Úvod do funkcionálního programování

studijní text

Libor Škarvada

Copyright 2002 Libor Škarvada

Testováno výhradně na studentech – při přípravě textu neutrpěla újmu žádná zvířátka.

Obsah

1 Úvod

Co je to funkcionální programování?

Funkcionální programovací paradigma patří mezi tzv. *deklarativní* paradigmata. Slovo *paradigma* (z řečtiny $\pi\alpha\rho\acute{\alpha}\delta\varepsilon\iota\gamma\mu\alpha$ = vzor) značí myšlenkový vzor, metodu, přístup, model uvažování.

Klasický *imperativní* přístup se zaměřuje na to, *jak* problém řešit, popis řešení je rozepsán do kroků probíhajících v čase.

Naproti tomu deklarativní přístup klade důraz na to, *co* je řešením daného problému. Deklarativní zápis programu představuje již samotné řešení problému zapsané v jistém tvaru. Výpočet podle tohoto programu je pak transformací takového popisu do tvaru jednoduššího, použitelnějšího.

Rozdíl mezi imperativním a funkcionálním přístupem k řešení problému lze ilustrovat třeba na úloze *Najít index největšího prvku v posloupnosti* $a_1, a_2, \ldots, a_n$ *navzájem různých kladných celých čísel*. Imperativní program zavádí pomocné proměnné – paměťová místa i, j, max, a skládá se z kroků

1. polož $max = 0$
2. pro j od 1 do n opakuj: jestliže $a_j > max$, pak polož $i = j$ a $max = a_j$
3. výsledkem je hodnota proměnné i.

Deklarativní popis řešení téže úlohy může vypadat takto:

Takové $i \in \{1, \ldots, n\}$, pro které platí: pro všechna $j \in \{1, \ldots, n\}$, je $a_i \geq a_j$.

Funkcionální paradigma je založeno na zápisu programu ve tvaru výrazu. Nejdůležitějšími složkami těchto výrazů jsou funkce a jejich aplikace na argumenty. Výpočet funkcionálního programu spočívá ve zjednodušování výrazu až do doby, kdy výraz dále zjednodušit nelze. Tento dále nezjednodušitelný tvar považujeme za výsledek výpočtu.

Například problém *Najít součin čísel 6 a 7* je zapsán výrazem

```
6 * 7
```

Tento výraz je aplikací funkce násobení `(*)` na argumenty 6 a 7. Výpočtem – zjednodušením výrazu – dostaneme hodnotu 42. Ta je dále nezjednodušitelná, tj. je odpovědí – řešením problému.

Skutečnost, že výraz e se zjednoduší (*redukuje*) na hodnotu c, budeme značit $e \rightsquigarrow^* c$, takže výše uvedenou redukci můžeme zapsat

$$\texttt{6 * 7} \rightsquigarrow^* \texttt{42}$$

Výrazy se skládají z jednodušších výrazů – *podvýrazů*. Podvýrazy výrazu $M\ N$ jsou sám výraz $M\ N$, výraz M, výraz N a podvýrazy těchto podvýrazů. Například ve výrazu

```
sqrt ( sin x )
```

jsou tyto podvýrazy

```
sqrt
sin
x
sin x
sqrt ( sin x )
```

Nejdůležitější způsob, jak ze dvou výrazů vytvořit výraz složitější, je aplikace funkce na argument. Aplikaci výrazu M na výraz N zapisujeme $M\,N$. Například zapsáním výrazu `pi` za výraz `sin` dostáváme výraz

```
sin pi
```

jehož význam je „aplikace funkce sinus na číslo π“.

Některé funkce a operátory lze aplikovat na více než jeden argument. Například operaci sčítání `(+)` lze aplikovat na dvě čísla (sčítance). Takovým funkcím a operátorům, které lze aplikovat na dva argumenty, říkáme *binární*. Obecně může být arita funkce nebo operátoru libovolná – toto však upřesníme v kap. 6.

Aplikaci binárního operátoru na dva argumenty můžeme zapsat buď takto (prefixově)

```
(+) 19 23
```

anebo takto (infixově)

```
19 + 23
```

Nutnou podmínkou, aby mělo smysl aplikovat výraz M na výraz N, je, aby M byla funkce. Například má smysl psát

```
square 5
not False
```

neboť `square` je funkce z čísel do čísel a `not` je funkce z logických hodnot do logických hodnot, ale nemají smysl aplikace

```
5 square
True 8
```

Rovněž by nemělo smysl aplikovat funkci na *libovolný* argument, výrazy

```
toUpper False
not 7
sqrt '#'
```

nemají smysl.[1] V tzv. *typovaných jazycích* se problémů se zacházením s takovými „nesmyslnými“ výrazy elegantně zbavíme tím, že je za výrazy vůbec nepovažujeme:

[1] `toUpper` je funkce pro převod na velká písmena; jejím argumentem musí být znak.

kompilátor typovaného jazyka výskyt takového „výrazu“ v programu považuje za syntaktickou chybu. V *netypovaných jazycích*, kde se na synaktické úrovni nedají rozlišit výrazy smysluplné od nesmyslných, by však třeba výraz `not 7` byl legálním programem. Teprve jeho výpočet (tj. až v době běhu) by skončil chybovým hlášením.

Klasické imperativní jazyky zpravidla chápou funkce zcela odlišně od ostatních hodnot: většinou umožňují funkce definovat pouze jako konstanty, bez možnosti předávat je jako parametry nebo vracet funkce jako výsledek aplikace jiných funkcí. V Pascalu například můžeme mít pole čísel, ale nemůžeme mít pole funkcí. Na druhé straně, funkcionální jazyky pohlížejí na funkci stejně jako na každou jinou hodnotu. Zejména tedy může funkce být parametrem jiné funkce nebo může být funkce výsledkem výpočtu. Funkcím, jejichž argument nebo výsledek může být zase funkce, říkáme *funkce vyšších řádů*. Je možno například definovat operátor `(.)` vyššího řádu

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

který realizuje skládání funkcí. Pak výraz `sin . sqrt` má hodnotu funkce počítající sinus druhé odmocniny svého argumentu, zatímco `sqrt . sin` je funkce počítající druhou odmocninu sinu svého argumentu.

Skutečnost, že ve funkcionálních jazycích není použití funkcionálních hodnot oproti jiným hodnotám omezeno, se někdy vyjadřuje obratem *funkce jsou občané první kategorie*[2].

Srovnání s imperativním programováním

Základní sémantickou jednotkou funkcionálního jazyka je *výraz*. Z jednodušších výrazů se podle syntaktických pravidel mohou skládat výrazy složitější. Sémantická vazba mezi podvýrazy ve výrazu je dána přirozeným pravidlem:

> Ve výrazu $F\,A$ je funkce, která se bude aplikovat na argument, dána hodnotou podvýrazu F; hodnota argumentu, na který se tato funkce bude aplikovat, je dána hodnotou podvýrazu A.

Základní sémantickou jednotkou imperativního jazyka je *příkaz*. Na rozdíl od výrazů, příkazy obvykle nemají hodnotu[3]. Proto musí existovat jiný prostředek pro „výměnu dat“ mezi příkazy. Tímto prostředkem je tzv. *stav*. Stavem se rozumí vektor hodnot programových proměnných v daném okamžiku výpočtu. Příkazy mohou stav měnit. Nejznámějším příkazem pro změnu stavu je *přiřazovací příkaz*.

Existence stavu má jeden závažný důsledek. Mějme například funkci, která pro každé kladné celé číslo n spočte součet druhých mocnin přirozených čísel od 1 do n, a zapišme ji v imperativním jazyce (Pascalu)

```
function sumsq (n:integer): integer;
```

[2] First-class citizens

[3] Přesněji: jejich hodnota bývá definována jako prázdná – speciální prvek `()`. To není pravda ve všech jazycích, např. v C nebo v Algolu 68 mohou i příkazy vracet netriviální hodnoty. Avšak při skládání příkazů P, Q se hodnota sekvence $P;Q$ obvykle definuje jako hodnota příkazu Q, a hodnota příkazu P zůstává stejně nevyužita...

```
var s, k: integer;
begin
  s := 0;
  for k := 1 to n do
    s := s + k*k;
  sumsq := s
end
```

Při volání této funkce s argumentem n se výraz `s+k*k` vyhodnocuje n-krát, ale pokaždé má jinou hodnotu, zejména hodnoty proměnných `s` a `k` jsou v každém průchodu cyklu jiné.

To se ve funkcionálních jazycích stát nemůže.[4] Čistě funkcionální jazyky mají totiž vlastnost, jíž říkáme *referenční transparentnost*:

> Jestliže se ve stejném kontextu vyskytne tentýž podvýraz, pak jeho hodnota bude vždy stejná.

