IB015 Neimperativní programování

Redukční strategie, Seznamy, Vstup-výstup

Jiří Barnat
Libor Škarvada

Redukční strategie

Redukce, redukční strategie

Redukční krok

- Úprava výrazu, v němž se některý jeho podvýraz nahradí zjednodušeným podvýrazem.
- Upravovaný podvýraz (**redex**) má tvar aplikace funkce na argumenty, upravený podvýraz má tvar pravé strany definice této funkce do níž jsou za formální parametry dosazené skutečné argumenty.

Redukční strategie

- Předpis, který určuje jaký podvýraz se bude upravovat v následujícím redukčním kroku.

Základní redukční strategie

Striktní redukční strategie

- Při úpravě aplikace $F\ X$ nejdříve úplně upravíme argument X. Teprve nelze-li už upravovat argument X, upravujeme výraz F. Až nakonec upravíme (podle definice funkce) celý výraz $F\ X$.
- Při úpravě výrazů tedy postupujeme **zevnitř**.

Normální redukční strategie

- Upravovaným podvýrazem je celý výraz; nelze-li takto upravit aplikaci $F\ X$, upravíme nejdříve výraz F, pokud to nestačí k tomu, abychom mohli upravit $F\ X$, upravujeme částečně výraz X, ale pouze do té míry, abychom mohli upravit výraz $F\ X$.
- Při úpravě výrazů tedy postupujeme **zvnějšku**.

Líná redukční strategie

Líná redukční strategie

- Normální redukční strategii, při níž si pamatujeme hodnoty upravených podvýrazů a žádný s opakovaným výskytem nevyhodnocujeme více než jednou.
- Využívá referenční transparentnost.
- Nelze aplikovat na výrazy s vedlejším efektem.

Haskell

- Používá normální redukční strategii.
- Též označováno jako **líné vyhodnocování**.

Příklady vyhodnocování s různou strategií

Definice funkce

- `cube x = x * x * x`

Striktní redukční strategie

- `cube (3+5)` $\leadsto$ `cube 8` $\leadsto$ `8 * 8 * 8` $\leadsto$ `64 * 8` $\leadsto$ `512`

Normální redukční strategie

- `cube (3+5)` $\leadsto$ `(3+5) * (3+5) * (3+5)` $\leadsto$ `8 * (3+5) * (3+5)` $\leadsto$ `8 * 8 * (3+5)` $\leadsto$ `64 * (3+5)` $\leadsto$ `64 * 8` $\leadsto$ `512`

Líná redukční strategie

- `cube (3+5)` $\leadsto$ `(3+5) * (3+5) * (3+5)` $\leadsto$ `8 * 8 * 8` $\leadsto$ `64 * 8` $\leadsto$ `512`

Praktická poznámka k Haskellu

Praktická realizace líného vyhodnocování

- Vzdálená ideální teoretické situaci.
- Ne vše lze líně vyhodnotit (výrazy s vedlejším efektem).
- V Haskellu garantováno pouze pokud je stejný podvýraz ve výrazu zaveden pomocí lokální definice.

Příklad

- `let z=3*3 in z + z + z`

Více info viz

- https://wiki.haskell.org/GHC/FAQ#Does_GHC_do_common_subexpression_elimination.3F
- https://wiki.haskell.org/GHC_optimisations#Common_subexpression_elimination

Vliv strategie na výsledek vyhodnocování

Pozorování

- Použitá strategie může ovlivnit chování programu.

