznam všech výsledků
- díky línému vyhodnocování se seznam (při dobré implementaci) postupně generuje a zpracovává, tj. nemáme v paměti celou strukturu najednou. Př. `length(komb 6 [1..11])`
- Př: kombinace, tj. seznam všech kombinací

```
komb 0 ps = [[]]
komb _ [] = []
komb k (p:ps) = [p:k1|k1<-komb (k-1) ps]++komb k ps
```

- Př: vrácení prvku a zbytku seznamu všemi způsoby (bez ==)

```
vyber :: [a] -> [(a,[a])]
vyber [] = []
vyber (x:xs) = (x,xs):[(y,x:ys)|(y,ys)<-vyber xs]
> vyber [1,2,3]
[(1,[2,3]),(2,[1,3]),(3,[1,2])]
```

Generuj a testuj

- Programy typu ”generuj a testuj” se lehce *píšou* (včetně NP-úplných), někdy s použitím stručných seznamů
- výstup může být seznam výsledků, případně zpracovaný (`minimum`), existence výsledku ...
- Př: přesný součet podmnožiny (mírně optimalizováno): `batoh1 b xs` platí, pokud existuje $l \subset xs$, tž. $\sum_{x \in l} x = b$.

```
batoh1 :: Int -> [Int] -> Bool
batoh1 0 _  = True
batoh1 _ [] = False
batoh1 b (x:xs) =b>0 &&(batoh1 (b-x)xs ||batoh1 b xs)

batoh2 :: (Num a, Ord a) => a -> [a] -> [a] -> [[a]]
batoh2 0 _      acc = [acc]
batoh2 _ []     acc = []
batoh2 b (x:xs) acc = if b<=0 then []
   else batoh2 (b-x) xs (x:acc) ++ batoh2 b xs acc
```

Kartézský součin víc seznamů

- Trade-off čas vs. paměť

```
kartn1, kartn2 :: [[a]] -> [[a]]
kartn1 [] = [[]]
kartn1 (xs:xss) = [x:ks|x<-xs, ks<-kartn1 xss]
kartn2 [] = [[]]
kartn2 (xs:xss) = [x:ks|ks<-kartn2 xss, x<-xs]
kartn3 [] = [[]]
kartn3 (xs:xss) = [x:ks|let kk=kartn3 xss,x<-xs,ks<-kk]
> kartn1 [[1,2],[3],[5,6]]
[[1,3,5],[1,3,6],[2,3,5],[2,3,6]]
```

- Chování: kartn1 generuje (stejné) ks opakovaně, jako při prohledávání stavového prostoru do hloubky.
- Fce kartn2 vygeneruje *velkou* d.s. obsahující ks jednou. Pokud si ji potřebuje pamatovat, tak je to (až neúnosně) paměťově náročné.
- Fce kartn3 generuje d.s. kk jednou a pamatuje si ji.

Operátory, deklarace

- Operátory jsou syntaktický cukr, pro přehlednost a pohodlí zápisu
- Haskell má pouze binární operátory, priority 9-0 (0 nejnižší)
- Vyšší priorita váže víc, jako v matematice
- Aplikace funkce váže nejvíc, proto musí být složené argumenty v závorkách (i při porovnávání se vzorem)
- př.: `fact 3+4` vs. `fact(3+4)`
- Lze použít i pro alfanumerické operátory

```
infixl 6 +      --sčítání, doleva asoc.
infixr 5 ++      --append, doprava asoc.
infix 4 ==      --porovnávání, neasoc.
infix 4 `elem`      --prvek seznamu
```

Operátory

- funkční volání váže těsněji než nejvyšší priorita 9
- 9,doleva `!!` – n-tý prvek, `(mat!!1)!!2`
- 9,doprava `.` – tečka, skládání funkcí
- 8,doprava `^ ^^ **`
- 7,doleva `* / 'div' 'mod' 'rem' 'quot'`
- 6,doleva `+ -` – i unární -
- 5,doprava `: ++` – `1:(2:[])`
- 5,neasoc `\\` – delete
- 4,neasoc `== /= < <= > >= 'elem' 'notElem'`
- 3,doprava `&&` – doprava vhodnější pro líné vyh.
- 2,doprava `||`

Na asociativite záleží, pokud jsou arg. různého typu (`!!` `:` `elem`)

Uživatelské typy, nerekurzivní

• Klíčové slovo `data`, vlevo od = typ (s parametry), vpravo datové konstruktory (s parametry), jednotlivé varianty oddělené svislítkem `|`

```
data Bool = False | True -- z prelude
data Ordering = LT | EQ | GT
data Point a = Pt a a  -- polymorfní
data Maybe a = Nothing | Just a -- pro chyby
data Either a b = Left a | Right b -- sjednocení typů
```

• Bool, Ordering, Point ... jsou jména typových konstruktorů
• False, LT, Pt, ... jsou jména datových konstruktorů
• Vyhodnocený tvar výrazu obsahuje (pouze) datové konstruktory, vyhodnocené datové konstruktory můžou obsahovat nevyhodnocené argumenty
• Datové konstruktory jsou funkce (odvozené z typu) a mají svůj (i polymorfní) typ: `True::Bool, Just :: a -> Maybe a`

Uživatelské typy 2

- Datové konstruktory generují výraz určitého typu. D.k. nejsou přetížené (nelze overloading), ale můžou být polymorfní (př. `Just`, `Left`)
- Každá varianta typu má dat. konstruktor (`Either`)
- D.k. pomáhají typovému systému při odvozování typů a slouží k rozlišení variant hodnoty (jako tagy)
- Porovnávat se vzorem lze i uživatelské typy

```
jePocatek (Pt 0 0) = True
jePocatek _        = False
unJust (Just x) = x
```

- Konkrétní hodnoty jsou konkrétního typu:

```
Pt 1 2 :: Point Int
Pt "ab" "ba" :: Point String
```