Například funkci `sumsq` z předchozího příkladu můžeme definovat

```
sumsq n  =  if  n==1  then  1  else  n*n + sumsq (n-1)
```

Pak při každém zavolání této funkce bude například hodnota parametru `n` stejná na všech čtyřech místech výrazu na pravé straně definice funkce `sumsq`. Ostatně ani neexistuje způsob, jak bychom ji mohli změnit: ve funkcionálním programování nemáme přiřazovací příkaz ani jiné „příkazy“, které by měly vedlejší efekty, tj. které by měnily globální stav.

Nabízí se tu námitka, že funkce `sumsq` je referenčně transparentní jen do té doby, než dojde k jejímu opětnému vyvolání; pak se hodnota proměnné `n` změní – sníží o jedničku. Ve skutečnosti však k porušení referenční transparentnosti nedochází: proměnná `n` má ve výrazu `if n==1 then 1 else n*n + sumsq (n-1)`. pouze čtyři výskyty a mimo definici funkce `sumsq` tato proměnná neexistuje. Při dalším zavolání funkce `sumsq` jde o novou proměnnou, která má s původní společné jen jméno.

Funkcionální jazyky nemají pojem stavu ani přiřazení do proměnné. Proměnné mohou být vázány na hodnotu, ale tato hodnota nemůže být přepsána. Jediný způsob, jak svázat proměnnou s hodnotou, je při aplikaci funkce na argument: proměnná (formální parametr) se sváže s argumentem (skutečným parametrem). Máme-li například funkci `square` definovanou

```
square x  =  x * x
```

a aplikujeme-li ji na argument `(6+7)` ve výrazu

```
square (6+7)
```

pak při výpočtu se proměnná `x` sváže s výrazem `(6+7)`, výraz `square(6+7)` se nahradí pravou stranou definice funkce `square`, tj. výrazem `x * x`. V něm se však proměnná `x` nahradí výrazem, s nímž je svázána, takže na konci prvého kroku výpočtu bude mít celý výraz tvar

[4] Myslí se *čistě* funkcionální jazyky. Některé jazyky, např. LISP, ML, kombinují funkcionální a imperativní přístup a počítají se stavem – tyto jazyky nepatří mezi čistě funkcionální.

```
(6+7) * (6+7)
```

Vazbě formálních parametrů funkcí na skutečné parametry se říká *prostředí*.

Skutečnost, že ve funkcionálních programech není pojem stavu, usnadňuje uvažování o programech, dokazování jejich správnosti. Určité transformace funkcionálních programů se definují a provádějí mnohem jednodušeji, než odpovídající transformace programů imperativních.

Shrnutí

- Ve funkcionálním programování je následující pojetí programu a výpočtu:
  program ..... výraz
  výpočet ..... úprava výrazu
  výsledek ..... dále nezjednodušitelný tvar výrazu
- Funkce jsou „občané první kategorie“ mající stejná „práva a povinnosti“ jako například konstanty.
- Funkcionální jazyky jsou referenčně transparentní, stejný výraz má ve stejném prostředí vždy stejnou hodnotu.

2 Výrazy, definice

Funkce a operátory

Základní způsob, jak vytvářet složitější výrazy z jednodušších, je aplikace funkce na argumenty. Aplikaci funkce `f` na argumenty `x`, `y`, `z` zapisujeme

```
f x y z
```

Pokud je argumentem proměnná, není nutno ji uzavírat do závorek.

V souladu s terminologií jazyka Haskell budeme *operátorem* nazývat binární funkci, jejíž jméno nezačíná písmenem. Například `(+)`, `(-)`, `(*)`, `(/)`, `(^)` budou běžné aritmetické operátory, `(==)`, `(/=)`, `(<)`, `(<=)`, `(>)`, `(>=)` relační, a `(&&)`, `(||)` booleovské operátory.

Uvedený *prefixový* tvar aplikace však není vždy pohodlný. Proto v případě binárních operátorů budeme kromě prefixového zápisu aplikace

```
(+) x y
```

používat též *infixový* zápis

```
x + y
```

Navíc, u asociativních binárních operátorů můžeme vynechávat nadbytečné závorky a využívat obvyklé precedence, takže třeba

```
(+) ((*) ((*) 1 2) 3) ((*) 3 4)
```

lze zapsat přehledněji

```
1 * 2 * 3  +  3 * 4
```

Prefixově zapsaná aplikace má při výpočtu přednost před infixovou: `f x+y` je totéž jako `(f x) + y`.

Všimněme si, že odkazujeme-li se na operátory samostatně (např. vystupují-li ve výrazu jako argumenty vyšších funkcí apod.), uvádíme je v kulatých závorkách. V infixově zapisované aplikaci je uvádíme bez závorek. Naopak binární funkce (jejich jméno je alfanumerické a začíná malým písmenem) uvádíme bez závorek a jejich aplikaci píšeme obvykle prefixově. Při alternativním infixovém zápisu je uvádíme v obrácených apostrofech.

```
21  `lcm`  6
lcm  14    21
1996 `mod` 977
mod  1992  50
```

Definice

Kdybychom byli při vytváření výrazů odkázáni pouze na pevně danou malou množinu konstant a funkcí, byly by naše vyjadřovací schopnosti silně omezené. To, co dává funkcionálním jazykům vyjadřovací sílu, je možnost definovat nové konstanty a funkce.

Jestliže chceme ve výrazu použít nějaký podvýraz vícekrát, např.

```
(pi/2) * x  +  (pi/2) * y
```

můžeme v něm *lokálně* zavést novou proměnnou `d`:

```
let  d = pi/2  in  d * x  +  d * y
```

Části výrazu mezi `let` a `in` říkáme *definice*. Nově definované proměnné (nalevo od rovnítka) jsou vázány na výrazy (napravo od rovnítka) a jsou použitelné v části výrazu za `in`. Těchto lokálních definic může být v rámci jednoho výrazu `let` i více naráz a mohou být použity i na pravých stranách definic:

```
let  a = 3
     b = 4
     c = 5
     s = (a+b+c)/2
in  s*(s-a)*(s-b)*(s-c)
```

Alternativně můžeme lokální definice zapisovat pomocí konstrukce `where`. Ta se od výrazu `let` liší tím, že lokální definice je uvedena až za výrazem, v němž je použita.

```
f x y  =  d * x  +  d * y  where  d = pi/2

p s  =  s*(s-a)*(s-b)*(s-c)  where  s = (a+b+c)/2
```

Druhý rozdíl spočívá v tom, že konstrukce `where` může stát jen v nadřazené definici, zatímco `let` je výraz a může být tedy podvýrazem libovolného výrazu.

Řekli jsme, že definice za slovem `where` mají jen lokální působnost – vztahují se pouze na podvýraz před `where`. Často bývá pohodlné definovat si nové funkce i globálně – pro všechny výrazy, s nimiž pracujeme v programu. Proto je v jazyce Haskell také možnost definovat konstanty a funkce i *globálně*. Definice se zapíší do zvláštního souboru – *skriptu*.

Definovat můžeme i funkci. Kromě jejího jména zapíšeme nalevo od definičního rovnítka i její parametry, což budou (v nejjednodušším případě) jména proměnných:

```
cube x  =  x * x * x
max3 a b c  =  max a (max b c)
u .-. v  =  if u>v then u-v else 0
```

Poslední řádek definuje operátor `(.-.)`, ekvivalentní způsob zápisu je

```
(.-.) u v  =  if u>v then u-v else 0
```

Podstatným důvodem, proč definice zvyšují naši vyjadřovací sílu, je fakt, že definice mohou být rekursivní:

```
fact n  =  if n==0 then 1 else n*fact(n-1)
a ^ n  =  if n==0 then 1 else a*(a^(n-1))
```

Na místě formálních parametrů funkcí a operátorů nemusejí vystupovat jen proměnné. Obecně zde jsou tzv. *vzory*. Vzor popisuje množinu argumentů, které tomuto vzoru „vyhovují“. Definice funkce se pak rozpadá na více větví, *klausulí*. Při výpočtu se pak z definice použije ta větev, jejíž vzory vyhovují skutečným argumentům. Například výše uvedenou definici funkce `fact` lze pomocí vzorů zapsat

```
fact 0  =  1
fact n  =  n * fact(n-1)
```

Častým vzorem, jak je vidět z těchto příkladů, je konstanta. Takový vzor vyhovuje pouze jedinému skutečnému argumentu – tomu, jenž je roven právě této konstantě. V tomto příkladě to je 0 v prvním řádku definice. Naopak vzor, kterým je proměnná, vyhovuje všem argumentům. To je případ druhého řádku definice. V našem příkladě případ, že argument funkce `fact` je nulový, vyhovuje oběma řádkům definice. Proto, aby taková definice byla korektní, dohodneme se, že se při výpočtu použije vždy *první* větev definice, jejíž vzory vyhovují skutečným argumentům funkce. V uvedeném příkladě se tedy první větev definice bere v případě nulového a druhá větev v případě nenulového argumentu.