Příklad 1

- Uvažme funkci `const`

```
const :: a -> b -> a
const x y = x
```

- Při striktním vyhodnocování dojde k dělení nulou

`const 2 (1/0)` ⇝ **ERROR**

- Při líném vyhodnocování k němu nedojde

`const 2 (1/0)` ⇝ 2

Vliv strategie na výsledek vyhodnocování

Příklad 2

- Uvažme funkci `undf`

```
undf x :: Int -> Int
undf x = undf x
```

- Striktní vyhodnocování následujícího výrazu vede k zacyklení

```
head (tail [undf 1, 4]) =
head (tail (undf 1 : 4 : [])) ⇝
head (tail (undf 1 : 4 : [])) ⇝
...
```

- Při líném vyhodnocování k zacyklení nedojde:

```
head (tail [undf 1, 4]) =
head (tail (undf 1 : 4 : [])) =
head (tail (undf 1 : 4 : [])) ⇝
head (4 : []) ⇝ 4
```

Obecné vlastnosti redukčních strategií

Churchova-Rosserova věta

- Výsledná hodnota ukončeného výpočtu výrazu nezáleží na redukční strategii: pokud výpočet skončí, je jeho výsledek vždy stejný.

Interpretace věty

- Churchova-Rosserova věta **nevylučuje různé chování** výpočtu při různých strategiích. Při některých strategiích může výpočet skončit, při jiných cyklit. Nebo je výpočet podle jedné strategie delší než podle jiné. Nikdy však **nemůže skončit dvěma různými výsledky**.

Obecné vlastnosti redukčních strategií

O perpetualitě

- Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M zacyklí, pak se tento výpočet zacyklí i s použitím striktní redukční strategie.

Interpretace věty

- Věta o perpetualitě říká, že z hlediska možnosti zacyklení výpočtu je striktní redukční strategie nejméně bezpečná. Když se při jejím použití výpočet nezacyklí, pak se nezacyklí ani při žádné jiné strategii.

Obecné vlastnosti redukčních strategií

O normalizaci

- Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M nezacyklí, pak se tento výpočet nezacyklí ani s použitím normální redukční strategie.

Interpretace věty

- Věta o normalizaci říká, že z hlediska možnosti zacyklení výpočtu je normální redukční strategie nejbezpečnější. To neznamená, že by se s jejím použitím výpočet zacyklit nemohl; z věty však plyne, že když se to stane a výpočet se i při normální redukční strategii zacyklí, pak se zacyklí i při každé jiné strategii.

Líné vyhodnocování

Jiný pohled

- Při použití líné/normální redukční strategie je výraz vyhodnocen až v okamžiku, kdy je potřeba pro další výpočet.
- Přístup, který jde nad rámec redukční strategie.

Příklady

- Líné čtení řetězce ze vstupu:

```
getContents :: IO String
```

- Líné vyhodnocování Boolovských operátorů v imperativních programovacích jazycích.

```
(True OR (1/0)) = True
(open(...) OR die)
```

– "umře"pokud `open` selže.

Líné vyhodnocování a nekonečné datové struktury

Nekonečné datové struktury

- Vyhodnocení výrazu až v okamžiku, kdy je potřeba pro další výpočet, umožňuje manipulaci s nekonečnými datovými strukturami.
- Příkladem nekonečné datové struktury je **nekonečný seznam**.

Příklad

- Seznam nekonečně mnoha jedniček:

```
jednicky = 1 : jednicky
```

- Vyhodnocení výrazu `jednicky` se zacyklí při každé strategii:

```
jednicky  ⇝ 1:jednicky  ⇝ 1:1:jednicky  ⇝ ···
```

- Ale je-li výraz `jednicky` podvýrazem většího výrazu, tak se jeho vyhocení při líné strategii nemusí zacyklit.

```
head jednicky = head jednicky  ⇝ head (1:jednicky)  ⇝ 1
```

Příklady definice nekonečných seznamů

Nekonečný rostoucí seznam všech přirozených čísel

- `nats = 0 : zipWith (+) nats jednicky`

	0	1	2	3	4	5	...	nats
+	1	1	1	1	1	1	...	jednicky
0	1	2	3	4	5	6	...	

Fibonacciho posloupnost

- `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`

		0	1	1	2	3	5	...	fibs
+		1	1	2	3	5	8	...	tail fibs
0	1	1	2	3	5	8	13	...	