Typ Maybe

- Typ `Maybe a` slouží na přidání speciální hodnoty `Nothing` k typu daném parametrem `a`
- Hodnota `Nothing` je mimo typ a proto typ `a` můžeme používat celý (tj. `Nothing` nezabírá místo)
- Typ `Maybe` je polymorfní a proto jedna funkce nad `Maybe` slouží všem typům
- Typické použití: `lookup`, je polymorf. a pro `[]` mám co vracet `:: Maybe b`

```
mapMaybe :: (a->b) -> Maybe a -> Maybe b
mapMaybe f Nothing  = Nothing
mapMaybe f (Just x) = Just (f x)
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ [] = Nothing
lookup k ((h,v):xs)
  | k == h    = Just v -- zabalení v
  | otherwise = lookup k xs
```

Typ Either

- Umožňuje disjunktní sjednocení typů
- ”map” pro `Either` je jiné, potřebuje 2 fnc. arg.
- `swap` vyměňuje složky
- `left` aplikuje funkci `f` pouze na levé složky
- `swap` a `left` jsou ”pomocné” funkce

```
mapEither (a->b)->(c->d)->Either a c -> Either b d
mapEither f g (Left x)=Left(f x)
mapEither f g (Right y)=Right(g y)
swap :: Either a b -> Either b a
swap (Left x) = Right x
swap (Right y)= Left y
left f ei = mapEither f id ei

> map (zpJablka `mapEither` zpHrusky) es
```

- funkcionální parametr zkonstruuji z jednodušších funkcí (bez λ-funkce)

(Pojmenované položky)

- Dosud: položky podle polohy

```
data PointF = Pt Float Float
pointx :: PointF -> Float
pointx (Pt x _) -> x
```

- Hs98/Hs2010: pojmenované položky, jiná syntax pattern matchingu

```
data Point = Pt {pointx,pointy::Float} -- pojmenované složky
pointx::Point->Float --  implicitně zaváděné selektory
absPoint :: Point -> Float
absPoint p = sqrt(pointx p*pointx p + pointy p*pointy p)
absPoint2 (Pt {pointx=x, pointy=y}) =  sqrt(x*x + y*y)
```

Rekurzivní uživ. typy

```
data Nat = Z | S Nat --  monomorfní, numerály
data Tree a = Leaf a --  polymorfní
            | Branch (Tree a) (Tree a)
data Tree2 a = Void
             | Node (Tree2 a) a (Tree2 a)
data Tree3 a b = Leaf3 b
               | Branch3 (Tree3 a b) a (Tree3 a b)
data NTree a = Tr a [NTree a] --  n-ární stromy
```

- různé typy stromů pro různé použití (Tree2 pro BVS, NTree pro n-ární)
- omezení (na regulární typy): konstruktory typu na pravé straně definice mají stejné argumenty jako na (definující) levé straně

Rekurzivní typy

- pozorování: rekurzivní typy se dobře zpracovávají rekurzivními funkcemi
- přístup na složky: (zatím) porovnání se vzorem, (uživatelem definované) selektory jako `leftSub`

```
leftSub :: Tree a -> Tree a
leftSub (Branch l _) = l
leftSub _ = error "leftSub: unexpected Leaf"
ozdobT2 :: Int -> Tree2 a -> (Int, Tree2 (Int,a))
ozdobT2 i Void = (i,Void)
ozdobT2 i (Node l x r) = (iR,Node tL (iL,x) tR)
 where (iL,tL) = ozdobT2 i      l
       (iR,tR) = ozdobT2 (iL+1) r
test = Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
```

- ozdobT2: předávání stavu, test: testovací data

Typ NTree

```
data NTree a = Tr a [NTree a]
```

- nepřímá rekurze při definici typu
- struktura končí prázdným seznamem, nemá konstruktor pro ukončení rekurze
- typ konstruktoru: `Tr :: a -> [NTree a] -> NTree a`
- typické zpracování: 2 vzájemně rekurzivní funkce, jedna pro stromy, druhá pro seznam stromů

```
hloubkaNT (Tr x ts) = 1+maximum (hloubky ts)
 where hloubky [] = [0]  -- zarážka pro maximum
       hloubky ts = map hloubkaNT ts
```

Typová synonyma 1

- klíčové slovo `type`
- na pravé straně od '=' jsou jména typů
- neobsahuje datový konstruktor
- systém při výpisech t.synonyma nepoužívá, vrací rozepsané typy

```
type RGB = (Int,Int,Int)
type Complex = (Double,Double)
type Matice a = [[a]]
```

Typová synonyma 2

- klíčové slovo `newtype`
- definování nekompatibilního typu stejné struktury (typový systém nedovolí typy míchat, např. Int a Euro)
- datový konstruktor na pravé straně má právě jeden argument, a to původní typ
- typový konstruktor Euro vlevo a datový konstruktor Euro vpravo jsou v různých namespace
- datový konstruktor se používá pouze při kompilaci, run-time je bez overheadu
- časté použití: nová verze typu, která má jiné standardní operace, např. == a >

```
newtype Euro = Euro Int
plusEu :: Euro -> Euro -> Euro --  přetížení + později
Euro x `plusEu` Euro y = Euro (x+y)
```

...

- obecnější pomocná funkce:

```
lift2Eu :: (Int->Int->Int) -> Euro->Euro->Euro
lift2Eu op (Euro x) (Euro y) =  Euro (x`op`y)
plusEu = lift2Eu (+)
```

- fce `lift2Eu` je analogická fci `zipWith` pro (nerekurzivní) typ Euro

Case

- výraz `case` je základní metoda pro rozlišování konstruktorů
- výraz `case` je obecné porovnávání se vzorem, použitelné jako výraz, i pro uživatelské typy
- aktivuje se deterministicky první vzor, který vyhovuje
- používá 2D layout (nebo `{}` a `;`)
- lze použít také pro 1 zpracovávaný výraz (bez závorek okolo)

```
case (vyraz,...) of
  vzor  -> vyraz
  vzor2 -> vyraz2
  ...
```

Case: příklad

- `take2 n xs` vybírá prvních `n` prvků z `xs`, pokud existují, jinak všechny
- na nejvyšší úrovni definice funkce se typicky používají pattern matching místo `case`, viz `take` (z prelude)

```
take2 :: Int -> [a] -> [a]
take2 n xs = case (n,xs) of
   (0,_)  -> []
   (_,[]) -> []
   (n,x:xs) -> x: take2 (n-1) xs

take 0 _  = []
take _ [] = []
take n (x:xs) =  x:take (n-1) xs
```