Podobně funkci `sumsq` pro výpočet součtu čtverců z kap. 1 lze zapsat i takto:

```
sumsq 1  =  1
sumsq n  =  n*n + sumsq (n-1)
```

Jiným příkladem mohou být definice booleovských operátorů, v nichž vzory jsou opět booleovské konstanty nebo proměnné.

```
False && x  =  False
True  && x  =  x

False || x  =  x
True  || x  =  True

not True   =  False
not False  =  True
```

Vzory však nemusejí být jen konstanty a proměnné. Obecněji je vzor výraz, jehož žádný podvýraz nelze redukovat – skládá se pouze z proměnných a tzv. *konstruktorů*. Jiné konstruktory než konstanty poznáme v kap. 5.

Shrnutí

- Výrazy se skládají z aplikací funkcí a operátorů na argumenty a z lokálních definic.
- Globální definice zavádějí nové funkce a konstanty, které platí ve všech dále použitých výrazech.
- Definice mohou být rekursivní.
- Funkce lze definovat pomocí vzorů – výrazů popisujících možné tvary skutečných argumentů.

3 Výpočty

Mějme výraz `(2*3) * (4+3)`. Tento výraz můžeme vyhodnocovat dvěma způsoby, lišícími se pořadím podvýrazů, které upravujeme dříve:

```
(2*3) * (4+3) ⇝ 6 * (4+3) ⇝ 6 * 7 ⇝ 42
(2*3) * (4+3) ⇝ (2*3) * 7 ⇝ 6 * 7 ⇝ 42
```

V dalším příkladě lze daný výraz vyhodnocovat dokonce devíti různými způsoby.

Příklad 1 Jestliže je funkce `sq` definována

```
sq x  =  x * x
```

pak všechny možné způsoby vyhodnocení výrazu `sq(sq 3)` jsou:

```
sq(sq 3) ⇝ sq 3*sq 3 ⇝ (3*3)*sq 3 ⇝ 9*sq 3 ⇝ 9*(3*3) ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq 3*sq 3 ⇝ (3*3)*sq 3 ⇝ (3*3)*(3*3) ⇝ 9*(3*3) ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq 3*sq 3 ⇝ (3*3)*sq 3 ⇝ (3*3)*(3*3) ⇝ (3*3)*9 ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq 3*sq 3 ⇝ sq 3*(3*3) ⇝ (3*3)*(3*3) ⇝ 9*(3*3) ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq 3*sq 3 ⇝ sq 3*(3*3) ⇝ (3*3)*(3*3) ⇝ (3*3)*9 ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq 3*sq 3 ⇝ sq 3*(3*3) ⇝ sq 3*9 ⇝ (3*3)*9 ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq(3*3) ⇝ (3*3)*(3*3) ⇝ 9*(3*3) ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq(3*3) ⇝ (3*3)*(3*3) ⇝ (3*3)*9 ⇝ 9*9 ⇝ 81
sq(sq 3) ⇝ sq(3*3) ⇝ sq 9 ⇝ 9*9 ⇝ 81
```

V příkladě 1 pokaždé dojdeme k výsledku, 81, nezávisle na pořadí vyhodnocování. To se však nemusí stát vždy: existují výrazy, z nichž určitým pořadím vyhodnocování dojdeme k výsledku (normální formě), ale při jiném pořadí vyhodnocovaných podvýrazů se výpočet zacyklí.

Příklad 2 Mějme funkce `f`, `const` definované předpisy

```
f x       =  f x
const x y  =  x
```

Pak výraz `const 3 (f 1)` lze vyhodnocovat nekonečně mnoha způsoby. Jednak těmi, které vedou k výsledku 3, ale před tím libovolně dlouho upravujeme podvýraz `f 1`,

```
const 3 (f 1) ⇝ 3
const 3 (f 1) ⇝ const 3 (f 1) ⇝ 3
const 3 (f 1) ⇝ const 3 (f 1) ⇝ const 3 (f 1) ⇝ 3
    ⋮
```

ale existuje také vyhodnocování, kdy podvýraz `f 1` „upravujeme nekonečně dlouho“, tj. výpočet se zacyklí a k výsledku nedospějeme nikdy:

```
const 3 (f 1) ⇝ const 3 (f 1) ⇝ const 3 (f 1) ⇝ ···
```

Pro jednoznačnost výpočtu je tedy užitečné stanovit pravidla, podle nichž se budou výrazy vyhodnocovat.

Nechť funkce f je definovaná předpisem tvaru

$$f \quad x_1 \quad \ldots \quad x_n \quad = \quad N$$

Redukční krok je úprava výrazu, při níž se některý jeho podvýraz tvaru

$$f \quad M_1 \quad \ldots \quad M_n$$

nahradí pravou stranou definice funkce, tj. výrazem N, v němž se každý výskyt proměnné x_i nahradí odpovídajícím výrazem M_i. Skutečnost, že výraz P se jedním redukčním krokem upraví na výraz Q, značíme $P \rightsquigarrow Q$. To je případ redukcí v příkladě 2.

V obecnějším smyslu redukčním krokem nazýváme i takové transformace výrazu, při nichž v nějakém podvýrazu vyčíslíme jednoduchou operaci (aritmetickou, logickou,...) aplikovanou na jednoduché argumenty, například `sq(3*3)` $\rightsquigarrow$ `sq 9` v příkladě 1.

Redukční strategie je pravidlo, které pro každý výraz jednoznačně určuje první redukční krok.[5]

Normální redukční strategie je taková, při níž aplikaci

$$M \ N_1 \ \ldots \ N_n$$

upravujeme tak, že nejdříve redukujeme, pokud to jde, jeho podvýraz M použitím normální redukční strategie. V případě, že M redukovat nelze, pak to musí být funkce a celý výraz $M \ N_1 \ \ldots \ N_n$ nahradíme pravou stranou její definice, dosadivše výrazy N_i za její formální parametry. Zjednodušeně lze říci, že při normální redukční strategii postupujeme *zvnějšku a zleva*.

V příkladě 1 normální redukční strategii odpovídá první redukce. V příkladě 2 odpovídá normální redukční strategii také první (nejkratší) redukce.

Striktní redukční strategie je taková, při níž aplikaci

$$M \ N_1 \ \ldots \ N_n$$

upravujeme tak, že nejdříve, pokud to jde, redukujeme jeho podvýraz N_n použitím striktní redukční strategie. Pokud podvýraz N_n nelze redukovat, redukujeme (striktně) podvýraz N_{n-1}, když ani ten nelze redukovat, redukujem podvýraz N_{n-2}, atd. Pokud nelze reukovat ani jeden z výrazů $N_1, \ldots, N_n$, redukujeme striktní redukční strategií podvýraz M. Pokud ani ten nelze redukovat, je M funkce a celý výraz $M \ N_1 \ \ldots \ N_n$ nahradíme pravou stranou její definice, dosadivše hodnoty N_i za její formální parametry. Zjednodušeně lze říci, že při striktní redukční strategii postupujeme *zevnitř a zprava*.

V příkladě 1 pouze poslední redukce odpovídá striktní redukční strategii, a v příkladě 2 odpovídá striktní redukční strategii nekonečná (zacyklená) redukce.

Normální redukční strategii se také říká *volání jménem*. Tohoto názvu se používá zejména v imperativních programovacích jazycích. Striktní redukční strategii se říká, zejména v imperativních jazycích, *volání hodnotou*.

[5] Prakticky tedy redukční strategie určuje, *který podvýraz* se bude redukovat jako první. V obecnějších redukčních systémech (než jsou funkcionální jazyky) redukční stragie určuje nejen který podvýraz se bude redukovat, ale i *jak* se bude redukovat.

Viděli jsme, že volba redukční strategie může ovlivnit funkcionální výpočet natolik, že rozhodne o tom, zda vůbec dospějeme k výsledku. Naštěstí si alespoň můžeme být jisti, že pokud k výsledku dospějeme, bude jediný, tj. nemůže se stát, abychom dvěma strategiemi došli k různým výsledkům.

