Haskell v příkladech.

Jan HRIC*

6. prosince 2002, v.0.9

Obsah

*`Jan.Hric@mff.cuni.cz`

Abstrakt. Haskell v příkladech.

1 Úvod

Cílem této knihy je ukázat na příkladech základní myšlenky funkcionálního programování a jazyka Haskell.

Funkcionální programování je založeno na zápisu právě počítané hodnoty (programu) ve tvaru výrazu. Hlavní součásti výrazu jsou funkce a jejich aplikace na argumenty. Program je ve tvaru rovností, které se používají jako přepisovací pravidla, tj. výraz na levé straně se nahradí výrazem na pravé straně. Výpočet programu spočívá ve zjednodušování výrazu podle přepisovacích pravidel, dokud to lze. Závěrečný dále nezjednodušitelný výraz, pokud existuje, považujeme za výsledek výpočtu.

Například první výraz se vyhodnotí pomocí vestavěnš funkce `*` aplikované na argumenty `5` a `6`. Druhý výraz je aplikace funkce *kvadrat* na jeden (složený) argument. Při vyhodnocení se použije definovaná funkce *kvadrat*.

```
> 5 * 6
30

> kvadrat (3+4)
49
```

Program v Haskellu je posloupnost definic funkcí, zapsaná v externím souboru, nazývaném *skript*. Program se načte do prostředí. Výpočty jednotlivých výrazů pak mohou používat definované funkce. Příklad jednoduchého skriptu je

```
kvadrat x  = x * x
na_treti x = x * kvadrat x
```

Haskell je typovaný jazyk. Každý výraz má určitý typ. Uživatel má možnost definovat nové typy, které jsou rovnocenné s vestavěnými typy. Naopak, typy které používá, musí definovat. Jednotlivé funkce programu nemusí mít typ uveden, protože Haskell si je schopen typ odvodit. Z tohoto pohledu se jedná o jazyk s implicitním typováním.

Každý praktický programovací jazyk musí mít vestavěné funkce, které (efektivně) realizují základní operace, např. aritmetické. Na druhé straně některé očekávané operace, např. logické, lze zapsat v jazyku jako knihovnu. V případě Haskellu je při spouštění prostředí načítán soubor `prelude.hs`. Tento soubor obsahuje zdrojový kód, který z našeho hlediska budeme považovat za součást jazyka. Pokud budeme vysvětlovat funkce uvedené v tomto souboru, upozorníme na to v textu nebo v komentáři k zdrojovému kódu.

V tomto textu se věnujeme základům funkcionálního programování a základům jazyka Haskell. Vysvětlované myšlenky jsou zachyceny na malých příkladech. Nevěnujeme se programování ve velkém, tj. vysvětlení, jak navrhnout a rozdělit rozsáhlý program nebo systém na části.

Další informace a programy je možno získat na webové stránce `http://haskell.org`.

Porovnání s imperativním programováním. Hm.

2 Hodnoty a typy

2.1 Čísla ...

Haskell má celá čísla, např. `5` je číselná konstanta. Každý výraz je určitého typu, píšeme `5 :: Int`. Identifikátor `Int` je jméno typu. Ve jménech identifikátorů se rozlišuje medzi velkými a malými písmeny, konkrétní typy (typové konstanty) začínají velkým písmenem.

Symbol "`::`" znamená "má typ" nebo "je typu", tj. 5 má typ *Int*.

Na číslach jsou definovány běžné aritmetické operace `+`, `*` ...

Dále je v Haskellu zaveden typ `Integer` pro celá čísla s libovolnou presností. Např. 10000000000 :: Integer. Základní operace jako (+), (*), (-) pro typy Int a Integer jsou stejné a přetížené. Chování a definice přetížených funkcí bude vysvětlena později, v kapitole o typových třídách.

Reálna čísla jsou čísla s desetinnou tečkou, např. `1.42 :: Float`, resp. typu `Double`

2.2 Boolovské hodnoty

Logické konstanty jsou `True` a `False`. Základní logické operace jsou `(&&)` – *and*, `(||)` – *or*, `not`.

Binární operace mají líné vyhodnocování zleva.

HM.

.

3 Seznamy

3.1 Porovnávání se vzorem

Porovnávání se vzorem je technika ...XX ... syntaktická konstrukce, rys jazyka

Příklady

Příklad 3.1 *Transpozice matice.*

Příklad 3.2 *Převrácení šifrovací mřížky.* a) podle svislé osy, b) podle vodorovné osy. Zrcadlová mřížka.

Příklad 3.3 *Otočení šifrovací mřížky.* a) doprava b) doleva.

Operace na seznamech

Délka seznamu. take, drop, takeWhile, dropUntil

(Funkcionální argumenty) (rozumná) Motivace, toUpper, toLower -¿ map, filter, zipWith.

3.2 Stručná syntax seznamů

Protože seznamy se používají často, byla navržena speciální syntax pro jejich pohodlnější zpracování.

Zápis je `[f x1 ... xk | kvalifikatory ]` ..., kde `f` je funkce počítající výstup pro každou skupinu hodnot vygenerovanou kvalifikátory. Jednotlivé kvalifikátory jsou ve tvaru

- `x <- seznam` (generátor)
- `p` (test)
- `let x =  vyraz` (definice)

Příklad. Funkce *map* () Pro každý prvek z *xs*, pojmenovaný *x*, se do výsledného seznamu dá *f x*.

```
map :: (a->b) -> [a] -> [b] -- prelude
map f xs = [ f x | x<-xs ]
```

Příklad. Funkce filter. Vybere ze seznamu *xs* všechny prvky, které splňují podmínku *p*.

```
filter :: (a->Bool) -> [a] -> [a] -- prelude
filter p xs = [ x | x<-xs, p x ]
```

Již vybrané prvky se můžou použít v dalších generátorech (napravo). Naopak generovaný prvek se nemusí použít ve výrazu, ale není to obvyklé. Místo *x*, které se nepoužívá v dalších kvalifikátorech ani ve výstupní funkci, lze použít vzor podtržítko (_).

```
usp_dvoj n :: [(Int,Int)]
usp_dvoj n = [(x,y)|x<-[1..n],y<-[x..n]]
osm_mezer :: [Char]
osm_mezer = [' ' | x <- [1..8] ]
```

Příklad. Kartézský součin je jednoduchá operace. V první verzi používáme explicitní rekurzi, ve druhé stručné seznamy.

```
cart :: ([a],[b]) -> [(a,b)]
cart ([],_) = []
cart (x:xs,ys) =
  map (\y->(x,y)) ys ++ cart (xs,ys) -- OK
```

Kartézský součin pomocí stručných seznamů.

```
cart1 :: ([a],[b]) -> [(a,b)]
cart1 xs ys = [ (x,y) | x<-xs, y<-ys]
```

– def. pomocou foldr:

```
cart2 (xs,ys) =
  foldr (++) []
        (map (\x->map (\y->(x,y)) ys) xs)
```

– i jinak

Hodnoty generovaných proměných se fixují zleva, proto volání

```
> [(x,y) | x<-[1,2], y<-[3,4] ]
```

dá výsledek `[(1,3),(1,4),(2,3),(2,4)]`.

Kombinace, variace ... Pomocí stručných seznamů lze generovat struktury i rekurzivně.