Funkce 1: anonymní lambda-funkce

- funkce se dají ”za běhu” konstruovat lambda-abstrakcí (λ)
- formální parametry mají rozsah platnosti tělo definice

```
succ x = x+1 --  obvyklý zápis
succ = \x -> x+1 --  ekvivalentní
add = \x y -> x+y --  víc parametrů
add = \x -> \y -> x+y
```

- aplikace funkce (mezerou) je (jediný) selektor funkce. Hodnotu funkce lze zjistit v jednotlivých bodech.
- Anonymní funkce se často používají jako argumenty funkcí vyšších řádů. Definují se lambda abstrakcí tam, kde se použijí.
- *referenční transparentnost*: volání funkce na stejných parametrech vrátí stejnou hodnotu. Protože nemáme globální proměnné, přiřazení a sideefekty.
- Následně: výsledek nezávisí na pořadí vyhodnocování a proto je možné líné vyhodnocování

Funkce 2: skládání

- na funkcích není definována rovnost (==)
- skládání funkcí, identita `id` jako neutrální prvek pro skládání

```
(.) :: (b->c)->(a->b)->(a->c)
f . g = \x -> f(g x)
id x = x
```

- platí `id.f = f = f.id`
- aplikace, pro pohodlnější zápis

```
($) :: (a->b)-> a -> b
f $ x = f x
```

- `$` je asoc. doprava: `f3 $ f2 $ f1 x`

Funkce 3: zkrácený (bezbodový) zápis

- definice funkcí lze psát bez posledních argumentů na obou stranách; typový systém si poradí; tzv. bezbodový zápis (v teorii η-redukce $\lambda x.Fx = F$, "eta"-r.)
- typicky používáme při specializaci funkcí $\rightarrow$ proto funkcionální argumenty chceme jako první
- zafixování argumentu při částečné aplikaci neznamená optimalizaci, funkce čeká na všechny své argumenty a až potom se spustí

```
zip = zipWith (,)
odd :: Int -> Bool
odd = not . even --  bezbodový zápis, bez argumentu x
```

- př.: fce `filter` s opačnou podmínkou:

```
negFilter f = filter(not.f) -- vs. \x->not(f x)
```

Funkce 4: transformace funkcí

- transformace funkcí: někdy potřebujeme upravit funkce pro plynulé použití
- prohození dvou argumentů, `flip`, v prelude

```
flip :: (a->b->c) -> b->a->c
flip f x y = f y x
> map (flip elem tab) [1,2,3]
```

- funkce curry a uncurry, v prelude

```
curry :: ((a,b)->c) -> a->b->c
curry f x y = f (x,y)
uncurry :: (a->b->c) -> (a,b)->c
uncurry f (x,y) = f x y
```

Funkce 5: closure

- př: data jsou dvojice, ale funkce očekává samostatné argumenty → přizpůsobíme funkci

```
paryAll :: (a->a->Bool) -> [(a,a)] -> Bool
paryAll f xs = and $ map (uncurry f) xs
> paryAll (==) [(1,1),(5,5)]
```

- *Closure*, uzávěr: částečně aplikovaná funkce si zapamatuje svoje argumenty z lexikálního rozsahu platnosti, i když ji předáte mimo lexikální rozsah
- nejsou problémy s dealokací proměnných, používané nelokální proměnné jsou dostupné z closure

```
const :: a -> b -> a
const x y = x
let z = ..., cF = const z in cF
```

- Díky closure se hodnota `z` uvnitř `cF` zachová a může se použít i mimo blok, kde byla definována.

Nekonečné d.s. 1

- líné vyhodnocování vyhodnocuje pouze to, co je potřeba, shora (a jen jednou): pro výpis, pro pattern matching, pro arg. vestavěných funkcí ... (striktní vyhodnocení `$!`, striktní konstruktory)
- líné vyhodnocování umožňuje potenciálně nekonečné datové struktury
  - použije se jen konečná část (anebo přeteče paměť)
  - n.s. má konečnou reprezentaci v paměti ; už zpracované části se uvolní (pro garbage collector)

```
numsFrom n = n:numsFrom (n+1)
factFrom n = map fact (numsFrom 0)
fib = 1:1:[a+b|(a,b)<-zip fib (tail fib)]
```

Nekonečné d.s. 2

- Vytvoření nekon. struktur, z prelude

```
repeat x = x: repeat x
cycle xs = xs ++ cycle xs
iterate f x = x:iterate f (f x)

jednMat = iterate (0:) (1:repeat 0)
```

- např. při sčítání `jednMat` s konečnou maticí pomocí `zipWith` se vygeneruje a použije pouze konečná část

```
plusMat m1 m2 = zipWith (zipWith (+)) m1 m2
            ... (\r1 r2-> zipWith (+) r1 r2) ...
```

Nekonečné struktury a funkce

- psát funkce kompatibilně s líným vyhodnocováním:
  - mít funkce které odeberou pouze potřebnou část d.s. (`take`, `takeWhile` ...), tj. nestriktní funkce
  - mít funkce, které zpracují nekon. d.s. na nekon. d.s., tj. které na základě části vstupu vydají část výstupu. (`map`, `zipWith`, `filter` ...). Ale `reverse` takto nefunguje.
- def: *striktní funkce* vyhodnotí vždy svůj argument (úplně)
- pro striktní funkce platí: $f\perp = \perp$ (nedefinovaná hodnota, bottom)
- jazyk s hladovým/dychtivým vyhodnocováním (eager evaluation), např. Scheme, dovoluje psát striktní funkce; ale líné vyhodnocování (lazy eval.) umožňuje psát i nestriktní fce, např. také vlastní "řídící" struktury (např. `mojeif`) a optimalizované vyhl. v BVS *zapsáno* jako průchod celé struktury
- past: `[x|x<-[1..], x<4]`
  - nedá očekávaný výsledek `[1,2,3]`, ale `1:2:3:`$\perp$, tj. cyklí; systém neodvozuje "limitní" chování