Veta 1 (Churchova-Rosserova vlastnost pro funkcionální jazyky) Pro každý výraz M platí: redukujeme-li M pomocí dvou redukčních strategií a v obou případech obdržíme po konečném počtu kroků výsledek, pak jsou oba výsledky stejné.

Věta 1 však ještě neříká nic o tom, zda při použití nějaké redukční strategie výsledek *vůbec* obdržíme. Následující věta říká, že normální redukční strategie je nejlepší strategií, pokud se chceme vyhnout zacyklení.

Veta 2 Lze-li nějakou redukční strategií redukovat výraz M na výslednou hodnotu (normální formu) M', pak k této hodnotě M' dospějeme po konečném počtu kroků při použití normální redukční strategie.

Také říkáme, že normální redukční strategie je *efektivně normalizující*. Jednoduchou ilustrací věty 2 je náš příklad 2: s vyhodnocováním výrazu `const 3 (f 1)` neskončíme, dokud redukujeme striktně. Jakmile však přejdeme k normální redukční strategii, jsme hotovi po jediném redukčním kroku.

Z příkladu 1 však také vidíme, že normální redukční strategie nemusí vždy vést k *nejkratšímu* výpočtu. Je to proto, že při náhradě aplikace funkce na výraz (`sq(sq 3)`) pravou stranou funkční definice, ve které se formální parametr vyskytuje na více místech (`x*x`), dojde k „rozmnožení“ vnitřního výrazu (`sq 3 * sq 3`), a ten se pak opakovaně redukuje. Protože však funkcionální jazyk je referenčně transparentní, víme, že všechny (oba) výskyty vnitřního výrazu (`sq 3`) se budou redukovat stejně, a proto si opakovanou práci můžeme ušetřit. Proto normální redukční strategii poněkud modifikujeme a takto modifikovanou strategii nazveme *línou redukční strategií*.

Líná redukční strategie je taková, při níž aplikaci

$$M\ N_1\ \ldots\ N_n$$

upravujeme tak, že nejdříve redukujeme, pokud to jde, jeho podvýraz M línou redukční strategií. Když M takto redukovat nelze, pak M musí být funkce a celý výraz $M\ N_1\ \ldots\ N_n$ nahradíme pravou stranou její definice, dosadivše výrazy N_i za její formální parametry. Nyní si však při každém násobném dosazení zapamatujeme, které podvýrazy jsou stejné (vzniklé dosazením za různé výskyty téže proměnné), a při jejich případném dalším vyhodnocování je redukujeme jen jednou (jakoby „všechny naráz“).

Při striktní redukční strategii tedy vyhodnocujeme argumenty funkcí právě jednou. Při normální redukční strategii vyhodnocujeme argumenty funkcí tolikrát, kolikrát na to při výpočtu dojde, tj. nulkrát, jednou nebo víckrát. Při líné redukční strategii však každý argument vyhodnocujeme nejvýše jedenkrát.

Příklad 3 Buď `eat` funkce definovaná

```
eat x y  =  x && (x || y)
```

Redukujeme-li výraz `eat (not False) (not True)` striktně, vyhodnocuje se každý argument (`not False`, `not True`) jednou. Redukujeme-li výraz normálně, vyhodnocuje se první argument dvakrát a druhý ani jednou. Redukujeme-li výraz líně, vyhodnocuje se první argument pouze jednou a druhý se nevyhodnocuje vůbec.

Pro línou redukční strategii platí analogie věty 2, tj. pokud má nějaký výraz normální formu (tj. výslednou hodnotu), pak k ní dospějeme po konečném počtu kroků při líné redukční strategii.

Líná redukční strategie se používá v moderních funkcionálních jazycích a díky ní je možno pracovat jednoduše i s nekonečnými datovými strukturami (viz odst. 8).

Této výhody lze využívat zejména v referenčně transparentních jazycích, tedy v takových, kde hodnota výrazu nezávisí na jeho umístění v programu. Některé jazyky jako ML, OCaML, Erlang[6] nejsou referenčně transparentní a hodnota výrazů v jejich programech závisí na pořadí vyhodnocování (Věta 1 platí jen pro čistě funkcionální, tedy referenčně transparentní jazyky!). Při normální či líné strategii je velmi nesnadné určit pořadí vyhodnocování podvýrazů, nejsou-li předem známa vstupní data. Proto se v jazycích, které nejsou čistě funkcionální, obvykle používá striktní redukční strategie. Kromě toho je striktní redukční strategie výhodnější v menší spotřebě paměti při vyhodnocování.

Shrnutí

- Výrazy můžeme vyhodnocovat obecně mnoha způsoby. Nejdůležitější z nich jsou tzv. normální a striktní redukční strategie.
- Výběr redukční strategie neovlivní hodnotu výsledku, může však ovlivnit fakt, zda výsledek vůbec obdržíme.
- Normální redukční strategie je efektivně normalizující, tj. výpočet se nezacyklí, pokud nemusí.
- Zlepšením normální redukční strategie je líná redukční strategie, která se vyhýbá opakovanému vyhodnocování stejných podvýrazů.
- V moderních čistě funkcionálních jazycích se většinou používá líná redukční strategie.

[6] tyto jazyky nejsou *čistě funkcionální*

4 Typy

V moderních funkcionálních jazycích, k nimž patří i Haskell, má každý výraz, a tedy i každá hodnota, svůj *typ*. Například hodnota `True` má typ `Bool`, tj. typ všech logických hodnot, stejně jako hodnota `False` nebo jako výrazy `not True`, `x && y`. Hodnota 1 má typ `Int`, typ všech malých celých čísel.[7] Desetinná (počítačem rozlišitelná) čísla mají typ `Float`, znaky mají typ `Char`.

Typ můžeme zjednodušeně chápat jako množinu hodnot.[8]

Skutečnost, že výraz M má typ σ, zapisujeme $M::\sigma$. Má-li více výrazů, $M_1, \ldots, M_k$, stejný typ σ, lze psát $M_1, \ldots, M_k::\sigma$. Například

```
False, True :: Bool
'a', 'b' :: Char
1 :: Int
```

Tomuto zápisu se říká *typová anotace*. Definuje-li se nová hodnota, je dobrým zvykem ji typově anotovat:

```
pi :: Float
pi  = 3.14159265
prompt :: Char
prompt  = '?'
```

Typové anotace nejsou v Haskellu povinné. Předchozí definice mohly být zapsány bez nich, interpret nebo kompilátor si umí typy odvodit sám. Existuje však několik důvodů, proč typové anotace uvádět.

- zdrojový text programu je čitelnější,
- v typově anotovaném programu se odhalí více typových chyb (typová chyba ve výrazu může být taková, že se pozná, až když je odvozený typ srovnán s typem deklarovaným, a zjistí se rozdíl),
- v některých případech je výpočet podle programu s typovými anotacemi rychlejší, protože automaticky odvozený typ je příliš obecný a přeložený program by předpokládal výpočet i s hodnotami, které se pak nikdy nevyskytnou.

Kromě základních typů (`Bool`, `Float`,...) můžeme vytvářet typy složené pomocí tzv. *typových konstruktorů*. Nejdůležitějším typovým konstruktorem ve funkcionálním programování je typový konstruktor $\rightarrow$ (šipka, v Haskellu zapisovaná `(->)`). Jsou-li σ, τ dva typy, pak typ $\sigma \rightarrow \tau$ je typem všech (unárních) funkcí, jejichž argument je typu σ a funkční hodnota (výsledek) je typu τ.

Jsou-li ϱ, σ, τ tři typy, pak typ $\varrho \rightarrow \sigma \rightarrow \tau$ je typem všech binárních funkcí (tj. funkcí dvou proměnných), jejichž první argument je typu ϱ, druhý argument je typu σ a výsledek je typu τ.

[7] Vedle typu `Int` celých čísel zobrazitelných v jednom slově procesoru existuje typ `Integer` celých čísel, jejichž délka je omezena pouze velikostí paměti.

[8] Přesnější význam pojmu typ však zahrnuje i operace, jež s hodnotami daného typu pracují, viz kap. 10.

Jsou-li $\sigma_1, \sigma_2, \ldots, \sigma_n, \tau$ nějaké typy, pak $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_n \to \tau$ je typ všech n-árních funkcí, jejichž argumenty mají po řadě typy $\sigma_1, \ldots, \sigma_n$ a funkční hodnota je typu τ.

Například typové anotace některých operací v Haskellu vypadají takto:[9]

```
toUpper :: Char -> Char
not :: Bool -> Bool
(&&) :: Bool -> Bool -> Bool
odd :: Int -> Bool
(+) :: Int -> Int -> Int
(<) :: Int -> Int -> Bool
```

Podobně jako aplikaci binárních operátorů na dvě hodnoty lze zapisovat infixově i prefixově, lze i funkcionální typ `a -> b` zapisovat také `(->) a b`. Infixový zápis je však běžnější.