Příklad. Kombinace (bez opakování). První parametr je počet prvků v kombinaci, druhý seznam prvků, ze kterých se vybírá. Zadání druhého parametru explicitně jako seznam umožňuje generovat kombinace z libovolných (zadaných) prvků.

```
komb :: Int -> [a] -> [[a]]
komb 0 _          = [[]] -- vhodně "dodef."
komb (n+1) []     = []
komb (n+1) (x:xs) =
  [x:ko| ko<- komb n xs] ++ komb (n+1) xs
```

Toto řešení má nevýhodu, že zkouší všechny možnosti i pro *n ¿ length xs*, dokud nenarazí na některou koncovou podmínku. Program lze zefektivnit explicitním testováním vztahu *n* a *length xs*. Kód přidáme mezi 1. a 2. klauzuli původního programu.

```
komb k xs | k == length xs = [xs]
          | k >  length xs = []
```

Cvičení. Kombinace s opakováním.

Příklad. Variace s opakováním. Pro variace změníme reprezentaci domény, tj. variace budeme chápat (a vracet) jako přiřazení z oboru definice (domain) do oboru hodnot (range). Tj. i obor hodnot předáváme jako explicitní seznam a výsledek bude seznam dvojic, viz typ.

```
varOpak :: [a] -> [b] -> [[(a,b)]]
varOpak [] _ = [[]] -- vhodně "dodefinováno"
varOpak (x:xs) r =
  [(x,y):vs| y<-r, vs <- varOpak xs r]
```

Reprezentaci analogickou ke kombinacím můžeme získat z této rozšířené reprezentace:

```
varOpak1 :: Int -> [b] -> [[b]]
varOpak1 k r =
  map      -- pro všechny variace
   (map    -- pro všechny prvky ve var.
     snd)  -- zahazuju dom., vracím hodnotu
   (varOpak  -- volání pův. fce dostane:
     [1..k]   -- domain jako seznam
     r)    -- range bez změn
```

Cvičení. Variace bez opakování.

Varianty programu a efektivita generování. Efektivita, čas vs. paměť.

V definici `varOpak.2` se hodnota seznamu pro generování *vs* počítá znovu pro každou vybranou hodnotu y. To je neefektivní. Jeden způsob odstranění je prohodit pořadí generátorů

```
varOpak (x:xs) r =
  [(x,y):vs| vs <- varOpak xs r, y<-r ]
```

ale pak dostáváme variace v jiném, méně prirozeném pořadí. Druhý způsob odstranění (časové) neefektivity je zavést novou proměnnou před generátorem y, která se ale bude využívat až za ním, tj. vlastní generátory zůstanou v původním pořadí.

```
varOpak k r =
  [(x,y):vs
  | let vars = varOpak xs r, y<-r, vs<-vars]
```

Při tomto způsobu se bude seznam pro generátor vyhodnocovat jen jednou. Ale pak musí ve vyhodnoceném tvaru být celou dobu v paměti. Protože vyhodnocený tvar je veliký, může paměť lehce přetéct. Jinými slovy, vyměnili jsme časovou efektivitu za zvýšenou spotřebu paměti.

Příklad. Částečné variace. ...

... permutace

Ladění programů: testování pomocí vlastností. (viz systém Quickcheck) Pro kombinační čísla C(n,k) platí rovnost

$$C(n+1,k+1) = C(n,k+1) + C(n,k)$$

Po přepsání do programu můžeme její platnost testovat pro jednotlivá čísla. Nesplnění rovnosti znamená chybu v programu (nebo v popisu podmínky `:-(`).

```
prop_komb_rozklad n k =
  length(komb (k+1) [1..n+1]) ==
      length(komb (k+1) [1..n])
    + length(komb k [1..n])
test_komb = filter (\(n,k,test)->test)
              [prop_komb_rozklad n k
              | n<-[3..6],k<-[1..n] ]
```

Uvedená testovací funkce testuje vlastnost systematicky pro malé hodnoty vstupů. Testovací funkce vrátí konkrétní hodnoty n a k, na kterých se projevila chyba. Jiný způsob je testovat na (pseudo)náhodných datech.

Vlastnosti funkcí uvedené v další kapitole o dokazování lze použít i pro testování.

3.3 Dokazování vlastností programů

O funkcích v programech můžeme dokazovat tvrzení, obvykle strukturální indukcí. Tyto tvrzení lze použít například pro úpravy programů do efektivnějšího tvaru nebo pro dokázání správnosti vůči specifikaci (dané průhledným programem).

Příklad. Asociativita *append*. Funkce je definovaná

```
[]    ++ v = v          (1)
(a:u) ++ v = a:(u++v)   (2)
```

Pro konečné seznamy platí

```
x ++ (y ++ z) == (x ++ y) ++ z
```

Dokazovat to budeme indukcí podle struktury `x`.

Počáteční tvrzení pro x = [] je

```
[] ++ (y ++ z) == ([] ++ y) ++ z
```

Postupnými úpravami podtržených výrazů dostáváme pro levou stranu

$\underline{\texttt{[] ++ (y ++ z)}}$

= podle append.1

`(y ++ z)`

a pro pravou stranu

$\underline{\texttt{([] ++ y)}}$ `++ z`

= podle append.1

`(y ++ z)`

teda rovnost platí. Na levé straně jsme použili rovnost append.1 pro hodnotu $v = y{+}{+}z$, na pravé straně jsme použili stejnou rovnost pro úpravu uvnitř výrazu, tj. v kontextu $C[.] = (. + +z)$.

Indukční krok je tvrzení

```
(a:x) ++ (y ++ z) == ((a:x) ++ y) ++ z
```

za předpokladu

```
x ++ (y ++ z) == (x ++ y) ++ z
```

Upravíme levou stranu

$\underline{\texttt{(a:x) ++ (y ++ z)}}$

= podle append.2

`a:(x ++ (y ++ z))`

a pravou stranu

$\underline{\texttt{((a:x) ++ y)}}$ `++ z`

= podle append.2

```
(a:(x ++ y)) ++ z
      =      podle append.2
a:((x ++ y) ++ z)
```

na tvary, které jsou podle indukční hypotézy stejné.

Výraz na levé straně je efektivnější pro výpočet, protože seznam `x` se prochází (ve voláních `++`) pouze jednou.

Cvičení. Dokažte pro konečné seznamy `x`

```
x ++ [] == x
```

Cvičení. Dokažte pro funkce f a g a seznam x

```
(map f . map g) x == map (f.g) x
```

Výraz na pravé straně je efektivnější, protože se nebuduje seznam pro mezivýsledky po volání `g`.

Cvičení. Dokažte pro konečné seznamy x a y

```
reverse (x ++ y) == reverse y ++ reverse x
```

Rovnost je (část) specifikace, která vyjadřuje požadované chování funkce `reverse`.

3.4 Strukturální rekurze

Rekurze programu je často řízená datovými strukturami, se kterými se pracuje. Může být řízená strukturou vstupu nebo strukturou výstupu, ale může mít i strukturu, která se přímo v programu neobjevuje. (To si lze představit jako dvě rekurze, kde první vygeneruje složky abstraktní struktury a druhá je hned zpracuje. Např. quicksort je program tohoto druhu.)

Rekurze řízená vstupem odpovídá funkci `fold`. Tato parametrizovatelná funkce je abstrakce strukturální rekurze. Zde si ukážeme příklady parametrické strukturální rekurze pro seznamy. Podobný přístup lze použít i pro jiné rekurzivní struktury včetně struktur definovaných uživatelem.