Líné vyhodnocování

- Líné vyhodnocování umožňuje nekonečné struktury, ale:
- Nevýhoda líného vyhodnocování: *memory leak, únik paměti*: nespočítání výrazu a neuvolnění paměti, dokud hodnota není potřeba:
- očekáváte číslo, ale v paměti se hromadí nevyhodnocený výraz
- (Hackerský koutek: uživatel může požádat o striktní vyhodnocení `x` pomocí `$!`: př. `f $! x`. Často pro jednoduché hodnoty `x`.)
- Líné vyhodnocování nepomůže pokaždé. První verze vyhodnocuje celé `xs`, druhá pouze potřebný počátek:

```
asponN  xs n = length xs >= n -- vyhodnocuje celé xs
asponN' xs n = (length $ zip [1..n] xs) == n -- opt.
```

Příklady

- Pascalův trojúhelník (po uhlopříčkách) ; idiom FP: zpracování celých struktur

```
pascalTr = iterate nextR [1] where
  nextR r = zipWith (+) (0:r) (r++[0])
```

- Stavové programování: návrhový vzor *iterátor* funkcí `until`
- ve FP lze (někdy) napsat kód s funkcionálními parametry vs. pseudokód v OOP

```
until2 done next init =
  head (dropWhile (not.done) --  mezivýsl. se průběžně
          (iterate next init) ) --  zahazují
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f = go  where --  z knih., paměť. 2,8x lepší, čas 1,1x
    go x | p x           = x
         | otherwise     = go (f x)
```

- vrací celý interní stav (se kterým se počítá), nemá výstupní projekci
- protože se mezivýsledky průběžně zahazují, nemám v paměti celý seznam najednou

Polymorfní typy

- Pro odvozování typů se používá Hindleyův-Milnerův algoritmus, na části polymorfního λ-kalkulu.
- T: Funkce má v polymorfním λ-kalkulu jeden nejobecnější typ.
- Typ je polymorfní nebo monomorfní. Další typy získáme z nejobecnějšího polymorfního typu substitucí.
- Odvozování typů: unifikací. Pro funkci `f::a->b` a výraz `x::a'` se odvodí substituce $\sigma = mgu(a, a')$ a typ výrazu `f x::b`σ
- Pragmaticky: při odvozování se zjistí nekonzistence, ne místo nebo důvod chyby. Př.: při neúspěšné unif. typů `t1=[(a,b)]` a `t2=([c],d)` chybí `head` na `t1` nebo `fst` na `t2`?
- V Haskellu se používají typová rozšíření. Někdy musí uživatel typovému systému napovědět (tj. explicitně uvést typ). Pak se *typové odvozování* (type inference) mění na jednodušší *typovou kontrolu* (type checking) `read "1"::Int, const (1::Int)`
- Typy se používají pouze při kompilaci. Běh je bez overheadu.

Polymorfizmus a typové třídy

- 1. Parametrický p., pomocí typových proměnných. Př: `length::[a]->Int`
- Na typu argumentu nezáleží. Stejná impl. pro všechny typy. Kód 1x a funguje i pro budoucí typy.
- 2. Podtypový p. (přetížení, overloading), pomocí typových tříd. Př: `(==) :: Eq a => a->a->Bool`
- Na typu argumentu záleží. Různé implementace pro různé typy, jedna instance pro jeden typ. Pro nové typy nutno přidat kód.
- Porovnání: V Hs funkce mají typový kontext. V OOP objekty mají TVM-tabulku virtuálních metod. Implementačně: t.třídy se předávají jako další parametry.
- Hs podle (odvozeného) typu argumentu (správně) určí, kterou instanci typové třídy použít. Případně chyba: ’žádná’ nebo ’nejednoznačná’ třída.

Typové třídy

- Ne všechny operace jsou definovány na všech typech. Typová třída je abstrakce těch typů, které mají definovány dané operace.
- T.třídy: `Eq Ord Show Read Enum Num Integral Fractional Float Ix ...`
- Tento mechanizmus odpovídá přetížení, tj. ad hoc polymorfizmu.
- class - zavedení typové třídy
- instance - definování typu jako prvku typové třídy, spolu s definováním operací.
- Typový kontext funkce: podmínky na typy, tj. na typ. proměnné
- Př: funkci `eqpair` lze použít pouze pro typy, na kterých je definována rovnost. Funkce není přetížená (tj. kompilace 1x), pouze využívá přetížení funkce `(==)` z typové třídy.

```
eqpair :: (Eq a,Eq b) => (a,b)->(a,b)->Bool
eqpair (x1,x2) (y1,y2) = x1==y1 && x2==y2
```

Přetížení - detaily

- Přetížené konstanty:

```
> :t 1
1 :: Num a => a
```

- implementace využívá `fromInteger`
- analogie k []::[a]
- Nastavení přepínače `t`, aby Haskell vypisovat typy

```
> :set +t
```

- Chybné použití (+) na typ Char

```
> 'a' + 'b'
 No instance for (Num Char)
```

- Pro jeden typ lze mít pouze jednu instanci. Ale pomocí `newtype` lze získat izomorfní typ pro novou instanci.

Typové třídy, Eq

- Deklarace typové třídy (class) obsahuje signaturu, tj. jména a typy funkcí. Tyto funkce budou přetížené.
- Dále může obsahovat defaultní definice některých funkcí. Ty se použijí, pokud nejsou předefinovány.
- Třída Eq: typy, které lze porovnávat (na rovnost). Např. Bool, Int, Integer, Float, Char. Taky seznamy a n-tice, pokud položky jsou instance Eq.

```
class Eq a where
  (==),(/=) :: a->a->Bool   --  typy funkcí
  x/=y = not (x==y) --  defaultní definice
```

- typ rovnosti: `(==)::Eq a=>a->a->Bool`
- při použití `==` na typ `a` musí být pro typ `a` definována instance třídy `Eq`.