V následujících kapitolách poznáme další typové konstruktory různé arity, pomocí nichž se tvoří typy seznamů nebo typy uspořádaných n-tic. Nyní si však můžeme všimnout, že i základní typy jako `Bool` nebo `Int` jsou zvláštními případy typových konstruktorů – jsou to tzv. *nulární typové konstruktory*: abychom pomocí nich vyjádřili typ, stačí je napsat samy o sobě. Neaplikují se na žádný další typ (na rozdíl třeba od šipky, kterou musíme aplikovat na dva typy), proto jsou nulární.

Polymorfní typy

Zatím jsme se setkali jen s typy, které se nazývají *monomorfní*. Lze je použít vždy jen v kontextu jediného typu. Například monomorfní funkci (tj. funkci monomorfního typu) lze aplikovat vždy jen na argument jediného typu a typ jejího výsledku může být také jen jediný. Takovou monomorfní funkcí je třeba funkce `toUpper`. Její typ je `Char->Char`.

Funkce `id` je tzv. *identita*. Je definovaná

```
id x = x
```

Jaký typ má tato funkce? Kdybychom identitu aplikovali jen na celá čísla, měla by typ `Int->Int`. Kdybychom ji aplikovali jen na znaky, měla by typ `Char->Char`, kdyby byla aplikována na funkce typu `Float->Bool`, měla by sama typ `(Float->Bool) -> (Float->Bool)` atd. Chceme však, aby identita byla univerzálně aplikovatelná na jakoukoliv hodnotu jakéhokoliv typu. Jinými slovy, aby jednou byla typu `Int->Int`, podruhé typu `Char->Char`, potřetí `(Float->Bool) -> (Float->Bool)` a jindy jakéhokoliv jiného typu $a \to a$, kde a zastupuje *libovolný* typ.

Jazyky s *polymorfními typovými systémy* (a Haskell mezi ně patří) umožňují zavést tzv. *polymorfní typy*, v jejichž zápisu kromě typových konstruktorů vystupují i tzv. *typové proměnné*. Například zmíněná identita má typ

```
id :: a -> a
```

[9] Typy operátorů `odd`, `(+)`, `(<)` budou v kap. 10 ještě upřesněny.

V typu `a->a` je `a` typová proměnná zastupující libovolný typ. To znamená, že např. ve výrazu `id 3` se typová proměnná `a` *specializuje* na typ `Int` (a celý výraz `id 3` má typ `Int`), ve výrazu `id '*'` se specializuje na `Char` (a celý výraz má typ `Char`) atd.

Podobně jako `id` existují v Haskellu polymorfní funkce `const`, `flip` definované následovně.

```
const x y = x
flip f x y = f y x
```

Funkce `const` vrátí první ze svých dvou argumentů. Tyto dva argumenty však mohou být libovolného typu. Jediným omezením na typ funkce `const` tedy je, že výsledek musí být stejného typu jako první argument. Je tedy

```
const :: a -> b -> a
```

Všimněme si, že kdybychom místo `a -> b -> a` napsali `a -> a -> a`, řekli bychom tím, že oba argumenty funkce `const` musejí mít stejný typ. Tím bychom však typ funkce `const` zbytečně omezili, protože bychom vyloučili výrazy `const toUpper 5` apod.

Funkce `flip` vyžaduje, aby jejím prvním argumentem byla binární funkce, a této funkci obrací pořadí argumentů. Například `flip const True 8` $\rightsquigarrow$ `const 8 True` $\rightsquigarrow$ 8 nebo `flip (-) 3 8` $\rightsquigarrow$ `(-) 8 3` = `8-3` $\rightsquigarrow$ 5 Její typová anotace (přidělující funkci `flip` *nejobecnější* typ) je tedy

```
flip :: (a -> b -> c) -> b -> a -> c
```

Už dříve jsme se setkali s operátorem skládání funkcí `(.)`

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

Jeho (nejobecnější) typ je

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Tento polymorfní typ říká, že první dva argumenty operátoru `(.)` jsou funkce, ale typ argumentu prvé `b` musí být stejný jako typ výsledku druhé. Například ve výrazu `(not . odd) 4` má operátor `(.)` monomorfní typ (specializovaný z polymorfního) `(Bool->Bool)->(Int->Bool)->Int->Bool`, neboť typová proměnná `a` se specializuje na typ `Int` a typové proměnné `b`, `c` se specializují na typ `Bool`.

V zápisech typů v Haskellu rozeznáme typové proměnné od typových konstruktorů podle toho, že jejich jména vždy začínají malým písmenem. Jména typových konstruktorů jsou buď tvořena speciálními symboly (např. `(->)`), anebo začínají velkým písmenem (např. `Bool`, `Char`).

Shrnutí

- Každá hodnota v Haskellu má svůj typ. Typy hodnot vyjadřujeme typovými anotacemi.

- Nejjednodušší typy jsou vyjádřeny nulárními typovými konstruktory (`Bool` , `Int` , `Float` , `Char`).
- Složitější typy se tvoří ze základních pomocí dalších typových konstruktorů, z nichž nejdůležitější je `(->)` .
- Typy mohou být polymorfní – v jejich zápisu vystupují typové proměnné.

5 Datové struktury

n-tice

Jsou-li M a N výrazy, pak výraz `(`M`,`N`)` označuje *uspořádanou dvojici* výrazů M, N. Podobně `(`M`,`N`,`P`)` je *uspořádaná trojice* výrazů M, N, P atd. V některých jazycích[10] existuje dokonce speciální hodnota, *nultice* `()`.

Složky uspořádaných dvojic se zpřístupní pomocí funkcí `fst`, `snd`.

```
fst (x,y)  =  x
snd (x,y)  =  y
```

V těchto definicích jsou použity uspořádané dvojice `(x,y)` jako vzor. To znamená, že funkce `fst`, `snd` jsou definovány na uspořádaných dvojicích, jejichž složky mohou být libovolné (vzor `x` resp. `y` je proměnná, tj. vyhovuje všem výrazům).

Složky uspořádané n-tice mohou být různého typu. *Typ* uspořádané dvojice, trojice atd. vyjadřujeme *typovým konstruktorem*, který representuje kartézský součin typů složek. Zapisuje se opět pomocí závorek:[11]

```
(1,True)  ::  (Int,Bool)
(False,('%',0),3.14159)  ::  (Bool,(Char,Int),Float)
(gcd,not)  ::  (Int->Int->Int, Bool->Bool)
```

Binární operátor, který vytváří uspořádanou dvojici, se jmenuje *konstruktor uspořádané dvojice* a značí se `(,)`. Aplikován na výrazy M, N tvoří dvojici `(`M`,`N`)`, tj.

$$\texttt{(,)}\ M\ N = \texttt{(}M\texttt{,}N\texttt{)}$$

Druhý způsob zápisu `(`M`,`N`)` je však obvyklejší než prefixový.[12] Podobně máme ternární konstruktor `(,,)` uspořádaných trojic atd.

```
(,)  :: a -> b -> (a,b)
(,,) :: a -> b -> c -> (a,b,c)
```

Příklad: Uspořádané dvojice v programu by mohly představovat třeba racionální čísla – první složka čitatel, druhá jmenovatel zlomku. Pak by sčítání `(+/)` mohlo být definováno takto:[13]

```
(a,b) +/ (c,d)  =  (e/g,f/g) where e = a*d+b*c
                                   f = b*d
                                   g = gcd e f
```

[10] Např. v ML, a to ze dvou důvodů: ML má striktní vyhodnocování, ale „líné konstanty“ typu σ v nich můžeme simulovat funkcí typu `()` $\to \sigma$, a za druhé, ML dovoluje vedlejší efekty, takže např. přiřazovací „výraz-příkaz“ nemusí nic vracet: má v ML typ `Ref` $\sigma \to \sigma \to$ `()`.

[11] Značení je převzato z jazyka Haskell. Nemůže dojít k nedorozumění, protože typ se může v programu vyskytnout nanejvýš jako anotace (pod)výrazu za značkou `::`.

[12] Zápis `(`M`,`N`)` není, přesně vzato, infixový, protože kulaté závorky jsou zde povinnou součástí syntaxe. Tomuto zápisu se někdy říká „mixfix“.

[13] Funkce `gcd` počítá největšího společného dělitele svých argumentů.

Seznamy

Uspořádané n-tice mají pevný počet složek, které mohou být různého typu. Naproti tomu *seznam* je posloupnost, v níž mají všechny prvky stejný typ.