Parametrická strukturální rekurze. Nejprve rozebereme funkci *svinutí* ($foldr$) pro seznamy. Tato funkce v základní verzi podle daných dvou funkcionálních parametrů zpracuje daný seznam. Po vysvětlení základní verze si ukážeme varianty, které se používají ve případech, kdy nevyhovuje základní varianta rekurze.

Funkce $foldr$ je typu $(a \to b \to b) \to b \to [a] \to b$. Tj. má dva funkcionální parametry, které odpovídají konstruktorům seznamu, a dále vstupní seznam. Volání $foldr \oplus e\ xs$ bude počítat hodnotu funkce $x_1 \oplus (x_2 \oplus \ldots (x_n \oplus e) \ldots)$, kde xs je $[x_1, x_2, \ldots, x_n]$. Operace $\oplus$ ve výrazu asocionaná je doprava, proto se funkci $foldr$ říká svinutí sprava (závěrečné *r z anglického right*).

```
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

... vysvětlení strukturální rekurze na (++)

Pomocí funkce foldr se dají zapsat mnohé funkce zpracovávající seznamy bez nutnosti psát rekurzi explicitně. To znamená, že kód bude stručnější a případné chyby ve společné části se projeví ve více případech.

```
idL :: [a]->[a] -- identita pro seznamy
idL = foldr (:) []

sum :: [Int] -> Int     -- suma seznamu
sum = foldr (+) 0       -- z "prelude" ??

length :: [a] -> Int    -- délka seznamu
length = foldr (\x y -> y+1) 0 -- z "prelude"

concat :: [[a]] -> [a]  -- spojení seznamů
concat = foldr (++) []  -- z "prelude"

(++) :: [a]->[a]->[a]   -- spojení seznamů
(++) xs ys = foldr (:) ys xs    -- z "prelude"

reverse :: [a]->[a]     -- obracení seznamu
reverse = foldr (\x ys->ys++[x]) [] -- z "prelude"

filter :: (a->Boolean)->[a]->[a]
filter p = foldr (\x->if p x then (x:) else id) []

insertsort :: (Ord a) => [a] -> [a]
insertsort = foldr insert []  where
        insert x [] = [a]
        insert x (y:ys)
                | x <= y = x:y:ys
                | True   = y: insert x ys
```

V těchto příkladech kromě posledního jsme použili zkrácený zápis funkcí. To bylo možné, protože jediný parametr definovaných funkcí je zároveň posledním parametrem volání $foldr$.

Z definice funkce idL, která definuje identitu pouze pro typ seznamů, vidíme, že pokud konstruktory ponecháme beze změny, pak i výsledek se nemění.

Na příkladu funkce sum vidíme, že $foldr$ zobecňuje binární operaci, v tomto případě sčítání, na n-ární operaci, pracující na seznamu. Konstanta 0 je neutrální prvek sčítání a součet prázdného seznamu je vhodné proto definovat jako 0.

Podobně se funkce $length$ využívá konstantu 0 jako délku prázdného seznamu. Na rozdíl od sum se v funkcionálním parametru nevyužívá hodnoty právě zpracovávaného prvku seznamu, tj. formální paramer x se nevyskytuje v těle funkce. Za každý zpracovaný prvek seznamu se pouze přičte 1 k výstupní hodnotě. Protože hodnota prvků seznamu se nepoužívá, je typ vstupního parametru $[a]$, tj. seznam prvků libovolného typu, i když výstupní typ je určený.

První funkcionální parametr volání $foldr$ lze napsat jako (*const* (1+)). Tato funkce je konstantní v prvním parametru, tj. ignoruje ho, a přičítá 1 ke druhému, což je ekvivalentní původní funkci.

Případ funkce *concat* je opět zobecnění binární operace spojení dvou seznamů. Neutrální prvek spojování je prázdný seznam, který je hodnotou pro prázdný vstupní seznam. Protože spojování je nekomutativní operace, je důležité, v jakém pořadí se spojují dva seznamy v prvním (funkcionálním) parametru $foldr$.

Na příkladě definice (++) vidíme, že definovaná funkce může mít víc parametrů, které se můžou použít ve volání $foldr$.

Funkce $reverse$, $filter$ a $insertsort$ jsou příklad toho, že zpracování jednoho prvku může být složitější a výsledek nemusí obsahovat prvky v původním pořadí. V případě funkce $filter$ se využívá curryfikace, tj. funkcionální parametr zapsán jako funkce jedné proměnné, ale vrací funkci druhé původní proměnné. DOdejme, že $reverse$ v tomto případě má kvadratickou složitos, protože noné prvky se připojují po jednom na konec seznamu.

Varianty svinutí. Funkce $foldr$ není v některých prípadech dostatečně obecná anebo efektivní pro výpočet nově definované funkce strukturální rekurzí. Ukážeme si varianty pro tyto prípady:

1. hodnotu nové funkce nelze definovat pro prázdný seznam,
2. prvky seznamu chceme zpracovávat zleva,
3. úloha není dobře rozložitelná, a proto potřebujeme pracovat s obecnejšími průběžnými strukturami, ze kterých na konci spočítáme výsledek.

Prvý prípad je vhodný např. pro definici funkce *mins* pro výběr minimálního prvku ze seznamu. Z prázdného seznamu nelze vybrat nejmenší prvek, a proto změníme rekurzivní strukturu. Funkce $foldr1$ nepřipouští na vstupu prázdný seznam a bude hlásit chybu. Na rozdíl od $foldr$ proto nemá funkcionální parametr pro zpracování prázdného seznamu, ale pouze parametr pro zpracování vnitřních prvků, který má více omezující typ. Následná definice *mins* je přímočaré zobecnění binární funkce *min*.

```
foldr1 :: (a->a->a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
foldr1 _ []  = error "foldr1: empty list"

mins :: (Ord a) => [a] -> a
mins xs = foldr1 min
```

Poznamenejme, že *min* je funkce ze třídy Ord pro určení minima dvou prvků. Dále je vhodné si uvědomit, že typ výsledku je stejný jako typ prvků seznamu. To není podstatné omezení, protože vstupní seznam můžeme zpracovat pomocí map. Schéma definice by bylo $foo\ xs = foldr1\ f(map\ a\_to\_b\ xs)$.

Jiná varianta řešení tohoto problému je rozšířit typ výsledku tak, aby zahrnoval únikovou hodnotu. Pro tento případ lze použít typ $Maybe\ Int$, kde hodnota $Nothing$ se vrací pro prázdný seznam.

Druhá varianta je funkce *foldl* pro zpracování prvků seznamu zleva. Zpracování zprava může být v některých prípadech neefektivní. Je to vidět na funkcích *sum*, *length* a *reverse*. Ve všech případech při vyhodnocování vznikne zásobník aktivních volání funkcí uměrný délce vstupního seznamu, tj. pro výpočet je potřebná lineární paměť.

Změnou definic funkcí na zpracování zleva dostaneme efektivnější kód. Volání $foldl \oplus e\ xs$ počítá výraz $(\ldots((e \oplus x_1) \oplus x_2) \ldots \oplus x_n)$, kde xs je $[x_1, x_2, \ldots, x_n]$. Operace $\oplus$ ve výrazu asocionaná je doleva. Parametr e je akumulační parametr, který sbírá informace o dosud zpracované části seznamu.