Instance

- pro vestavěný typ

```
instance Eq Int where
  x==y = intEq x y
```

- pro uživatelský neparametrický typ: def. rozpisem

```
instance Eq Bool where
  False==False = True
  True ==True  = True
  _    == _    = False
```

- pro parametrický uživ. typ , tj. typový konstruktor; typ prvků je instance Eq

```
instance (Eq a)=> Eq (Tree a) where
Leaf a        == Leaf b         = a==b
Branch l1 r1 == Branch l2 r2 = l1==l2 && r1==r2
_             == _              = False
```

Eq 2

- víc podmínek v typovém kontextu

```
instance (Eq a,Eq b) => Eq (a,b) where
  (a1,b1)==(a2,b2) = a1==a2 && b1==b2
```

- první rovnost je na dvojicích, druhá na typu `a`, třetí na typu `b`

Deriving

- Některé typové třídy, `Eq`, `Ord`, `Show`, `Read`, `Enum` si lze nechat odvodit při definici nového typu
- Používá se klíčové slovo `deriving`
- Vytvoří se standardní definice: rovnost konstruktorů a složek, uspořádání podle definic konstruktorů v definici typu a dále lexikograficky, v Read a Show konstruktory identickými řetězci

```
data Bool = False | True
  deriving (Eq, Ord, Read, Show)
data Point a = Pt a a deriving Eq
```

Třída Ord

- Hodnoty typu jsou lineárně uspořádané
- Musíme mít pro typ už definovanou rovnost, typ je instance Eq

```
class (Eq a) => Ord a where
 (<=),(<),(>=),(>) :: a->a->Bool
 min, max :: a->a->a
 compare :: a-> a-> Ordering
 x<y = x<=y && x/=y
 (>=) = flip (<=) --  prohodí args.
 min x y = if x<=y then x else y
```

- Funkce <= je základní. Defaultní funkce lze (např. kvůli efektivitě) předefinovat.

```
flip op x y = y `op` x
data Ordering = LT | EQ | GT
  deriving (Eq, Ord, Read, Show, Enum)
```

Třída Ord 2

- Instance Ord pro typy a typové konstruktory.

```
instance Ord Bool where
True  <= False = False
_     <= _     = True

instance (Ord a, Ord b) => Ord (a,b) where
(a1,b1) <= (a2,b2) = a1<a2 || a1==a2 && b1<=b2

instance (Eq [a], Ord a) => Ord [a] where
[]     <= _ = True -- lexikograficky
(x:xs)<=(y:ys) = x<y||x==y&& xs<=ys
_      <= _ = False
```

- Pro jeden typ pouze jedno uspořádání se jménem `<=`. Jiné třídění lze definovat na novém typu s využitím `newtype`.

Třídy Show, Read, Enum

- Třída Show a: hodnoty typu a lze převést na znakové řetězce. (Pouze převod na String, vlastní výstup je samostatně.)

```
show :: a -> String
```

- Třída Read a: hodnoty typu a lze převést ze znakového řetězce.

```
read :: String -> a
```

- při použití `read` je často nutno uvést typ: `read "1":: Int`
- Třída Enum a: výčtové typy - dovoluje používat speciální syntax pro typ a

```
enumFrom :: a->[a] -- [n..]
enumFromTo :: a->a->[a] --[k..n]
enumFromThen :: a->a->[a] --[k,l..]
enumFromThenTo :: a->a->a->[a] --[k,l..n]
```

Čísla

- Třída Num: potřebuje Eq a Show, instance pro Int, Integer, Float ...

```
(+), (-), (*) :: a->a->a
abs, negate, signum :: a->a
fromInt :: Int -> a --  převod ze standardních čísel
fromInteger :: Integer -> a
```

- Třída Integral a: potřebuje Num, instance Int, Integer

```
div, mod, quot, rem :: a->a->a
toInteger :: a-> Integer
```

- Třída Fractional a: potřebuje Num, hodnoty jsou neceločíselné. Instance: Float, Double (a přesné zlomky `Ratio a`).

- Třída Floating a: potřebuje Fractional, instance: Float, Double

```
exp, log, sin, cos, sqrt :: a->a
(**) :: a->a->a --
```
umocňovaní

- Třída Bounded a

```
minbound, maxbound: a
```

- Třída Ix a: pro indexy polí

```
range :: (a,a) -> [a]
index :: (a,a) -> a -> Int
```

Př: zbytkové třídy mod 17 (neúplné)

```
data Z17 = Z17 Int deriving (Eq, Ord, Show)
instance Num Z17 where
  Z17 x + Z17 y = Z17 ((x+y)`mod`17)
  (*) = lift2Z17 (*)
  fromInt x = Z17 (x `mod` 17)
lift2Z17 op (Z17 x) (Z17 y) = Z17 ((x`op`y)`mod`17)
```

- Motivace: FFT napsaná pomocí + - * (a fromInt) funguje v $\mathbb{C}$ (`Data.Complex`) i $\mathbb{Z}_{17}$ (`Z17`)
- tj. stejný kód parametrizovaný operacemi z typových tříd funguje pro různé typy

Třída Functor

- Třída pro typový konstruktor (ne pro konkrétní typ), na kterém lze definovat analogii `map`: struktura typu zůstane, jednotlivé prvky se změní podle funkcionálního parametru `fmap`.

```
class Functor a where
  fmap :: (b->c) -> a b -> a c

instance Functor [] where
  fmap f xs = map f xs
instance Functor Tree2 where
  fmap _f Void = Void
  fmap  f (Node l x r) = Node (fmap f l) (f x)
                              (fmap f r)
instance Functor Maybe where
  fmap f x = mapMaybe f x
instance Functor ((->) r)  where
  fmap = (.)  -- (a->b)->(r->a)->(r->b)
```

Strukturální rekurze pro seznamy: foldr

- Funkce foldr (svinutí) pro seznamy, doprava rekurzivní
- Nahrazuje konstruktory funkcemi, výsledek je lib. typu (vs. map, filter)
- Př: `foldr f z (1:2:3:[])` počítá `1`f`(2`f`(3`f` z))`

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

length xs = foldr (\_ n-> n+1) 0 xs
```

- někdy potřebujeme výslednou hodnotu dodatečně zpracovat (prumer), někdy potřebujeme jiný druh rekurze (maximum)

Použití foldr (12x + 1x)

```
sum xs = foldr (\x s->x+s) 0 xs
product xs = foldr (*) 1 xs
faktorial n = product [1..n]
reverse xs = foldr (\x rs->rs++[x]) [] xs -- v O(n^2)
concat xss = foldr (++) [] xss
xs ++ ys = foldr (:) ys xs
map f xs = foldr (\x ys->f x:ys) [] xs {- ((:).f)
iSort cmp xs = foldr (insert cmp) [] xs -- vs. jiné sort
  where insert = ...
and xs = foldr (&&) True xs
```

Funkce and ”zdědí” líné vyhodnocování (zleva) od `&&`