Existují dvě základní operace, pomocí nichž se tvoří seznamy, tzv. *seznamové konstruktory*. První z nich je konstanta (tj. nulární konstruktor) `[]`, která označuje prázdný seznam. Druhou operací je binární konstruktor `(:)`, který přidává prvek do seznamu: je-li s seznam prvků a x je hodnota téhož typu jako mají prvky seznamu s, pak x`:`s je seznam, který vznikne ze seznamu s přidáním prvku x na jeho začátek.

Prázdný seznam je tedy `[]`, jednoprvkový seznam obsahující dvojku bude `2:[]`, dvouprvkový seznam čísel 1, 2 je `1:(2:[])`, atd. Dohodneme se, že operátor `(:)` sdružuje operandy zprava. To znamená, že například tříprvkový seznam čísel 1, 2, 3 můžeme namísto `1:(2:(3:[]))` zapisovat `1:2:3:[]`.

Je také možno zapisovat výčtem seznamy tak, že se jejich prvky, oddělené čárkami, zapíší mezi hranaté závorky. Následující zápisy jsou tedy ekvivalentní:

```
3:[]       ≡   [3]
2:3:[]     ≡   2:[3]     ≡   [2,3]
1:2:3:[]   ≡   1:2:[3]   ≡   1:[2,3]   ≡   [1,2,3]
```

Navíc, speciálně pro seznamy znaků, tj. seznamy typu `[Char]`, můžeme použít notace, v níž znaky zapisujeme bez oddělovačů za sebe a seznam uzavřeme do uvozovek (`"`):

```
[]              ≡                    ≡   ""
'c':[]          ≡   ['c']            ≡   "c"
'b':'c':[]      ≡   ['b','c']        ≡   "bc"
'a':'b':'c':[]  ≡   ['a','b','c']    ≡   "abc"
```

Podle toho, který ze dvou seznamových konstruktorů je na nejvyšší úrovni, poznáme u seznamu, zda je prázdný:

```
null []     =  True
null (x:s)  =  False
```

Funkce `null` testuje svůj argument (seznam) na prázdnost a je definována podle vzorů: prázdnému seznamu odpovídá pouze první vzor, `[]`, neprázdnému seznamu odpovídá pouze druhý vzor, `(x:s)`.

Protože proměnné z druhého vzoru jsme na pravé straně definice nepoužili, nemuseli jsme je vůbec pojmenovávat. „Bezejmennou“ proměnnou značíme symbolem `_` (podtržítko). Její význam je „cokoliv“. Vyskytne-li se ve vzoru vícekrát, může každý její výskyt odpovídat jinému výrazu. Druhý vzor jsme tedy mohli zapsat `(_:_)`.

Definice podle vzorů, odpovídajících jednotlivým konstruktorům, je u seznamů častá.

```
length []     =  0
length (_:s)  =  1 + length s
```

Funkce `length` počítá délku (počet prvků) seznamu.

Nepokrývají-li vzory v definici všechny možné tvary argumentu, zůstává funkce nedefinovaná pro ty argumenty, které neodpovídají žádnému vzoru v její definici. Například funkce `head` resp. `tail` vrací první prvek seznamu resp. tzv. *zbytek* seznamu (seznam bez prvního prvku).

```
head       ::  [a] -> a
head (x:_)  =  x

tail       ::  [a] -> [a]
tail (_:s)  =  s
```

Hodnoty výrazů `head []` a `tail []` nejsou definovány.

Další užitečné funkce pro práci se seznamy

Binární operátor `(!!)` vybírá ze seznamu prvek daného indexu, přičemž prvky jsou indexovány od nuly. Například `[3,1,7]!!2` $\rightsquigarrow^*$ 7.

```
(!!)       ::  [a] -> Int -> a
(x:_) !! 0  =  x
(_:s) !! k  =  s !! (k-1)
```

Funkce `(!!)` je ovšem definována jen pro nezáporný druhý argument.[14]

Důležitými funkcemi jsou `map` a `filter`. Funkce `map` umožňuje aplikovat funkci `f` na všechny prvky seznamu. Například hodnotou výrazu `map square [1,5,11]` bude `[1,25,121]`, `map toUpper "abc"` vrátí hodnotu `"ABC"`, a výraz `map even [2,3,5]` se vyhodnotí na `[True,False,False]`.

```
map         ::  (a->b) -> [a] -> [b]
map _ []     =  []
map f (x:s)  =  f x : map f s
```

Funkce `filter` „přefiltruje“ seznam (svůj druhý parametr) vybírajíc z něho pouze prvky splňující danou podmínku (zadanou jako první parametr). Například `filter odd [1,1,2,5,12,35]` má hodnotu `[1,1,5,35]`.

```
filter         ::  (a->Bool) -> [a] -> [a]
filter _ []     =  []
filter p (x:s)  =  if  p x  then  x:s'  else s'
                   where s' = filter p s
```

Funkce `take` vrátí prvých n (první parametr) prvků seznamu (druhý parametr). Má-li seznam méně než n prvků, vrátí ho funkce celý.

```
take         ::  Int -> [a] -> [a]
take 0 _      =  []
take _ []     =  []
take n (x:s)  =  x : take (n-1) s
```

[14] Další podmínku, že její první argument musí být seznam a druhý argument celé číslo, považujeme za implicitní a vyplývající z typu funkce. Nesplnění takové podmínky se totiž pozná už v době čtení výrazu při typové analýze, mimo vlastní výpočet.

Podobně funkce `drop` vrátí zbytek seznamu po odstranění prvých n prvků, je-li seznam „příliš krátký“, bude výsledkem prázdný seznam.

```
drop         ::  Int -> [a] -> [a]
drop 0 s      =  s
drop _ []     =  []
drop n (_:s)  =  drop (n-1) s
```

Podobnou roli jako funkce `map` pro unární operace hraje funkce `zipWith` pro binární operace.

```
zipWith                 :: (a->b->c) -> [a] -> [b] -> [c]
zipWith op (x:s) (y:t)  =  op x y : zipWith op s t
zipWith _  _     _      =  []
```

Funkce aplikuje operaci `op` na odpovídající si prvky obou seznamů a z výsledků těchto aplikací vytvoří seznam. Například `zipWith (*) [1,2,3] [1,4,9]` bude po vyhodnocení seznam `[1,8,27]`.

Poslední řádek definice funkce `zipWith` „odchytí“ případy, kdy je první nebo druhý seznam prázdný. Nemusejí být prázdné oba současně, to znamená, že funkci `zipWith` lze aplikovat i na dva nestejně dlouhé seznamy. Délka výsledného seznamu bude rovna délce kratšího z nich.

Funkci `zip`, která ze dvou seznamů vytváří seznam dvojic, lze definovat

```
zip             ::  [a] -> [b] -> [(a,b)]
zip (x:s) (y:t)  =  (x,y) : zip s t
zip _     _      =  []
```

anebo šikovněji, pomocí už známé funkce `zipWith`, takto:

```
zip s t  =  zipWith (,) s t
```

nebo dokonce, jak uvidíme v kap. 6, takto:

```
zip  =  zipWith (,)
```

Časová složitost (příklad)

Operátor `(++)` spojuje dva seznamy stejného typu.

```
(++)       ::  [a] -> [a] -> [a]
[]    ++ t  =  t
(x:s) ++ t  =  x : (s ++ t)
```

Nyní spočteme, kolik redukčních kroků je potřeba ke spojení dvou seznamů, s `++` t. Všimněme si, že na tvaru druhého parametru délka výpočtu (tj. počet redukčních kroků) nezávisí. První parametr se však při výpočtu „zkracuje“. Označme jeho délku

n. Dále označme $c_a(n)$ hledanou dobu výpočtu (tj. počet kroků potřebných k vyhodnocení výrazu s `++` t, je-li n délka seznamu s). Zřejmě $c_a(0) = 1$, a pro $n > 0$ je $c_a(n) = 1 + c_a(n-1)$. Dohromady tedy $c_a(n) = n + 1$ pro každou délku n.

Funkce `reverse'` obrátí pořadí prvků v seznamu:

```
reverse'       ::  [a] -> [a]
reverse' []     =  []
reverse' (x:s)  =  reverse' s ++ [x]
```

Tato rekursivní definice je zřejmá. Případ pro prázdný seznam je triviální, a je-li `reverse'` s obrácením seznamu s, pak seznam x`:`s obrátíme tak, že na konec obráceného seznamu s připojíme prvek x.

Zjistíme, kolik redukčních kroků je potřeba k obrácení seznamu délky n. Toto číslo označíme $c'_r(n)$.