```
foldl :: (b->a->b) -> b ->[a] -> b
foldl f e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

sum = foldl (+) 0
length = foldl (\y x->y+1) 0
reverse = foldl (flip (:)) []
```

Ve všech případech jsme využili skutečnost, že funkce pro zpracování prvku je asociativní a proto lze pořadí vykonávání operací změnit.

Skutečnost o efektivitě vyhodnocování je ještě trochu složitější a zatím mimo rozsah tohoto textu. Technicky vzato, v případě $reverse$ jsme si pomohli, protože časová efektivita se zlepšila z kvadratické na lineární. Výsledný seznam se buduje zevnitř a teda nové prvky se přidávají na začátek. Funkcionální parametr je $(flip(:))$, který z klasického konstruktoru seznamů vytvoří prohozením argumentů požadovanou funkci správného typu.

V případě sum a $length$ líné vyhodnocování způsobí odložení vyhodnocení druhého argumentu, i když ten ve skutečnosti vyhodnotit lze (na rozdíl od vyhodnocování výsledku ve $foldr$). Zde uvedené definice by proto měli opět lineární paměťovou spotřebu. Tyto dvě funkce jsou definovány pomocí varianty $foldl'$, definované v `prelude`, která akumulační parametr vyhodnocuje hned. Technické prostředky, jak toho dosáhnout, budou možná popsány v koutku pro hackery.

Analogicky k $foldr1$ je v `prelude` definovaná funkce $foldl1$ pro svíjení neprázdného seznamu zleva.

Třetí varianta je použití svíjení v situaci, kdy úloha není přímočaře rozložitelná. To znamená, že z prvního prvku seznamu a výsledku pro zbytek seznamu nelze spočítat výsledek pro celý seznam. V takovém případě musíme použít bohatší průběžné struktury, které výpočet umožní a na konci z nich extrahovat pouze požadované hodnoty. Tuto techniku lze použít na všechny minulé varianty funkce $fold$.

Příklad takové nerozložitelné funkce je průměr. První řešení má nevýhodu, že vstupní seznam se prochází dvakrát. Svinutí je schováno ve funkcích sum a $length$.

```
prumer :: [Float] -> Float
prumer xs = sum xs / fromInteger (length xs)
```

Pro jednoprůchodový výpočet zavedeme dvojici akumulátorů, které budeme upravovat společně. Závěrečná hodnota se získá vydělením čísel z akumulátorů.

```
prumer xs =
  let (sum,len) =
        foldl (\(s,l) x->(s+x,l+1)) (0,0) xs
    in sum/len
```

Unfold pro seznamy. Z imperativího programování je známo použití iterátoru pro procházení (nějaké reprezentace) seznamu. Pro implementaci iterátoru potřebujeme tři funkce $done$, $next$ a $first$ pro testování konce seznamu (anebo existence dalšího prvku), posunu na další prvek a vrácení aktuálního. Pro zachycení průbežně se měnící informace o zbylých prvcích se používá stav vracený funkcí $next$. Jednotlivé prvky seznamu jsou vraceny postupně jako výsledky funkce $first$.

Ve funkcionálním programování můžeme analogickou metodu použít pro budování seznamu jako výstupní datové struktury. Vzhledem k tomu, že nemáme stavové proměnné, vracené hodnoty nebudou postupné hodnoty v nějaké proměnné, ale konkrétní prvky ve vytvářeném seznamu. Stav, tj. interní informace o ještě nezpracovaných prvcích, se udržuje v nějaké datové struktuře.

Funkci pro vytváření výstupního seznamu nazveme `unfold`. Tato funkce volá opakovaně test ukončení `done` a pokud lze pokračovat ve výstavbě výsledku, vytvoří pomocí funkce `step` dvojici obsahující první prvek výstupu a reprezentaci zbytku. Tyto funkce jsou parametry `unfold`.

```
unfold done step x =
    if done x
      then []
      else let (y,ys) = step x
            in y : unfold ys
```

Příklad. Třídění výběrem je aplikace unfold.

```
sort xs = unfold (==[]) find_min xs
  where
    find_min [x] = (x,[])
    find_min (x:xs) =
      let (y,ys) = find_min xs in
        if x<y then (x,y:ys) else (y,x:ys)
```

Zobecnění unfold. Vzhledem k tomu, že reprezentace výstupní datové struktury je explicitní, nejsme při jej vytváření omedzení lineární časovou posloupností, tj. výsledek nemusí byť seznam. Funkce analogické unfold můžeme navrhnout i pro jiné výstupní datové struktury.

Příklad. Quicksort. Rekurze v quicksortu má tvar binárního stromu. Quicksort můžeme napsat jako vytváření tohoto stromu a jeho následné zpracování. Proto zavedeme `fold` a `unfold` pro binární stromy.

```
foldbt fnil fnode bt = f bt
  where f Bnil = fnil
        f (Bnode l x r) =
           fnode (f l) x (f r)

unfoldbt done step s = f s
  where f s = if done s then Bnil
              else let (l,x,r) = step s in
               Bnode (f l) x (f r)

qsort xs =  unfoldbt (==[]) split
         . foldbt [] (\l x r-> l++[x]++r)
  where
    split (x:xs) = ([y|y<-xs,y<x],
                    x,
                   [y|y<-xs,y>=x] )
```

Cvičení. Popište strukturu mergesortu a naprogramujte ho analogickým způsobem.

Programování vyšších řádů pro zpracování celých datových struktur.

V imperatívních programovacích jazycích je podobnost mezi strukturou složených příkazů strukturou dat, které zpracovávají. Složkám příkazů odpovídá složka datové struktury. Například for cyklus při zpracování pole zpracovává jeden prvek pole v jednom průchodu těla. Tento vztah mezi strukturou příkazu a dat zobecníme na jiné datové struktury.

Pro zpracování uživatelské datové struktury chceme definovat vhodné řídící struktury. Při tom můžeme vhodně použít programování vyššího rádu. Složkám příkazů odpovídají funkcionální parametry. Takto naprogramujeme parametrické funkce pro zpracovávání celých struktur. Příklady takových funkcí jsou `fold` a `map` pro různé datové struktury.

Použití funkcí pak odpovídá nějakému složenému příkazu, přičemž tělu odpovídajíci funkcionální parametry se volají pro zpracování částí datové struktury.

Vytvořené funkce zpracovávaní konkrétní strukturu a jsou polymorfní vzhledem k jejím složkám. Pro konkrétní typový konstruktor `T` má funkce typ `(` *funkc. parametry* `) -> T a -> b`. Jiný typový konstruktor je ošetřen jinou funkcí s jiným typem. V rozšíření funkcionálního programování nazývaném *polytypické programování*[1] lze popsat algoritmus jednou obecně a systém odvodí jeho konkrétní podobu pro daný typ (anebo obecný popis interpretuje).

V Haskelu lze pomocí typových tříd a konstruktu `deriving` odvodit funkce pro třídy Eq, Ord, Read, Show, ... ??

4 Uživatelsky definované typy

Kromě předdefinovaných datových typů si může uživatel definovat vlastní datové typy. ...

Definice některých typů a operace s nimi nejsou součástí jádra systému, ale jsou předdefinovány v souboru `prelude.hs`, který se natahuje při inicializaci systému. Tento soubor je textový zdrojový soubor a je ho možno změnit (rozšířit).

4.1 Typ `Maybe`

Typ `Maybe a` s jedním typovým parametrem `a` přidáva k tomuto typu jednu speciální hodnotu. Tato hodnota se používá pro zachycení výjimky nebo chyby. Definice typu je

```
data Maybe a = Nothing
             | Just a
```

Přidání speciální hodnoty tímto způsobem má dvě výhody. Nemusíme vyhradit z původního typu hodnotu a teda nezmenšujeme rozsahu typu. Druhá výhoda je, že toto rozšíření je použitelné pro všechny typy a proto operace s typem `Maybe` jsou polymorfní.

Jako příklad použití si ukážeme funkci `lookup`. Tato funkce vyhledává v seznamu podle klíče a vrací hodnotu s/združenou s klíčem. Pokud klíč nenajdeme, vracíme hodnotu `Nothing`.