```
or xs = foldr (||) False xs
any p xs = foldr (\x b->p x||b) False xs -- duálně: all
all p xs = foldr (\x b->p x&&b) True xs
prumer xs = s/fromInt n where
  (s,n) = foldr (\x (s1,n1) -> (x+s1,1+n1)) (0,0) xs
```

Složenou hodnotu potřebujeme postzpracovat.

Varianty fold

- Varianty: na neprázdných seznamech `foldr1`; doleva rekurzivní `foldl` (a `foldl1`)

```
minimum :: Ord a => [a] -> a
minimum xs = foldr1 min xs
  --  fce. min nemá neutrální prvek, pro []
foldr1 :: (a->a->a) -> [a] -> a
foldr1 _ [x] = x
foldr1 f (x:xs) = x `f` foldr1 xs
```

- Zpracování prvků zleva, vlastně pomocí akumulátoru, pro konečné seznamy

```
foldl :: (a->b->a)->a->[b]->a
foldl f e [] = e
foldl f e (x:xs) = foldl f (e `f` x) xs
reverse = foldl (\xs x-> x:xs) [] --  lineární slož.
```

Ještě k fold

- V GHC v.8 je `foldr` a varianty definováno obecněji:
- Typ/typový konstruktor `t` považujeme za kontejner a přidáváme prvky postupně do výsledné hodnoty typu `b`.

```
foldr :: Foldable t => (a->b->b)->b-> t a ->b
```

- Zpracování předpon a přípon: `scanr, scanl`. ”fold” vydával pouze konečný výsledek, ”scan” vydává seznam mezivýsledků (jiný druh rekurze)

```
scanr :: (a->b->b)->b-> [a] -> [b]
scanl :: (a->b->a)->a-> [b] -> [a]
> foldr (:) [0] [1,2,3]
[1,2,3,0] :: [Integer]
> scanr (:) [0] [1,2,3] --  zpracování přípon
[[1,2,3,0],[2,3,0],[3,0],[0]] :: [[Integer]]
> scanl (flip (:)) [0] [1,2,3] --  zpracování předpon
[[0],[1,0],[2,1,0],[3,2,1,0]] :: [[Integer]]
```

Obecná rekurze: stromy

- Myšlenka nahrazovat konstruktory d.s. uživatelskými funkcemi jde použít i pro jiné struktury.

```
foldT :: (a->b)->(b->b->b)->Tree a->b
```

- Nahrazuju konstruktory `Leaf::a->Tree a` a `Branch`, kde typ `b` je místo `Tree a`

```
foldT fL fB (Leaf a) = fL a
foldT fL fB (Branch l r) = fB (foldT fL fB l)
                              (foldT fL fB r)
hloubka t = foldT(\_->1)(\x y -> 1+max x y) t
sizeT t = foldT(\x->size x)((+).(1+))t --\x y->1+x+y
mergeSort :: Ord a => Tree a -> [a] --  fáze 2 mergesortu
mergeSort t = foldT (\x->[x]) merge t
```

- Mergesort používá strukturální rekurzi podle stromu bez vnitřních vrcholů (`Tree a`)

... i pro nerekurzivní typy

- Myšlenka nahrazovat konstruktory d.s. uživatelskými funkcemi jde použít i pro nerekurzivní typy: zpracování hodnoty zabalené v `Maybe`.

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x

> maybe 0 (*2) (readMaybe "5")
10
> maybe 0 (*2) (readMaybe "")   -- při chybě default 0
0
```

- typ výstupu `maybe` je libovolný typ `b`, pro porovnání u `fmap` je to `Maybe b`

Unfold (myšlenka), pro seznamy

- Obecný vzor pro tvorbu seznamu, i nekonečného, tj. rekurze podle *výstupu*

```
unfold :: (b->Bool)->(b->(a,b))->b->[a]
unfold done step x
  | done x = []
  | otherwise = y: unfold done step yr
     where (y,yr) = step x
```

- Převod čísla do binární repr., od nejméně významných bitů

```
int2bin n = unfold (0==) (\x->(x`mod`2,x`div`2)) n
> int2bin 11 = 1:i 5=1:1:i 2=1:1:0:i 1=1:1:0:1:i 0=
    1:1:0:1:[]
```

- Pro struktury, pokud má datový typ víc konstruktorů, použijeme Either – případně opakovaně.

```
unfoldT2 :: (b->Either () (b,a,b)) -> b -> Tree2 a
```

Unfoldr, z knih. Data.List

- Pro seznamy můžeme použít `Maybe` pro analýzu hodnoty.

```
unfoldr :: (b->Maybe(a,b)) -> b -> [a]
unfoldr f b0 =
  let go b = case f b of
               Just (a,b1) -> a : go b1
               Nothing     -> []
  in go b0
```

- Příklady použití, `b` je stav

```
> unfoldr (\b -> if b == 0 then Nothing
                             else Just (b, b-1)) 10
[10,9,8,7,6,5,4,3,2,1]
iterate f == unfoldr (\x -> Just (x, f x))
selectSort cmp xs = unfoldr selectMin xs
  where selectMin []     = Nothing
        selectMin (x:xs) = Just (selectMin' x xs)
        ...
```

Počítání s programy, rovnostní odvozování

- Typické použití odvozování při dokazování vlastností programu:
  1. pro částečnou správnost vůči specifikaci (a požadavkům),
  2. optimalizaci/efektivitu.
- V bezstavovém programování se s programy lehce počítá. (Čistý λ-kalkul je rovnostní teorie.)
- Příklad: asociativita (++), pro konečné seznamy x, y, z.
- Chceme dokázat: `x++(y++z) = (x++y)++z`, levá strana je efektivnější
- Rozbor případů pro `x=[]` a `x=a:v`. Podtržená část se upravuje:

```
  [] ++ (y++z)
= {def ++.1}
  y++z
= {def ++.1}
  ([]++y)++z
```

```
  (a:v)++(y++z)
= {def ++.2}
  a:(v++(y++z))
= {indukční předpoklad}
  a:((v++y)++z)
= {def ++.2}
  (a:(v++y))++z
= {def ++.2}
  ((a:v)++y)++z
```

- DC: (map f.map g) x = map (f.g) x , pravá strana je efektivnější, protože netvoří mezilehlý seznam
- Pozn. Např. pro `fmap` z `Functor` má (podle teorie) platit
  1. `fmap id = id`
  2. `fmap f . fmap g = fmap (f.g)`

  Haskell takové rovnosti neumí ověřit (automaticky, zatím), ale umožňuje přidat optimalizační přepisovací pravidlo.