Funkce generující nekonečné seznamy

Nekonečné opakování jednoho prvku

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

Nekonečné opakování seznamu

- ```
  cycle :: [a] -> [a]
  cycle x = x ++ cycle x
  ```

Opakovaná aplikace funkce

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

Příklady

Alternativní definice

- `jednicky = repeat 1`
- `jednicky = iterate (+0) 1`
- `jednicky = iterate (id) 1`
- `jednicky = cycle [1]`
- `nats = iterate (+1) 0`

Další příklady

- `take 10 (iterate (*2) 1)` $\leadsto^*$
- `take 5 (iterate ('a':) [])` $\leadsto^*$
- `take 10 (iterate (*(-1)) 1)` $\leadsto^*$
- `take 8 (cycle "Ha ")` $\leadsto^*$

Příklady

Alternativní definice

- `jednicky = repeat 1`
- `jednicky = iterate (+0) 1`
- `jednicky = iterate (id) 1`
- `jednicky = cycle [1]`
- `nats = iterate (+1) 0`

Další příklady

- `take 10 (iterate (*2) 1)` $\leadsto^*$ `[1,2,4,8,16,32,64,128,256,512]`
- `take 5 (iterate ('a':) [])` $\leadsto^*$ `["","a","aa","aaa","aaaa"]`
- `take 10 (iterate (*(-1)) 1)` $\leadsto^*$ `[1,-1,1,-1,1,-1,1,-1,1,-1]`
- `take 8 (cycle "Ha ")` $\leadsto^*$ `"Ha Ha Ha"`

Zápis seznamů

Zápis seznamu výčtem

Prostý výčet

- Zápis pomocí základních datových konstruktorů `(:)` a `[]`.

  `1:2:3:4:5:[]`
- Ekvivalentní zkrácený zápis (syntaktická zkratka pro totéž).

  `[1,2,3,4,5]`

Hromadný výčet

- Seznamy hodnot, které lze systematicky vyjmenovat (enumerovat) lze zadat tzv. **hromadným výčtem**.
- Seznamy zadané enumerační funkcí `enumFromTo`

  `enumFromTo 1 12` $\rightsquigarrow^*$ `[1,2,3,4,5,6,7,8,9,10,11,12]`

  `enumFromTo 'A' 'Z'` $\rightsquigarrow^*$ `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`
- Všechny uspořádatelné typy jsou enumerovatelné.

Enumerační funkce a syntaktické zkratky

Nekonečná enumerace

- Enumerovat lze i hodnoty typů s nekonečnou doménou.
- Hromadným výčtem lze definovat nekonečné seznamy.

```
nats = enumFrom 0
```

Enumerace s udaným vzorem

- Udáním druhého prvku lze definovat vzor enumerace:

take 10 (enumFromThen 0 2) $\leadsto^*$ [0,2,4,6,8,10,12,14,16,18]

enumFromThenTo 0 3 15 $\leadsto^*$ [0,3,6,9,12,15]

Přehled enumeračních funkcí a syntaktických zkratek

Enumerační funkce	Typ	Zkratka
`enumFrom m`	`Enum a => a->[a]`	`[m..]`
`enumFromTo m n`	`Enum a => a->a->[a]`	`[m..n]`
`enumFromThen m m'`	`Enum a => a->a->[a]`	`[m,m'..]`
`enumFromThenTo m m' n`	`Enum a => a->a->a->[a]`	`[m,m'..n]`

Intenzionální zápis seznamu

Intenzionální definice seznamu

- Prvky seznamu jsou generovány společným pravidlem, které předepisuje jak prvky z nějaké nosné množiny přepsat na prvky generovaného seznamu.
- Příklad: prvních deset násobků čísla 2

```
[ 2*n | n <- [0..9] ]
```

Obecná šablona

- `[ definiční_výraz | generátor a kvalifikátory ]`
- Za každý vygenerovaný prvek vyhovující všem kvalifikátorům se do definovaného seznamu přidá jedna hodnota definičního výrazu.
- Definiční výraz může a nemusí použít generované prvky.
- Kvalifikátory a generátory se vyhodnocují **zleva doprava**.