```
lookup :: a -> [(a,b)] -> Maybe b
lookup _key [] = Nothing
lookup  key ((k,v):xs) =
          if k==key then Just v
                    else lookup key xs
```

[1] polytypické programování: Jeuring, ...

Uvědomte si, jak se dvě výše popsané výhody projevily v této funkci.

Typ `Maybe a` je druh monády. Operace jsou/budou rozebrány v příslušné kapitole.

Cvičení 1. Lze typový konstruktor `Maybe` aplikovat na typ `Maybe Int`. Pokud ano, jak budou vypadat hodnoty nového typu, pokud ne, proč?

Cvičení 2. Navrhněte rozšíření typu `Int` o dvě speciální hodnoty pro $+\infty$ a $-\infty$. Definujte na tomto typu uspořádání a aritmetiku (s chybovými hláškami).

4.2 Sjednocení typů: `Either`

Pracujeme v typově čistém jazyku a proto nelze implicitně přecházet mezi typy. Taktéž pokud chceme vracet hodnoty dvou typů, musíme z nich vytvořit jeden typ, se kterým již lze pracovat plnohodnotně. Pro vytváření (disjunktního) sjednocení zavedeme typový konstruktor `Either`.

```
data Either a b = Left a | Right b
```

Jako příklad použití uvedeme jednu z grafových operací, sumu grafů. Nově vytvořený graf má vrcholy vzniklé sjednocením vrcholů původních grafů a hrany v původních grafech zůstanou a navíc se přidají všechny hrany mezi vrcholy z různých grafů. Pokud původní grafy měli vrcholy typů `v1` a `v2`, pak typ vrcholů výstupního grafu je `Either v1 v2`. Grafy jsou reprezentovány jako dvojice vrcholů a hran.

```
sumagr :: ([a],[(a,a)])->([b],[(b,b)]) ->
         ([Either a b],[(Either a b,Either a b)])
sumagr (v1,h1) (v2,h2) =
        ([Left x1|x1<-v1]++[Right x2|x2<-v2],
         [(Left x, Left y)|(x,y)<-v1]++
           [(Right x,Right y)|(x,y)<-v2]++
           [(Left x, Right y)|x<-v1,y<-v2] )
```

Tento typ lze taky použít například v situaci, kdy na vstup dostávame data dvou (nebo několika) různých typů, případně jednoho druhu, ale ve dvou různých formátech. Pak konstruktory `Left` a `Right` slouží pro odlišení skutečného formátu.

4.3 Binární stromy

Binární vyhledávací stromy s vnitřními vrcholy jsou definovány pomocí typového konstruktoru `Bt/1`.

```
data Bt a = Bnil
          | Bnode (Bt a) a (Bt a)
```

Na stromech lze vykonávat obvyklé operace. Typ parametru může být složený a obsahovat i klíč i data, například jako dvojici. Pro převod mezi uspořádaným seznamem a stromem zavedeme funkce `leafs`, která pozbírá listy stromu, a `mk_bt`, která vytvoří strom.

Ve funkci `leafs` se používá pattern matching na uživatelsky definovaném typu. Vlastní zpracování probíhá rekurzivně, přičemž mezivýsledky se spojují funkcí `++` a datovým konstruktorem `:`.

```
leafs :: Bt a -> [a]
leafs Bnil = []
leafs (Bnode l x r) = leafs l ++ (x: leafs r)
```

Při výrobě stromu funkce `mk_bt` postupuje dělením vstupního uspořádaného seznamu. Při dělení se použijí (již známé) funkce `take` a `drop`, které vrací počáteční nebo koncovou část seznamu. Při konstrukci stromu se používají konstruktorové funkce pro typ `Bt`, které byly automaticky vytvořeny pro oba konstruktory `Bnil` a `Bnode` na základě deklarace typu `Bt`.

```
take :: Int->[a]->[a]
drop :: Int->[a]->[a]
mk_bt :: [a]->Bt a
mk_bt xs
  | xs == []  = Bnil
  | otherwise = Bnode (mk_bt (take k xs))
                      y
                      (mk_bt ys)
  where
    k = n `div` 2
    y:ys = drop k xs -- musí závorky y:ys??
```

Analogicky jako pro seznamy můžeme pro binární stromy vytvořit varianty funkcí `mapBt` a `zipWithBt`. První funkce projde binární strom a aplikuje operaci danou parametrem `f` na všechny prvky stromu. Druhá funkce prochází synchronizovaně dva stromy a vytváří jeden strom, který vzniká aplikováním funkce `f` na odpovídající si vrcholy, pokud existují. Pokud některý vrchol již neexistuje, tj. jeden ze stromů skončil dřív, výsledek je prázdný strom `Bnil`.

```
mapBt :: (a->b) -> Bt a -> Bt b
mapBt _f Bnil = Bnil
mapBt  f (Bnode l x r) =
  Bnode (mapBt f l) (f x) (mapBt f r)

zipWithBt :: (a->b->c) -> Bt a->Bt b->Bt c
-- zipWithBt f Bnil Bnil = Bnil
zipWithBt f (Bnode lx x rx) (Bnode ly y ry) =
     Bnode (zipWithBt f lx ly)
           (f x y)
           (zipWithBt f rx ry)
zipWithBt f _               _               =
     Bnil
```

```
zipBt :: Bt a -> Bt b -> Bt (a,b)
zipBt = zipWithBt (\x y->(x,y))
--zipBt x y = zipWithBt (\x y->(x,y)) x y
```

Poslední klauzule funkce `zipWithBt` ošetřuje ten případ, kdy aspoň jeden ze stromů skončil. To znamená, že jsou buď oba prázdné anebo nemají stejný tvar. Kvůli symetrii uvádíme v komentáři klauzuli ošetřující dva prázdné stromy, ale její činnost je zahrnuta v poslední klauzuli. Uvědomte si, že úplný rozbor případů bez použití `_` by vyžadoval čtyři klauzule.

Varianta funkce `zipBt` pro stromy, která tvoří z dvojice stromů strom dvojic, vznikne voláním `zipWithBt` s vhodným funkcionálním parametrem. Protože v této definiici jsou jména stromů ve stejné pořadí ve stejných argumentech, nemusíme je psát. Funkce `zipBt` by s explicitní rekurzí byla delší.

```
zipBt1 :: Bt a -> Bt b -> Bt (a,b)
zipBt1 (Bnode lx x rx) (Bnode ly y ry) =
           Bnode (zipBt1 lx ly)
                 (x,y)
                 (zipBt1 rx ry)
zipBt1 _               _               = Bnil
```

Poslední příklad této části je "roznásobení" stromu. Máme dán binární strom se seznamy hodnot. Chceme vyrobit seznam stromů s jednotlivými hodnotami, přičemž každý strom bude obsahovat jinou kombinaci hodnot a bude v seznamu právě jednou. Tato úloha je zobecnění kartézskeho součinu, který použijeme jako funkci `cart`.