Monády, idea

- Monáda `M a`: hodnoty typu `a` jsou zabaleny v ”kontejneru” `M`, který obsahuje přidanou informaci.
- `M a` je typ výpočtu: `[a]` pro nedeterminizmus, `Maybe a` pro výpočty s chybou, se stavem, s kontextem, s výstupem, `IO`, Id pro ”prázdnou” monádu ...
- Hodnotu `a` nemůžeme ”vybrat” z monády a pracovat s ní přímo. Monadická hodnota se zpracuje na monadickou hodnotu.
- Monadický kód určuje pořadí vyhodnocování (důležité pro `IO`)
- Místo fce `zdvih` podobné `map` se používá `(>>=)`, tzv. `bind`, které popisuje pořadí vyhodnocování (jednovláknovost)
- Druhá funkce je `return`, která vyrábí hodnotu `M a`.

```
zdvih :: (a -> M b) -> M a -> M b
map   :: (a -> M b) -> M a -> M(M b)
(>>=) :: M a -> (a -> M b) -> M b
return :: a -> M a
```

(Monády, alternativně)

- Funkce `f::a->M b` a `g::b->M c` máme problém složit (na rozdíl od nemonadických `f'::a->b` a `g'::b->c`). Funkce `(>>=)` pomůže: (jinak bychom potřebovali `g''::M b->M c`)

```
compM f g = \x -> f x >>= g
```

- Monády mají v teorii vlastnosti (asociativita `>>=`, `return` je neutrální prvek pro bind), které překladač nekontroluje (zatím).
- `return`, `join` a `map` (konkrétně `fmap` z třídy `Functor`) je alternativní definice monády, místo `return` a `(>>=)`

```
map  ::                  (a->b)->(M a->M b)
fmap :: Functor M =>     (a->b)->(M a->M b)
join :: M(M a) -> M a
join mma = mma >>= id
ma >>= f = join $ map f ma
```

Monády

- Monády pro typové konstruktory jsou definované pomocí typových tříd (v GHC 8 postupně `Functor M`, `Applicative M`, `Monad M`), tj. `>>=` a `return` jsou přetížené.
  (Následně jsou ze základních funkcí odvozené obecnější funkce, které jsou potom použitelné pro všechny monády.)
- Jednotlivé monády mají specifické výkonné funkce. (Každá monáda má jiné funkce, tj. nejsou součástí interface monády.)
- Používané monády nad typem `a`:
  - výpočet s chybou: `Maybe a`,
  - nedeterministický výpočet: `[a]`,
  - vstup/výstup: `IO a`,
  - výpočet s kontextem: `r->a`, (jako `Reader r`)
  - výpočet s výstupem: `(w,a)`, (jako `Writer w`)
  - výpočet se stavem: `s->(s,a)`, (jako `State s`)
  - continuation monad: `(a->r)->r`, (jako `Cont r`)
  - ...

Monáda: Maybe

- Typ `Maybe a`: monáda s chybou. Specifická funkce je vyvolání chyby, tj. vrácení `Nothing`.

```
return x       = Just x
Nothing >>= f = Nothing --  chyba se propaguje
Just x  >>= f = f x

data PlusT a = ConT a
             | PlusT a :+ PlusT a
             | VarT String
evalM :: Num a => PlusT a -> Maybe a
evalM (ConT a)    = Just a
evalM (va :+ vb) = evalM va >>= \a ->
                   evalM vb >>= \b ->
                   return (a+b)
evalM _           = Nothing
```

Monády: seznamy

- Seznam je monáda pro nedeterminizmus:
- Specifická akce je vrácení dvou (a víc) výsledků.

```
return x      = [x]
[]     >>= f  = []
(x:xs) >>= f  = f x++(xs>>=f)
join          = concat
```

- Zpracování výsledků probíhá zleva a do hloubky. Pořadí ve výstupním seznamu je určeno funkcí `>>=` .

Monadický vstup a výstup

- Potřebujeme určit pořadí operací, což při líném vyhodnocování nejde přímo.
- Pro V/V se používá typový konstruktor `IO`, kde `IO a` je typ vstupně-výstupních akcí, které mají vnitřní výsledek typu `a`. (IO je podobné monádě pro stav)

```
type IO a = World -> (a, World) --  pro představu
getChar :: IO Char --  načte znak do vnitřního stavu
putChar :: Char -> IO () --  vypíše znak, výsledek je nezajímavý
getLine :: IO String --  načte řádek
putString :: String -> IO () --  vypíše řetězec na std. výstup
```

- Na vnitřní stav se nedostaneme přímo, např. `upIO::IO a -> a`, porušila by se referenční transparentnost: `upIO getChar` vrací různé výsledky.

- Připomínám pro převody:
  ```
  show :: Show a => a -> String,
  read :: Read a => String -> a
  ```
- Čisté funkce nepracují s `IO`. Pragmaticky: používáme čisté funkce co nejvíc.
- Dříve uvedené funkce pracují se standardním vstupem a výstupem. Lze pracovat i se soubory, pomocí handle (otevření, načtení, výpis do, uzavření): knihovna `System.IO` se standardně načítá.

