IB015 Neimperativní programování

Redukční strategie, Nekonečné datové struktury
Zápis a generování seznamů

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**.
- Nicméně mluví se o **líném vyhodnocování**, zjednodušeně řečeno, vyhodnotí se pouze to, co je potřeba k dalšímu výpočtu.

Příklady vyhodnocování s různou strategií

Definice funkce

- `cube x = x * x * x`

Striktní redukční strategie

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

Normální redukční strategie

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

Líná redukční strategie

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

Praktická poznámka k Haskellu

Líná redukční strategie a Haskell?

- Je přítomna, ale ne v podobě zavedené na předchozím slajdu.
- Ne vše lze líně vyhodnotit (výrazy s vedlejším efektem).
- V Haskellu lze docílít, 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`

Referenčně transparentní výrazy

Referenční transparence

- Výsledek vyhodnocení výrazu nezávisí na kontextu, ve kterém se daný výraz vyhodnocuje.
- Může mít vedlejší efekt, ten ale nesmí ovlivnit výsledek.
- Haskell je **referenčně transparentní**,

Vedlejší efekt vyhodnocení výrazu či funkce

- Změna stavu světa, která je pozorovatelná vně volané funkce nad rámec návratové hodnoty.
- Například modifikace globální proměnné, modifikace hodnot v kontextu rodičovské funkce, modifikace externí paměti, atd.

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)` $\rightsquigarrow$ **ERROR**

- Při normálním vyhodnocování k němu nedojde

`const 2 (1/0)` $\rightsquigarrow$ `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 normálním 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.

Normální vyhodnocování

Jiný pohled

- Při použití 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.

Práce s nekonečnými seznamy

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**.

Nekonečné opakování jednoho prvku

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

Jak to funguje?

- `take 8 (repeat 1)` $\rightsquigarrow^*$ `[1,1,1,1,1,1,1,1]`
- `head (repeat 1)` $\rightsquigarrow$ `head (1 : repeat 1)` $\rightsquigarrow$ `1`
  ```

Další funkce generující nekonečné seznamy

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)
  ```

Vyhodnoťte

- `take 4 (iterate not True)` $\rightsquigarrow^*$
- `take 4 (iterate (+1) 0 )` $\rightsquigarrow^*$

Další funkce generující nekonečné seznamy

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)
  ```

Vyhodnoťte

- `take 4 (iterate not True)` $\rightsquigarrow^*$ `[True,False,True,False]`
- `take 4 (iterate (+1) 0 )` $\rightsquigarrow^*$ `[0,1,2,3]`

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

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"`

Líné vyhodnocování a rekurze

Líně vyhodnocená rekurze

- K definici nekonečných datových struktur jsme použili rekurzi.
- Rekurze se zanoří **pouze tolikrát, kolikrát je třeba**.

Příklad

- Seznam nekonečně mnoha jedniček:

```
jednicky = 1 : jednicky
```

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

`jednicky` $\rightsquigarrow$ `1:jednicky` $\rightsquigarrow$ `1:1:jednicky` $\rightsquigarrow \cdots$

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

`head jednicky` = `head jednicky` $\rightsquigarrow$ `head (1:jednicky)` $\rightsquigarrow$ `1`

Příklady rekurzivní 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	...	

Zápis a generování seznamů

Zápis seznamu výčtem

Prostý výčet

- Zápis pomocí základních hodnotových 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) ⇝* [0,2,4,6,8,10,12,14,16,18]
enumFromThenTo 0 3 15 ⇝* [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] ]`
  $\rightsquigarrow^*$ `[0,1,4,9]`
- `[ (c,k) | c<-"abc", k<-[1,2] ]`
  $\rightsquigarrow^*$ `[('a',1),('a',2),('b',1),('b',2),('c',1),('c',2)]`
- `[ 3*n | n<-[0..6], odd n ]`
  $\rightsquigarrow^*$ `[3,9,15]`
- `[ (m,n) | m<-[1..3], n<-[1..3], n<=m ]`
  $\rightsquigarrow^*$ `[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]`
- `[ (m,n) | m<-[1..3], n<-[1..m] ]`
  $\rightsquigarrow^*$ `[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]`
- `[ (x,y) | z<-[0..2], x<-[0..z], let y=z-x ]`
  $\rightsquigarrow^*$ `[(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 uspořádáných 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..]
  ```
  ```

Riziko zacyklení

Pozorování

- Využití generátorů seznamů ve spojení s nekonečnými seznamy trochu kazí čistotu deklarativního přístupu, ve kterém se v zásadě nezajímáme o způsob vyhodnocení, ale pouze o podstatu vyjádření vztahů.

Porovnejte a vysvětlete rozdíl

- `take 4 [ x+y | x <- [1..], y<-[1..], x<2 ]` $\rightsquigarrow^*$
- `take 4 [ x+y | y <- [1..], x<-[1..], x<2 ]` $\rightsquigarrow^*$

Riziko zacyklení

Pozorování

- Využití generátorů seznamů ve spojení s nekonečnými seznamy trochu kazí čistotu deklarativního přístupu, ve kterém se v zásadě nezajímáme o způsob vyhodnocení, ale pouze o podstatu vyjádření vztahů.

Porovnejte a vysvětlete rozdíl

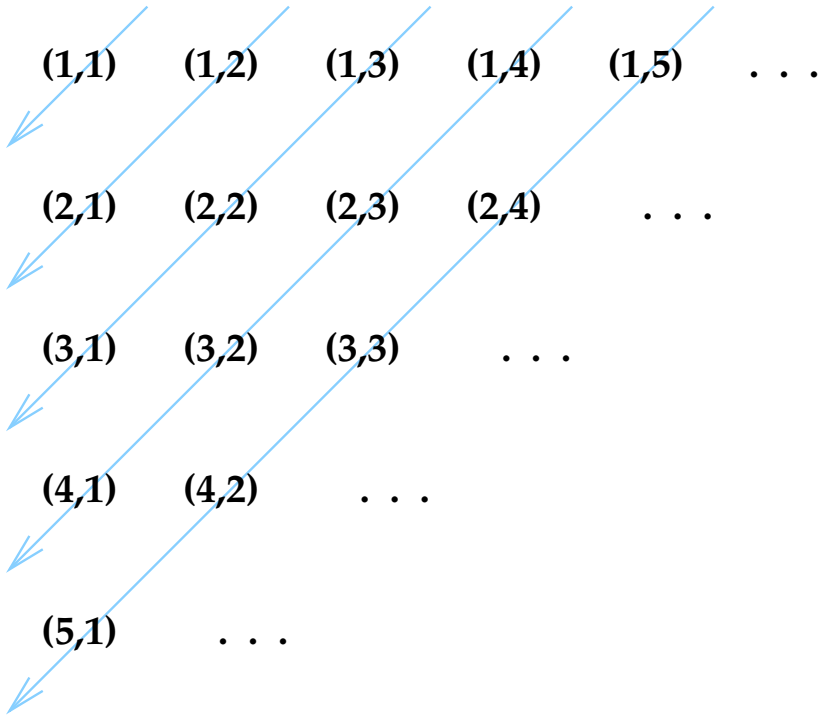
- `take 4 [ x+y | x <- [1..], y<-[1..], x<2 ]` $\rightsquigarrow^*$ `[2,3,4,5]`
- `take 4 [ x+y | y <- [1..], x<-[1..], x<2 ]` $\rightsquigarrow^*$ `[2`

`(druhý výpočet cyklí)`

Checkpoint

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.