Je-li $n = 0$, pak potřebujeme krok jediný, tedy $c'_r(0) = 1$. Nechť $n > 0$. Pak seznam, který obracíme, je neprázdný. Nechť je roven `[`x_1`,`x_2`,...,`x_n`]`.

Pomocí $n + 1$ kroků ho převedeme do tvaru
`(...(([] ++ [`x_n`]) ++ [`x_{n-1}`]) ++ ⋯ ++ [`x_2`]) ++ [`x_1`]`
podle druhé klausule v definici `reverse'`.

Dalšími $c_a(0)+c_a(1)+\cdots+c_a(n-2)+c_a(n-1) = 1+2+\cdots+(n-1)+n = \frac{n^2+n}{2}$ kroky ho podle definice operace `(++)` převedeme do výsledného tvaru `[`x_n`,`x_{n-1}`,...,`x_2`,`x_1`]`.

Celkem tedy potřebujeme

$$c'_r(n) = n + 1 + \frac{n^2+n}{2} = \frac{n^2+3n+2}{2}$$

kroků. Tento výsledek platí i pro případ $n = 0$.

Vidíme tedy, že počet redukčních kroků nutných k obrácení seznamu pomocí funkce `reverse'` je přímo úměrný druhé mocnině jeho délky. Říkáme, že funkce `reverse'` má *kvadratickou časovou složitost*.

To není nejlepší výsledek: seznamy lze obracet rychleji. Uvažme funkci `reverse`,

```
reverse   ::  [a] -> [a]
reverse u  =  rev [] u  where  rev s []    = s
                               rev s (x:t) = rev (x:s) t
```

Funkce `reverse` obrací seznam tak, že jeho prvky jeden po druhém „přeskládává“ v parametru pomocné funkce `rev`.

Označme $c_r(n)$ počet kroků nutných k obrácení seznamu délky n pomocí funkce `reverse`. Podobně označme $c_v(n)$ počet kroků nutných k přeskládání n-prvkového seznamu t na začátek seznamu s při výpočtu výrazu `rev` s t.

Zřejmě $c_r(n) = 1 + c_v(n)$. Kromě toho $c_v(0) = 1$, a pro $n > 0$ je $c_v(n) = 1 + c_v(n-1)$. Celkem dostáváme, že pro každou délku n je $c_v(n) = n + 1$ a $c_r(n) = n + 2$.

Délka výpočtu pomocí funkce `reverse` je tedy přímo úměrná délce obraceného seznamu. Říkáme, že funkce `reverse` má *lineární časovou složitost*.

Shrnutí

- Mezi základní datové struktury patří n-tice a seznamy.
- Složky n-tice mohou mít různý typ, všechny prvky seznamu musí být stejného typu.
- n-tice mají jeden konstruktor, seznamy se tvoří pomocí dvou konstruktorů.
- Funkce pracující s n-ticemi a se seznamy s výhodou definujeme podle vzorů.
- Při definování funkce se vyplatí provést analýzu její časové složitosti a snažit se tuto složitost minimalizovat.

6 Funkce vyššího řádu

Částečná aplikace

Nyní si poněkud zpřesníme pohled na funkce, kterým říkáme „funkce více proměnných“ a na aplikace takových funkcí na argumenty.

Přesně vzato, všechny funkce jsou pouze jednoparametrické – unární.[15] Aplikace na n argumentů

$$f \quad x_1 \quad x_2 \quad \ldots \quad x_{n-1} \quad x_n$$

je ve skutečnosti jednoduchou aplikací funkce $((\ldots((f \quad x_1)\ x_2)\ldots)\ x_{n-1})$ na jediný argument x_n, tj.

$$((\ldots((f \ x_1)\ x_2)\ \ldots\ )\ x_{n-1})\ x_n$$

„Neviditelný operátor aplikace“ tedy vlastně sdružuje zleva.

Podobně, typový konstruktor $\to$ v typových výrazech sdružuje zprava, takže zápis

$$\sigma_1 \to \sigma_2 \to \cdots \to \sigma_{n-1} \to \sigma_n \to \sigma$$

ve skutečnosti znamená

$$\sigma_1 \to (\sigma_2 \to \cdots \to (\sigma_{n-1} \to (\sigma_n \to \sigma))\ldots)$$

Například ve výrazu `(+) 3 4` se operátor `(+)` nejdříve aplikuje na svůj (jediný) argument 3 (tj. provede se aplikace `(+) 3`) a výsledkem této aplikace je funkce „přičítač trojky“ – funkce, která, je-li aplikována na argument (např. na čtyřku: `((+) 3) 4`), přičte k němu číslo 3, a výsledný součet bude výsledkem této druhé aplikace.

Situaci znázorňuje tabulka:[16]

výraz	typ	hodnota
`(+)`	`Int`→`Int`→`Int`	funkce, která, když ji aplikujeme na číslo x, vrátí funkci f, která, až bude aplikována na jiné číslo, bude k němu přičítat x
`(+) 3`	`Int`→`Int`	„přičítač trojky“, tj. funkce, která, když bude aplikována na nějaké číslo, vrátí číslo o 3 větší
`(+) 3 4`	`Int`	číslo 7 (výsledek aplikace „přičítače trojky“ na číslo 4)

Částečná aplikace se říká aplikaci nějaké funkce na „menší počet argumentů než je obvyklé“. Přesněji, je to aplikace, jejímž výsledkem je funkce. Například v definici

```
add3  =  (+) 3
```

je operátor `(+)` částečně aplikován na argument `3`.

V kap. 5 jsme měli definici

[15] Vedle konstant, což jsou vlastně nulární funkce. V některých jazycích se dokonce i konstanty pokládají za unární funkce typu `()` $\to \sigma$.

[16] Typ operátoru `(+)` je zde trochu zjednodušen, ve skutečnosti je operátor „přetížen“, viz kap. 10.

```
zip s t  =  zipWith (,) s t
```

Funkce `zipWith` je zde aplikována na argumenty `(,)`, `s`, `t`, tedy funkce `zipWith (,)` je aplikována na argumenty `s`, `t`, tedy funkce `zipWith (,) s` je aplikována na argument `t`. Funkce `zip` je zde definována předpisem, který říká, „jak se `zip` chová“ na argumentech `s` a `t`: chová se stejně jako funkce `zipWith (,)` na argumentech `s` a `t`. Proto je funkce `zip` rovna funkci `zipWith (,)` a původní definici můžeme zkrátit[17] na tvar

```
zip  =  zipWith (,)
```

Tím máme definovánu funkci `zip` pomocí částečné aplikace funkce `zipWith`.

Podobně jsme mohli funkci `reverse` v kap. 5 definovat pomocí částečné aplikace pomocné funkce `rev` na prázdný seznam:

```
reverse  =  rev []  where rev s []    = s
                          rev s (x:t) = rev (x:s) t
```

Uveďme si další jednoduchý příklad. Nechť `f` je funkce definovaná takto:

```
f x y z   =   if  x == y  then  y  else  z
```

Její typ je `a→a→a→a`.[18] Částečná aplikace `f False` však už není polymorfní, protože typ argumentu `False` vynucuje `a` = `Bool`. Její typ je tedy

```
f False  ::  Bool -> Bool -> Bool
```

Nás však více zajímá její hodnota. Podle předpisu, `f False` je funkce, která, je-li aplikována na argumenty y, z, vrátí hodnotu `(if False==`y `then` y `else` z`)`. Když vyšetříme výsledek pro všechny možné hodnoty y, z typu `Bool`, zjistíme, že se tato funkce chová stejně jako logická konjunkce, tj. `(&&)` = `f False`.

Podobně, částečnou aplikací funkce `f` na argument `True` dostaneme funkci

```
f True  ::  Bool -> Bool -> Bool
```

která je rovna[19] logické disjunkci, `(||)` = `f True`.

Akumulační seznamové funkce

Další příklady na částečnou aplikaci poskytnou tzv. akumulační seznamové funkce `foldr` a `foldl`. Pomocí těchto funkcí můžeme aplikovat nějakou binární operaci na všechny prvky seznamu navzájem, např. `foldl (+) 0 [`x_1`,...,`x_n`]` $= x_1 + \cdots + x_n$.

Všimněme si podobnosti následujících funkcí pracujících se seznamy:

[17] Takovému zkrácení se říká *η-redukce*. Vyplývá z tzv. *principu extensionality*, který říká: Jestliže platí $f\ x = g\ x$ pro všechna x ze sjednocení definičních oborů funkcí f a g, pak $f = g$.