IO 2

- V `IO` máme `(>>=) :: IO a -> (a -> IO b) -> IO b`
- příklad: převod na velké písmeno:

```
(getChar >>= putChar . toUpper) :: IO ()
 getChar >>= \c -> putChar (toUpper c)
```

- Operátor `(>>)` ignoruje výsledek první akce, např. `putChar::IO ()` (když nás zajímají pouze ”sideefekty”)

```
(>>) :: IO a -> IO b -> IO b
o1 >> o2 = o1 >>= \_ -> o2
> putStr "Hello, " >> putStr "world"
sequence_ :: [IO ()] -> IO ()
sequence_ = foldr (>>) (return ())
> sequence_ (map putStr ["Hello,"," ","world"])
```

- `IO` poskytuje jednoznačný výstup i při líném vyhodnocování

Program s IO, pro kompilaci

- příklad: převod 1 řádku na velká písmena
- Proveditelný program se jménem `Main` a typem `IO ()` lze skompilovat a spustit.

```
-- import System.IO  -- v Prelude
main :: IO () --  main pro kompilované programy
main =
  putStr "Zadej vstupní řádek:\n" >>
  getLine >>=
  \vstup -> putStr "Původní řádek:\n" >>
  putStr vstup >>
  putStr "\nPřevedeno na velká písmena:\n" >>
  putStr (map toUpper vstup)
```

Práce v monádě Maybe

- Definujme `safeHead` a analogicky `safeTail`.

```
safeHead :: [a] -> Maybe a --  výst. typ  Maybe
safeHead []     = Nothing  --  hlava neexistuje, chyba
safeHead (x:xs) = Just x   --  hlava zabalená v  Just
```

- Bezpečný součet prvních dvou čísel ze seznamu v monádě `Maybe`. Vrací `Nothing`, pokud čísla neexistují.

```
safePlus xs =
  safeHead xs  >>= \x1 ->
  safeTail xs  >>= \xs1 ->
  safeHead xs1 >>= \x2 ->
  return(x1+x2)
```

- Monáda ošetřuje informace navíc, zde `Nothing` a zabalení v `Maybe`, tj. pattern matching je schovaný.

Maybe 2

- Funkce `safeHead` a `safeTail` se při monadickém zpracování nemění: dostávají nechybové vstupy. Naopak, kontrolují vstup a případně chybu vyvolávají.

Do-notace

- Speciální syntax pro monády: do-notace: podobná stručným seznamům (bez `let`)

```
> do {x<-[1,2];y<-[3,5]; return(x+y)}
[4,6,5,7]
```

- do-notace umožňuje psát kód podobný procedurálním jazykům, proto je důležitá a oblíbená.
- Monadické zpracování běží na pozadí.
- příklad výše:

```
safePlus xs =
 do {x1 <- safeHead xs; xs1 <- safeTail xs;
     x2 <- safeHead xs1; return(x1+x2)}
```

Do-notace

- Do-notace je použitelná pro lib. monádu: je přetížená. Používá často 2D syntax.

```
do v1 <- e1
   v2 <- e2
   e3   -- když návr. hodn. nepotřebujeme, ale 2D-zarovnané
   ...
   vn <- en
   return (f v1 v2 ... vn)
```

- Je syntaktický cukr pro

```
e1 >>= \v1 ->
e2 >>= \v2 ->
e3 >>= \_  -> -- neboli  e3 >> ...
...
en >>= \vn ->
return (f v1 v2 ... vn)
```

Outline

Moduly, knihovny

- Použití hierarchických knihoven a modulů:
- Načte se celý soubor, případně pouze definice (funkcí, datových typů, ...) uvedené v závorkách. Knihovny jsou kompilované a proto rychlejší.
- Každý import na samostatném řádku

```
import Data.List (sort)
import qualified Data.List (sort) as DL
```

- V 2. případě používáme volání `DL.sort` pro rozlišení, tzv. *kvalifikované jméno*
- Definice modulu, v závorkách případně seznam exportů: (jménoTypu(konstruktory), funkce)

```
module Z17 (Z17(Z17),mInv) where
...
```

- Modul `Z17` se hledá jako soubor `Z17.hs` nebo `Z17.lhs`

Striktní vyhodnocení

- Měli jsme `($)` jako aplikaci: `f $ x = f x`
- Funkce `($!)` je striktní aplikace
- - volání `f $! x` vyhodnotí nejvyšší konstruktor `x`, potom volá `f x`
- Pokud je x elementárního typu, vyhodnotí se *úplně*; po vyhodnocení víme, že není $\bot$; x se vyhodnocuje do tzv. hlavové normální formy (HNF).
- Motivace: předcházení memory leak

Příklad: foldl'

- Příklad: foldl nevyhodnocuje akumulátor hned, tj. nastává memory leak.

```
length xs = foldl (\d x->d+1) 0 xs
> length [1,2]
==> foldl .. 0 [1,2]
==> foldl .. (1+1+0) []
```

- Akumulátor chceme vyhodnocovat hned: definujeme `foldl'` pomocí `$!`

```
foldl' f e []     = e
foldl' f e (x:xs) = (foldl' f $! (e`f`x)) xs
```

- Používá se primitivní funkce (nedefinovatelná v Hs): `seq :: a->b->b` : vyhodnotí nejvyšší konstruktor prvního arg. (tj. HNF), pak vrací 2. arg.

```
f $! x = x `seq` f x
```

Třídy pro monády

- Třídy `Functor f`, `Applicative f`, `Monad m`

```
class Functor f where
    fmap    :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
    pure    :: a -> f a
    (<*>)   :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
    (>>=)   :: forall a b. m a -> (a -> m b) -> m b
    (>>)    :: forall a b. m a -> m b -> m b
    return  :: a -> m a
    m >> k = m >>= \_ -> k
    return  = pure
```

- varianta `bind (>>=)` s přehozenými arg. pro porovnání typů

```
    (>>=)'  :: (a -> f b) -> f a -> f b
```

FFT

- Rychlá Fourierova transformace

```
fft :: Num a => a -> [a] -> [a]
fft _ [x] = [x]
fft e xs  = vs  where
  (sude,liche) = rozdel xs
  vs1 = fft (e*e) sude
  vs2 = fft (e*e) liche
  c2 = zipWith (*) vs2 (iterate (e*) 1)
  vs  = zipWith (+) vs1 c2 ++ zipWith (-) vs1 c2

t2fft = fft (Z17 2) [0,1,0,0,0,0,0,0] -- v Z17
t3fft = fft (0:+1)  [0,1,0,0]         -- v Complex
t4fft = fft (cis (pi/4)) [0,1,2,3,4,3,2,1]
```

Funkcionální styl 1 2

-
-
-
-
-
-