```
roznasBt :: Bt [a] -> [Bt a]
roznasBt Bnil = [Bnil]
roznasBt (Bnode l xs r) =
  map (\(l,(x,r))->Bnode l x r)
    (cart (ls, (cart (xs,rs)) )) -- OK
    -- (ccart ls (ccart xs rs) ) -- xx
  where ls = roznasBt l
        rs = roznasBt r
        ccart x y = cart (x,y) --  typy??
```

Stručnější způsob s využitím syntaxe pro seznamy

```
roznasBt1 :: Bt [a] -> [Bt a]
roznasBt1 Bnil = [Bnil]
roznasBt1 (Bnode l xs r) =
  [Bnode l1 x1 r1 | l1<-roznasBt1 l,
                    x1<-xs,
                    r1<-roznasBt1 r]
```

Testovací data lze vytvořit pomocí konstruktorů `Bnil` a `Bnode` typu `Bt`.

```
bt1 = Bnode (Bnode Bnil [1,2,3] Bnil)
            [11,12,13]
            (Bnode Bnil [21,22,23] Bnil)
bt2 = Bnode bt1 [31] bt1
```

– test: zipWith (++) bt2 bt1

.

.

5 λ–abstrakce

...

λ-abstrakce dovoluje definovat vlastní funkce. Funkce prvního řádu, které dostávají jako parametry hodnoty, počítají hodnoty výrazů v závislosti na aktuálních parametrech. Protože funkcionální parametry jsou části kódu, tak funkce vyššího řádu jsou parametrickými příkazy. Rozšíření jazyka je ale pouze z hlediska funkčnosti, ne z hlediska syntaxe. Jediné přidané "klíčové" slovo je název funkce, za kterým následují předávané parametry.

...

5.1 Postupnosti

Hm.

5.2 Nekonečné struktury

Hm. Dál.

6 Typové třídy

Hm. ... od ML

Příklad. Standardní třídy. Konstrukt deriving.

Třída `Functor`. Do

Grafové algoritmy. Vhodný nízký interface pomocou t.t., ... ??

7 Příklady

... stavba trie

... Huffman

.. zásobníkový stroj: s procedúrami, s haldou ; uvod do abstraktních strojů

... ocenené rovnosti (řešení koreferencí na entity)

... data mining: extrakce vzorů ze sekvencí; pro husté, řídké reprezentace; extrakce sekvence, dvojice (neusp., and), or, xor

... elektronicky: redukce grafu

... oceňování stavu (rubikovy kostky) při prohl. prostoru: různé prvky jsou ruzně oceněny -¿ použití zip a fold, navíc ocenění stejných správných sousedících kostek (i rohů); jednodušší pro uspořádání 15.

8 Techniky

Dodělat: akumulátor, stav, ...

vyšší: dyn. prog., Grafy a pod.; tabulky a fold; unifikace, antiunifikace

parsery;, metainterprety; nekonečné d.s.; fold, fold1 a fold s pamětí vstupu, a unfold; monády (!); podmínky

8.1 Pomocné funkce !!

Při kompozičním programování někdy potřebujeme pomocné funkce, které při skládání dají do souladu výsledky první (volané) funkce s předpoklady druhé (volající) funkce. Tady si Pak ...

Zavedení `flip`.

Funkce `curry` a `uncurry`. Funkce se obvykle píššou a používají tak, že argumenty vstupují samostatně. To je curryovaná/curryfikovaná forma. Pokud funkce vrací složenou hodnotu, tak ji musí vrátit v dvojici, obecně n-tici. Pro zpracování takových hodnot je přirozená uncurryfikovaná forma funkce, která dostává jeden argument a ten obsahuje všechny parametry. Pro přechod mezi dvěma formama funkce se používají funkce `curry` a `uncurry`.

```
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)
```

Příklad. Funkce `map2`. Již vysvětlená funkce `map` prochází seznam a aplikuje na každý prvek danou funkci jednoho argumentu. Analogicky chceme definovat funkci `map2`, která prochází synchronizovaně dva seznamy argumentů a aplikuje na odpovídající si prvky danou funkci dvou argumentů. Pokud jsou seznsmy růžě dlouhé, zkrátí se výsledek na délku kratšího.

```
map2 :: (a->b->c) -> [a]->[b]->[c]
map2 _f [] _  = []
map2 _f _  [] = []
map2  f (x:xs) (y:ys) = f x y : map2 f xs ys
```

Pomocí kompozičního programování můžeme využít funkce `zip`, která vydává seznam dvojic. Tento seznam zpracujeme funkcí `map`. Problém nastává v typu parametrů. Výsledky `zip` jsou v necurryfikovaném tvaru, `f` je v curryfikovaném tvaru, a proto `map` musí dostat jako funkcionální argument necurryfikovanou verzi `f`, tj. `uncurry f`.

```
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f xs ys = map (uncurry f) (zip xs ys)
```

Pokud bychom funkci `flip` neznali, můžeme použít jednoúčelovou λ-abstrakci.

```
zipWith_a :: (a->b->c) -> [a]->[b]->[c]
zipWith_a f xs ys =
        map (\(x,y)-> f x y) (zip xs ys)
```

Skládání funkcí. Pro pohodlnejší zápis bylo zavedeno skládaní funkcí. Funkční symbol `.` je zaveden jako infixní operátor.

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

Pomocí skládání můžeme tvořit složitejší funkce bez explicitního vypisování parametrů a závorek. Volání `map (f.g)` je kompaktnejší než použití anomymní funkce ve `map (\x->f(g x))`. Navíc se při vícenásobném skládání nekumulují závorky, protože tečka je infixní operátor.

Tisk budeme realizovat pomocí typu ShowS, který určuje tiskovou funkci. Ta před daný zbytkový řetězec přidá daný strom převedený do výstupního tvaru. Tímto způsobem tisku docílíme, že místo opakovaného skládání řetězců manipulujeme s funkcemi.

```
type ShowS = String -> String
showBt :: Show a => Bt a -> ShowS
showBt Bnil = ("-"++)
showBt (Bnode l x r) =
  ('<':).(showBt l ++).
  ('|':).(show x ++).
  ('|':).(showBt r ++).
  ('>':)
```

Voláním

```
showBt (Bnode Bnil 1 (Bnode Bnil 2 Bnil)) ""
```

dostaneme výstup `<-|1|<-|2|->>`.

Nevýhoda skládání funkcí je, že se aplikují na argument zprava doleva. To lze změnit zavedením další funkce, která bude skládat v opačném pořadí.

Zavedení `>.>`. Funkce skládání `(.)` má nevýhodu, že se jednotlivé funkce používají odzadu. Pokud chceme aby použití funkcí odpovídalo jejich zápisu zleva doprava, zavedeme funkci `>.>`. Je to varianta skládání funkcí, ale funkce se skládají v opačném pořadí. Funkci zavedeme jako levě asociativní infixní operátor.