Kvalifikátory – Generátory

Generátor

- `nová_proměnná <- seznam` nebo `vzor <- seznam`
- Definuje novou proměnou použitelnou buď v definičním výrazu nebo v libovolném kvalifikátoru vyskytujícím se vpravo.
- Nová proměnná postupně nabývá hodnot prvků v seznamu.
- V případě použití vzoru, se vygeneruje tolik instancí, kolik prvků v seznamu odpovídá použitému vzoru.

Kombinace více generátorů

- Při použití více generátorů se generují všechny kombinace.
- Pořadí kombinací je dáno uspořádáním generátorů v definici.
- Nejvyšší váhu má generátor vlevo, směrem doprava váha klesá.

Kvalifikátory – Predikáty a lokální definice

Predikát

- Výraz typu `Bool` .
- Může využít proměnné definované od predikátu vlevo.
- Vygenerované instance, které nevyhovují predikátu, nebudou brány v potaz pro definici výsledného seznamu.

Lokální definice

- `let nová_proměnná = výraz`
- Definuje novou proměnou použitelnou buď v definičním výrazu nebo v libovolném kvalifikátoru vyskytujícím se vpravo.
- Výraz může využít proměnné definované vlevo.

Příklady 1/3

- `[ n^2 | n <-[0..3] ]`

  $\leadsto^*$ `[0,1,4,9]`

- `[ (c,k) | c<-"abc", k<-[1,2] ]`

  $\leadsto^*$ `[('a',1),('a',2),('b',1),('b',2),('c',1),('c',2)]`

- `[ 3*n | n<-[0..6], odd n ]`

  $\leadsto^*$ `[3,9,15]`

- `[ (m,n) | m<-[1..3], n<-[1..3], n<=m ]`

  $\leadsto^*$ `[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]`

- `[ (m,n) | m<-[1..3], n<-[1..m] ]`

  $\leadsto^*$ `[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]`

- `[ (x,y) | z<-[0..2], x<-[0..z], let y=z-x ]`

  $\leadsto^*$ `[(0,0),(0,1),(1,0),(0,2),(1,1),(2,0)]`

Příklady 2/3

- `[ replicate n c | c<-"xyz", n<-[2,3] ]`

  $\rightsquigarrow^*$ `["xx","xxx","yy","yyy","zz","zzz"]`

- `[ replicate n c | n<-[2,3], c<-"xyz"]`

  $\rightsquigarrow^*$ `["xx","yy","zz","xxx","yyy","zzz"]`

- `[ x^2 | [x]<-[[],[2,3],[4],[1,1..],[],[7],[0..]] ]`

  $\rightsquigarrow^*$ `[16,49]`

- `[ 0 | []<-[[],[2,3],[4],[0..],[],[5]] ]`

  $\rightsquigarrow^*$ `[0,0]`

- `[ x^3 | x<-[0..10], odd x ]`

  $\rightsquigarrow^*$ `[1,27,125,343,729]`

- `[ x^3 | x<-[0..10], odd x, x < 1 ]`

  $\rightsquigarrow^*$ `[]`

Příklady 3/3

Redefinice známých funkcí

- ```
  length :: [a] -> Int
  length s = sum [ 1 | _ <- s ]
  ```
- ```
  map :: (a->b) -> [a] -> [b]
  map f s = [ f x | x <- s ]
  ```
- ```
  filter :: (a->Bool) -> [a] -> [a]
  filter p s = [ x | x <- s, p x ]
  ```
- ```
  concat :: [[a]] -> [a]
  concat s = [ x | t <- s, x <- t ]
  ```

Nové funkce

- ```
  isOrdered :: Ord a => [a] -> Bool
  isOrdered s = and [ x<=y | (x,y) <- zip s (tail s) ]
  ```
- ```
  samohlasky :: String -> String
  samohlasky s = [ v | v <- s, v `elem` "aeiouy"]
  ```

Třídění prvků v seznamu – Quicksort

Úkol

- Napište funkci, která při aplikaci na konečný seznam uspořadatelných hodnot vrátí seznam těchto hodnot uspořádaných operátorem `<` .