```
infixl 9 >.>
>.> :: (a->b) -> (b->c) -> (a->c)
f >.> g = g . f   -- = g (f x)
```

Tato funkce a její operátorový zápis sdílejí výhody operátorů, tj. že zápis je přehlednější a nemusíme pracně počítat závorky za argumentem poslední vnořené funkce.

Zavedení `$`

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

Funkce `($)` je asociovaná doprava a proto následující zápisy jsou ekvivalentní. Zápis s použitím `($)` je nejpřehlednější (:-)).

- `map f $ filter g $ map h x`
- `map f (filter g (map h x))`
- `(map f . filter g . map h) x`
- `(map h >.> filter g >.> map f) x`

Opakované skládání funkcí. Pro libovolnou funkci `f::a->a` můžeme definovat opakované složení. Napíšeme funkci, které dáme v parametru n požadovaný počet opakovaní a výsledkem bude složená funkce. Pro $n = 0$ má smysl dodefinovat výsledek skládání jako identitu, tj preddefinovanou funkci `id::a->a` (z `prelude`).

```
f_iter :: Int -> (a->a) -> a->a
f_iter 0 f = id
f_iter (n+1) f = f >.> f_iter n f

f_iter :: Int -> (a->a) -> a->a
f_iter1 0 f = id
f_iter1 (n+1) f = f . f_iter1 n f
```

Skládání funkcí je asociativní, proto na závorkovaní z hlediska výsledku nezáleží. Funkci jsme zapsali dvěma způsoby. Z hlediska vyhodnocování je první verze paměťově efektívnější, protože si vždy rozbalí pouze jednu uroveň rekurze a tu vyhodnotí. Ve druhé verzi musíme rozvinout všech n složení funkce f než začneme počítat.

8.2 Zpracování struktury

Zpracování rekurzivních struktur pomocí strukturální rekurze je jednoduchý a přirozený způsob programování. Pokud abstrahujeme od konkrétních operací, pak lze se strukturou určitého typu sdružit operaci průchodu, fold, která dostáva kromě vlastní struktury příslušného typu ještě konkrétní operace, které se mají provádět.

Operace odpovídají konstruktorům typu a popisují, jak se má zpracovat struktura s příslušným konstruktorem v kořenu. Předpokládá se, že rekurzivní části struktury byly zpracovány rekurzivně.

Cvičení. Lze obecně použít fold na skládání funkcí následujícím způsobem?

```
fold (.) id [f_1, f_2, ... , f_n]
```

explicitně: nepřímá rekurze, omezení na regulární datové typy; nezávislost na pořadí (obvykle, ale foldl), speciální případy: fold1;

Tj. fold; Rozšir: polytyp. progr.

8.2.1 Unfold

Struktura, kterou zpracováváme, nemusí mít explicitní rekurzivní formu. Nemusí být zapsána pomocí explicitních konstruktorů (tagů) jako term nějakého uživatelského typu. Do této podoby ji lze vygenerovat. Jako příklad mohou slouzit čísla, která lze převést na unární reprezentaci (pomocí destruktoru). Jiný příklad jsou parsery, které dodají strukturu lineárnímu textu.

– rek. struktura čísel, vs. seznam čísel, vs. stav a iterátor; navíc reprezentace podle O'Keefeho: backtrackingová a posl. hodnot v proměnné. Převody?

8.3 Stavové programování

Technická pomůcka při prevodu rekurze na iteraci. Mám zásobník pod vlastní kontrolou.

Príklad: depth – hloubka binárního stromu, 7 způsobů ...

.. spíš rozšiřující téma, anebo odborné. Jak vygenerovat polytypicky posloupnost stavů, která bude krátká (z každé úrovně je tam jen jeden uzel), typově správná (hm) a přitom jednoduchá na popis. (jsou monády podobný případ, až na krátkost popisu?)

8.4 Kompoziční programování

Předpoklady:

Ve funkcionálním programování lze použít i jiný způsob programování než shora – dolů, tj rozklad na podproblémy a jejich ošetřování. Program můžeme skládat z připravených částí, které obsahují základní zpracování. Tento způsob vyžaduje dva předpoklady: máme dostatek různorodých funkcí (stavebních kamenů) v podobě knihovny a přitom jsou funkce dostatečně flexibilní. Této vlastnosti se dosahuje pomocí funkcionálních parametrů, protože ty umožňují předání libovolného kódu (vhodného typu) a jeho následné použití.

Příklad. Odebrání společné předpony dvěmi řetězcům. Klasické řešení rekurzí prochází řetězce a odebírá stejné znaky ze začátku. Kompoziční řešení generuje synchronizovaně seznam dvojic přípon a vrátí první dvojici, která má různé začátky.

```
strip_sp1::[a]->[a]->([a],[a])
strip_sp1 [] y = y
strip_sp1 x  [] = x
strip_sp1 (x:xs) (y:ys) = if x==y then strip_sp1 xs ys
                                   else (x:xs,y:ys)
strip_sp2 x y =
  head (filter same_firstc (zip (iterate tail xs)
                                (iterate tail ys) )
        where
        same_firstc ([],  _  ) = True
        same_firstc (_    [] ) = True
        same_firstc (x:_, y:_) = x==y
```

Druhý způsob v tomto případě využívá lazy vyhodnocování, protože volání `iterate tail xs` končí po vyprázdnění seznamu vždy chybou.

8.4.1 Kompozice funkcí

Při kompozičním programování se používají dva způsoby skládání funkcí. První je klasické skládání, kdy výsledek jedné funkce je argumentem další. Druhý způsob je vkládání funkcionálních parametrů, kdy se obecná funkce přizpůsobí pro konkrétní použití. Pro tento druhý způsob programování potřebujeme obecné funkce a pomocné funkce, které slouží jako interface při skládání. Časté nebo typické funkce je vhodné pojmenovat a zařadit do knihovny, pro jednoúčelové funkce je k dispozici λ–abstrakce

Shrnutí: ... u každé kapitoly, a příklady, ...

... a jeho ladění, ?patří sem iterátory?

8.5 Prohledávaní

.. generuj a testuj, and-or grafy; převod or-sekvence na and-sekvenci: seznam řešení místo backtrackingu; seznam možností a filtrace; zásobník větvení;

Kmprehence seznamů jsou speciální prípad iterátorů (a monád?)

8.5.1 Iterátory

Pri prohledávání nějakého prostoru si nemusíme pamatovat minulé větve, pokud je dokážeme zrekonstruovat z aktuálního stavu. Idea tohoto způsobu je, že si pamatujeme aktuální stav a pak máme k dispozii inkrementální metodu, jak aktuální stav změnit na následující.

Příklady: kart. součin, permutace, kombinace (s/bez opak) - už bylo, rozklady na součet čísel - dané nebo lib. délky, ...

omezení: nehodí se k prohledávání s opakováním, resp. opakované stavy je nutno přeskakovat v rámci inkrementálního kroku. Jiná možnost je pamatovat si poslední stav a směr příchodu , např. jako předposlední stav. Tato informace pak stačí k přechodu stromu do hloubky.

... iterátory a stavové programování, jak jednoduše zapsat stavy při výpočtu?, ...

8.6 Lazy výpočty

...

8.6.1 Nekonečné seznamy

Díky línému vyhodnocování lze pracovat s potenciálně nekonečnými strukturami. Věnovat se budeme hlavně seznamům. Měli by jsme si uvědomit, že vypsat celou nekonečnou strukturu nemůžeme a proto nakonec musíme odebrat z výsledné struktury (počáteční) konečnou část.

Při práci s nekonečnými seznamy potrebujeme tyto seznamy generovat a dále z nich potřebujeme odebírat konečnou část. Pro generaci jsou užitečné funkce

```
repeat :: a->[a]
repeat x = x : repeat x

cycle :: [a]->[a]
cycle xs = xs ++ cycle xs

iterate :: (a->a)->a->[a]
iterate f x = x:iterate f (f x)
```

Alternativní definice *iterate = x:map f (iterate f x)* je méně efektivní, protože n-tý každý prvek ve výstupu vyžaduje n aplikací funkce f, co dává celkem kvadratický počet volání f v závislosti na počtu zpracovaných prvků.

Při práci s nekonečnými seznamy je nutno používat "líné" funkce, které vracejí výsledek postupně, po zpracování části vstupu. Pokud bychom použili tzv. striktní funkci, která požaduje celý vstup, tak se výsledku nedočkáme. Program se zacyklí a případně mu přeteče zásobník. Dříve uvedené !! funkce `filter`, `map`, `zip` ... jsou v tomto smyslu "líné" a proto umožňují zpracovávat i nekonečné seznamy.

Druhý druh funkcí ubere z nekonečné struktury pouze konečnou část. ... Vhodná funkce je *take n*.

Past. Volání `[x|x<-[1..], x<3]` vrací hodnotu $1 : 2 : \perp$ místo očekávané $1 : 2 : []$, tj. $[1, 2]$. Důvodem zacyklení je, že systém testuje postupně všechny další čísla a dopředu netuší, že už žádné další nenajde. V tomto a podobných případech je nutno zaintegrovat podmínku do generátoru, pokud to jde.

Příklad. Erathostenovo síto apod ...

Nekonečné nemusí být jen seznamy, ale libovolné (rekurzivní) struktury.

```
haldaN n = Bt (halda 2*n) n (halda 2*n+1)
```

Ubrání konečné části může mít podobu useknutí stromu v určité hloubce nebo useknutí podstromů s vrcholem nad daným prahem.

Příklad. Nekonečná jednotková matice - funkce *maticeE*. První řádek matice je definován jako *1:repeat 0*. Další rádky se liší novou nulou vepředu. Změnová funkce je teda `(0:)` a použijeme ji v *iterate*.

```
maticeE :: [[Integer]]
maticeE = iterate (0:) (1: repeat 0)
```

Pro odebrání matice $m.n$ napíšeme funkci *takeMat*

```
takeMat m n mat = take n $ map (take m) mat
```

9 Pokročilé techniky

9.1 Příklady

Filtrace signalu (1D), filtrace obrázků (2D)

Hledání epizod.

Třídy ekvivalence.

Stavové přechody. Pro popis automatu.

Slévání precedencí. ...

Funkcionální update. Při budování dat. str. nemusím posílat měněná data, ale můzu poslat update funkci. Obecně, parametrizace přechodů nebo zpracování struktur funkcemi. Je to asi univerzálnější, ale možná to má špatný vliv při líném vyhodnocování. *Je to možná projev jednoúčelových continuation.*

Pokračování pro datové struktury. Při budování datové struktury (seznamu) se na konci přidají ladící informace, které jsou do funkce předány pomocí f. parametru, analogicky pokračování. **důsledek: ladění.** Při ladění (přinucení k vyhodnocování) je možno výstupní proud (seznam) strukturovat jako "Either výstup ladící_informace". Případně smíchané v jednom.

Kompoziční styl programování. Někdy se těžko zdvihá přes funkce. Jsou nutné zdvihové predikáty, analogické map, ale i pro víc argumentů. Pokud nemáme Konstruktorové třídy, resp. polytypy, operace pro zdvih jsou pro jednotlivé typy, analogicky polytypům, ale v záv. na konkr. typu.

Zvedání operací přes funkce

Parametrická inicializace. Při budování NTree (?) je v update potřeba inicializovat nové vrcholy. Update potřebuje vědět, čím má inicializovat hodnoty. Tzn. hodnoty vnořeného typu se přirozeně projeví při přirozené operaci hlavního typu. Návrh řešení: parametrické inicializační hodnoty (nastavené typicky jako konkrétní konstruktor), které se explicitně uvedou. Pojmenování hoodnot je explicitní označení toho, že táto část kódu závisí na typu hodnoty. Proto je nutno od typu resp. konstruktoru abstrahovat.

FP: Práce s celými d.s., ne s jednotlivými částmi.

9.2 Zbytky :-)

... case studies, ...

...monády, i/o?

... typové triedy

... slovník: symbolické odkazy + lookup

10 Soubor `prelude.hs`

...

11 Technické triky

12 Meta

struktura: popis situace, v čem je problém, návrh řešení, důsledky

ciele: Vysvetlenie jazyka vs Vysvetlenie pouzitia, Progr. idiomy, Odlisnosti od imp. progr. a OOP ?!

Na konci kapitoly shrnutí, cvičné příklady. Dokument jako vykonatelný kód (tex=lhs). Na začiatku kapitoly: v této kapitole se naučíte ...

Příklady a varianty: ?

Programy: ? ! Typy k funkciám!!

Odlišovať syntaktickú rovinu od sémantickej (čo som tým myslel?). Vysvetlenie syntaxe a vysvetlenie programátorských obratov.

12.1 Výuka na cvičeních.

lazy and, binarne stromy: listify, zip, DC: roznásobenie

12.2 Cizí obsahy

Učebný text - kapitoly: Výrazy a definice; Výpočty; Typy; Datové struktury; Funkce vyššího řádu; Funkcionální abstrakce; Seznamy; Definice datových typů; Typové třídy; Vstup a výstup;.

Kniha: I Basic FP; Intro; Basic types and simple programs; Reasoning about programs; Data structures: lists; Reasoning about lists; II Abstraction; Generalization; Function as values; Classes in Haskell; Checking types; III Larger-scale programming; Algebraic types; Case study: Huffmann codes; ADT; Lazy programming; I/O; Program behaviour;

Gentle Introduction to HS98: Intro; Values, types and other, Polymorphyc types, User-def. types, Recursive types, Type synonyms, Built-in types arnot special, list comprehensions and arith. sequences; Functions, zlambda abstraction, Infix operators, sections, Fixity declarations, Functions are not strict, Infinite d.s., Error function; Case expr. and pattern matching, Pattern matching semantics, Example, Case expr., lazy patterns, Lexical scoping and nested forms; Type classes and overloading; Types, again, Newtype declaration, Field labels, Strict data constructors; I/O, Basic I/O operations, programming with actions, Exceptions handling, Files, channels, and handles, Haskell and imperative programming; Standard Haskell Classes, Eq and Ord, Enumeration, Read and Show, Derived instances; About Monads, Monadic classes, Built-in monads, Using Monads; Numbers, Numeric class structure, constructed numbers, nNumeric coercions and overloaded Literals, Default numeric types; Modules, Qualified names, Abstract data types, More features; Typing pitfalls, let-bound polymorphism, Numeric overloading, The monomorphism restriction; Arrays, Index types, Array creation, Accumulation, Incremental Updates, Ex: Matrix multiplication; The Next Stage

Literatura

[1] P. Hudak, J. Peterson, J. H. Fasel, A Gentle introduction to Haskell, X.1999, elektronicky, `http://www.haskell.org`

[2] S. Thompson, Haskell, The Craft of Functional Programming, Addison-Wesley, Harlow, England, 1996

[3] R. Bird, P. Wadler, Introduction to Functional Programming, Prentice Hall, 1988

Annotation: vysvětlení technik Haskellského programování.

A Dodatky

Inžiniersky bottleneck.