Řešení

- Funkce `qSort` seřadí seznam hodnot vzestupně.
-
```
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (p:s) = qSort [ x | x<-s, x<p ]
              ++ [p] ++
              qSort [ x | x<-s, x>=p ]
```

Prvočísla – Eratosthenovo síto

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	3		5		7		9		11		13		15		17		19		21		23		25		27		29
			5		7				11		13				17		19				23		25				29
					7				11		13				17		19				23						29
									11		13				17		19				23						29
											13				17		19				23						29
															17		19				23						29
																	19				23						29
																					23						29
																											29

Prvočísla

- Pro každé p, $2 \leq p \in \mathbb{N}$ platí: p je prvočíslo, právě když p není násobkem žádného prvočísla menšího než p.
-
```
es :: Integral a => [a] -> [a]
es (p:t) = p : es [ n | n<-t, n`mod`p/=0 ]

primes = es [2..]
```

Vstup/výstup

Vstup/výstup a referenční transparentnost

Referenční transparentnost

- Daný výraz se vždy vyhodnotí na stejnou hodnotu, bez ohledu na okolí (kontext), ve kterém je použit.
- Programovací jazyk Haskell je referenčně transparentní.

Dopady na vstup-výstupní chování

- **Nelze napsat program, který by zpracoval vstup uživatele a vyhodnotil se podle zadaného vstupu na různé hodnoty.**
- Lze napsat program, který zpracuje vstup a podle vstupu vypíše na výstup různé výsledky.
- Hodnoty předávané skrze vstup-výstupní akce nesouvisí s hodnotou výrazu, který tuto vstup-výstupní akci realizuje.

Typy `IO()` , `IO a`

Vstup-výstupní akce

- Interakce programu s uživatelem nebo operačním systémem.
- Například výpis textu na terminálu, vytvoření nového souboru, načtení hodnoty proměnné prostředí, ...

Myšlenka

- Pro vstup-výstupní akce je zaveden speciální typ – **IO a** .
- Tento typ má formálně jednu jedinou textově nereprezentovatelnou hodnotu, a to **vstup-výstupní akci**.
- Výstupní akce mají typ **IO ()** .

```
putStrLn "Ahoj!":: IO ()
```

- Vstupní akce mají typ **IO a** , kde typová proměnná `a` nabývá hodnoty (typu) podle typu objektu, který vstupuje.

```
getLine :: IO String
```

Vnitřní výsledek a vedlejší efekt

Otázka

- Jestliže `getLine` načte řetězec ze vstupu a přitom má hodnotu vstup-výstupní akce, což je hodnota typu `IO a` , konkrétně zde `IO String` , tak potom by nás zajímalo, kde je onen načtený řetězec?

Vnitřní výsledek a vedlejší efekt

Otázka

- Jestliže `getLine` načte řetězec ze vstupu a přitom má hodnotu vstup-výstupní akce, což je hodnota typu `IO a`, konkrétně zde `IO String`, tak potom by nás zajímalo, kde je onen načtený řetězec?

Odpověď

- Načtený řetězec se uchová jako tzv. **vnitřní výsledek** provedení této vstupní akce.
- Skutečné načtení řetězce a zapamatování si vnitřního výsledku je realizováno jako **vedlejší efekt** vyhodnocení výrazu `getLine`.

Operátor »=

Přístup k hodnotě vnitřního výsledku

- Pomocí binárního operátoru `»=` .
- Ve výrazu `f »= g` funguje operátor `»=` tak, že vezme vnitřní výsledek vstupní akce `f` a na tento aplikuje unární funkci `g` , **jejímž výsledkem je ovšem vstup-výstupní akce**.
- Výraz `f »= g` tedy znamená, že:

```
f :: IO a
g :: a -> IO b
f »= g :: IO b
```

Operátor »=

- `(»=) :: IO a -> ( a -> IO b ) -> IO b`
- Následující zápis je ekvivalentní:

```
getLine »= putStr
getLine »= (\x -> putStr x)
```

Nesprávné použití operátoru »=

Otázka

- Operátor `(»=)` nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine »= length
getLine »= (\ x -> length x)
```

Nesprávné použití operátoru »=

Otázka

- Operátor (»=) nelze použít ke spojení vstup-výstupní akce a funkce, která není vstup-výstupní akce, proč?
- Příklad, **takto nelze**:

```
getLine »= length
getLine »= (\ x -> length x)
```

Odpověď

- Hodnota výrazu je závislá na zadaném vstupu!
- Porušuje referenční transparentnost.
- Typově nesprávně.
- Správné použití:

```
getLine »= print . length
getLine »= (\ x -> print (length x))
```

Řazení vstup-výstupních akcí

Funkce `return`

- Prázdná akce, jejíž provedení má za cíl pouze naplnit hodnotu vnitřního výsledku.

```
return :: a -> IO a
return ['A','h','o','j'] >>= putStr
```

Řazení akcí, operátor `>>`

- Binární operátor, který řadí vstup-výstupní akce.
- Zapomíná/ničí hodnotu vnitřního výsledku.
- Výraz má hodnotu poslední (druhé) vstup-výstupní akce.
- `(>>) :: IO a -> IO b -> IO b`
- Příklady použití:

```
putStr "Jeje" >> putChar '!'
getLine >> putStr "nic"
```

Základní funkce pro výstup

```
putChar :: Char -> IO ()
```

- Zapíše znak na standardní výstup.

```
putStr :: String -> IO ()
```

- Zapíše řetězec na standardní výstup.

```
putStrLn :: String -> IO ()
```

- Zapíše řetězec na standardní výstup a přídá znak konec řádku.

```
print :: Show a => a -> IO ()
```

- Vypíše hodnotu jakéhokoliv tisknutelného typu na standardní výstup, a přidá znak konec řádku.
- Tisknutelné typy jsou instancí třídy `Show` .
- Uživatelem definované typy nutno označit přídomkem `deriving Show` .

Základní funkce pro vstup

```
getChar :: IO Char
```

- Načte znak ze standardního vstupu.

```
getLine :: IO String
```

- Načte řádek ze standardního vstupu.

```
getContents :: IO String
```

- Čte veškerý obsah ze standardního vstupu jako jeden řetězec. Obsah je čten líně, tj. až když je potřeba.

```
interact :: (String -> String) -> IO ()
```

- Argumentem funkce `interact` je funkce, která zpracovává řetězec a vrací řetězec.
- Veškerý obsah ze standardního vstupu je předán této funkci a výsledek vytištěn na standardní výstup.

Práce se soubory

```
type FilePath = String
```

- Definuje typový alias.

```
readFile :: FilePath -> IO String
```

- Načte obsah souboru jako řetězec. Soubor je čten líně.

```
writeFile :: FilePath -> String -> IO ()
```

- Zapíše řetězec do daného souboru (existující obsah smaže).
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show` .

```
appendFile :: FilePath -> String -> IO ()
```

- Připíše řetězec do daného souboru.
- Hodnoty jiného typu než `String` lze konvertovat funkcí `show` .

Další vstup-výstupní akce

Moduly System a Directory

- Existují další vstup-výstupní funkce pro práci s adresáři či systémovými proměnnými.
- Tyto funkce jsou předdefinovány v modulech `System` a `Directory`.

Použití modulu

- Moduly, jejichž funkce chceme použít, je třeba označit.
- Lze to učinit v souboru s globálními definicemi použitím klíčového slova `import`.
- Příklad:

```
import Char
import Directory
main = getDirectoryContents ".." »=
       print . map (\x -> (toUpper.head) x : tail x)
```

Zápis pomocí `do`

Pozorování

- Syntaktická konstrukce `do` slouží k alternativnímu zápisu výrazu s operátory `>>=` a `>>`.

Následující zápis je ekvivalentní

```
putStr "vstup?"  >>
getLine          >>= \ x ->
putStr "výstup?" >>
getLine          >>= \ y ->
readFile x       >>= \ z ->
writeFile y (map toUpper z)
```

```
do putStr "vstup?"
   x <- getLine
   putStr "výstup?"
   y <- getLine
   z <- readFile x
   writeFile y (map toUpper z)
```

Funkce `sequence`

- Máme-li seznam vstup-výstupních akcí, můžeme je pomocí funkce `sequence` provést dávkově naráz.
- ```
  sequence :: [IO a] -> IO [a]
  sequence [] = return []
  sequence (a:s) = do x<-a
                      t <- sequence s
                      return (x:t)
  ```

Příklady použití

- V případě výstupních akcí je výsledkem vyhodnocení výrazu posloupnost výstupů, viz:

  ```
  sequence [ putStr "Ahoj", putStr " ", putStr "světe!"]
  ```
- V případě vstupních akcí, je výsledkem vyhodnocení výrazu seznam vstupů, který je uložený jako vnitřní výsledek vstup-výstupní akce, viz:

  ```
  sequence [ getLine, getLine, getLine ] >>= print
  ```
  ```

Funkce `mapM`

Funkce `mapM`

- Aplikuje unární funkci, jejíž výsledkem je vstup-výstupní akce, na seznam hodnot a vzniklý seznam vstup-výstupních akcí provede.

```
mapM :: (a -> IO b) -> [a] -> IO [b]
mapM f = sequence . map f
```

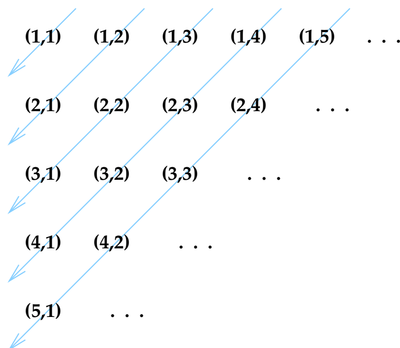
Příklady použití

- `mapM putStr ["Den","Noc"]`
  vypíše `DenNoc`
- `mapM (\ t -> putStr "Aa") [1,2,3,4,5]`
  vypíše `AaAaAaAaAa`
- `mapM (\ x->getLine) [1,2] >>= print`
  po zadání dvou řádků s obsahem `radek1` a `radek2`
  vypíše `["radek1","radek2"]`

Domácí cvičení 1/2

Definice seznamů

- Definujte seznam všech uspořádaných dvojic přirozených čísel tak, aby dvojice byly v definovaném seznamu uspořádány dle následujícího schématu:



- Nápověda: součet čísel v dvojici je po diagonále shodný a postupně se zvyšuje o jedna.

Domácí cvičení 2/2

Vstup výstup

- Napište program, který vyzve uživatele, aby zadal 16 čísel oddělených mezerou, a poté tato čísla vypíše v matici velikosti 4x4.
- Napište program, který ve vhodně formátovaném výstupu (např. rozbalený souborový strom) vypíše obsah aktuálního adresáře včetně všech jeho podúrovní.
- Vstup-výstupní programy vytvářejte jako samostatně spustitelné programy.