[18] V kap. 10 uvidíme, že v tomto případě je třeba polymorfní typ `a` omezit na třídu typů „připouštějících rovnost“, to však v našem příkladě není podstatné.

[19] Tato rovnost je pouze *denotační*, v Haskellu lze testovat na rovnost např. čísla nebo znaky, ale nikoliv funkce.

```
and       ::  [Bool] -> Bool
and []     =  True
and (x:s)  =  x && and s

or       ::  [Bool] -> Bool
or []     =  False
or (x:s)  =  x || or s

concat       ::  [[a]] -> [a]
concat []     =  []
concat (s:t)  =  s ++ concat t

compose       ::  [a -> a]  ->  a -> a
compose []     =  id
compose (f:s)  =  f . compose s
```

Funkce `and` resp. `or` počítá logickou konjunkci resp. disjunkci prvků seznamu, funkce `concat` zřetězí seznamy ze seznamu seznamů a funkce `compose` provádí skládání funkcí ze seznamu. Všechny čtyři definice mají podobnou strukturu. To můžeme zobecnit zavedením vyšší funkce `foldr`.

```
foldr            ::  (b -> a -> a) -> a -> [b] -> a
foldr _  a []     =  a
foldr op a (x:s)  =  x `op` foldr op a s
```

a předchozí čtyři funkce definovat kratčeji pomocí částečných aplikací funkce `foldr` na vhodné argumenty.

```
and      =  foldr (&&) True
or       =  foldr (||) False
concat   =  foldr (++) []
compose  =  foldr (.)  id
```

Aplikujeme-li funkci `foldr` na argumenty `(`$\oplus$`)`, a, `[`x_1`,`x_2`,...,`x_{n-1}`,`x_n`]`, dostaneme (postupným rozvinutím podle definice)

$$x_1 \oplus (x_2 \oplus \cdots \oplus (x_{n-1} \oplus (x_n \oplus a))\ldots)$$

Funkce `foldr` tedy aplikuje operátor `(`$\oplus$`)` na všechny prvky seznamu a na hodnotu a, sdružujíc operandy zprava.

To odpovídá tomu, co požadujeme od funkcí `and`, `or`, `concat`, `compose`.

`and [`x_1`,`x_2`,...,`x_n`]` = x_1 `&&` $(x_2$ `&&` $\ldots$ `&&` $(x_n$ `&& True`$)\ldots)$
`or [`y_1`,`y_2`,...,`y_n`]` = y_1 `||` $(y_2$ `||` $\ldots$ `||` $(y_n$ `|| False`$)\ldots)$
`concat [`s_1`,`s_2`,...,`s_n`]` = s_1 `++` $(s_2$ `++` $\ldots$ `++` $(s_n$ `++ []`$)\ldots)$
`compose [`f_1`,`f_2`,...,`f_n`]` = f_1 `.` $(f_2$ `.` $\ldots$ `.` $(f_n$ `. id`$)\ldots)$

Podobně jako `foldr`, definujeme funkci `foldl`.

```
foldl     ::  (a -> b -> a) -> a -> [b] -> a
foldl _  b []     =  b
foldl op b (y:t)  =  foldl op (b `op` y) t
```

Aplikujeme-li ji na argumenty `(⊗)`, b, `[`y_1`,`y_2`,...,`y_{n-1}`,`y_n`]`, dostaneme

$$(\ldots((b \otimes y_1) \otimes y_2) \otimes \cdots \otimes y_{n-1}) \otimes y_n$$

Funkce `foldl` tedy funguje tak, že aplikuje operátor `(⊗)` na hodnotu b a na všechny prvky seznamu, ale tentokrát operandy sdružuje zleva.

Akumulační seznamové funkce `foldr`, `foldl` tedy mají podobný efekt, ale `foldr` závorkuje operandy zprava, `foldl` závorkuje zleva. Díky tomu mají i odlišné typy.

Funkce `and`, `or`, `concat`, resp. `compose` jsou definovány pomocí `foldr` a operací `(&&)`, `(||)`, `(++)`, resp. `(.)` a pomocí konstant `True`, `False`, `[]`, resp. `id`. Jelikož všechny tyto čtyři operace jsou asociativní (nezáleží na směru sdružování operandů), mají oba operandy stejného typu a zmíněné konstanty jsou jejich neutrálními prvky, bylo by možné zavést funkce `and`, `or`, `concat` a `compose` i pomocí akumulační seznamové funkce `foldl`. V Haskellu je v těchto definicích použita funkce `foldr`, protože je jednodušší a někdy vede k paměťově efektivnějšímu výpočtu.

Vraťme se k funkci `reverse` z kap. 5. Pomocná funkce `rev` v ní byla definována

```
rev s []     =  s
rev s (x:t)  =  rev (x:s) t
```

dala se tedy definovat pomocí `foldl`:

```
rev  =  foldl flipcons   where flipcons s x = x:s
```

Když ještě využijeme standardní funkce `flip`, která „obrací pořadí argumentů“,

```
flip f x y  =  f y x
```

můžeme celou funkci `reverse` definovat elegantně takto:

```
reverse  =  foldl (flip (:)) []
```

Currifikace

Uvažme dvě funkce:

```
add        ::  Int -> Int -> Int
add  x  y   =  x + y

add'       ::  (Int,Int) -> Int
add' (x,y)  =  x + y
```

Obě funkce jsou různé (mají různý typ), ale obě realizují sčítání celých čísel. Funkce `add` je vyšší funkce – po aplikaci na argument vrátí funkci (například `add 3` je známý

„přičítač trojky“), funkce `add'` vyžaduje na místě argumentu dvojici a vrací číslo. Výhoda první z nich je právě v tom, že ji můžeme částečně aplikovat na jeden argument.

Kdybychom chtěli v nějakém výrazu použít „přičítač trojky“ vyjádřený pomocí funkce `add`, stačilo by napsat `add 3`. Kdybychom totéž chtěli vyjádřit pomocí funkce `add'`, potřebovali bychom k tomu složitější zápis, např. zavedení další pomocné funkce.

Funkci `add` můžeme definovat pomocí `add'` předpisem

```
add x y  =  add' (x,y)
```

Obecně, z každé funkce f', jejímž argumentem je n-tice, můžeme vytvořit vyšší funkci, kterou lze postupně aplikovat na n argumentů odpovídajících složkám n-tice. Tomuto převodu nižších funkcí na vyšší se říká *currifikace*[20]

Mějme dvě funkce, f, f', definované pomocí stejného výrazu e:

f `x y` `=` e

f' `(x,y) =` e

Pak přechod od funkce f' k funkci f obstarává funkce `curry`, opačný směr funkce `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
```

Tedy $f =$ `curry` f' a $f' =$ `uncurry` f .

Situaci můžeme přirovnat k rozdílu mezi kompilátorem a interpretem programovacího jazyka. Interpret zpracovává zdrojový text programu i jeho vstupní data naráz a rovnou produkuje výsledky výpočtu. Kompilátor je vyšší funkce: aplikujeme-li ho na zdrojový text programu, vytvoří funkci (přeložený kód programu), kterou (později) aplikujeme na vstupní data, abychom získali výsledky. Výhoda kompilace (částečné aplikace) se projeví, budeme-li chtít tentýž program spouštět mnohokrát na různých vstupních datech. Dá se tedy zjednodušeně říci

$$\textit{kompilátor} = \texttt{curry}\ \ \textit{interpret}$$

Shrnutí

- Máme pouze unární funkce – funkcionální aplikace je aplikace funkce na *jeden* argument.
- Také funkce, kterým se pro jednoduchost říká „n-ární“, jsou unární funkce. Po aplikaci na svůj argument vrátí $(n-1)$-ární funkci.

[20] Podle logika Haskella B. Curryho. Tvar currifikace zde používáme místo pravopisně správnějšího tvaru curryifikace.

- Částečná aplikace je aplikace n-ární funkce na k argumentů, $k < n$ (tj. vlastně k postupných aplikací). Částečné aplikace umožňují přímo pracovat s funkcemi vyšších řádů.

- Jiný způsob zacházení s víceparametrickými funkcemi je spojit jejich parametry do n-tic. Tím se však ztrácí možnost takto definované funkce částečně aplikovat. Tyto „n-ticové“ funkce se dají převést na vyšší funkce procesem zvaným currifikace.

7 Funkcionální abstrakce

Výrazy s let

V kap. 2 a dále jsme viděli použití lokálních definic uvnitř `let` výrazů nebo pomocí konstrukce `where`. Například obě definice

```
reverse  =  foldl fc  where fc = flip (:)

reverse  =  let  fc = flip (:)  in  foldl fc
```

jsou navzájem ekvivalentní.

Zatímco však zápisu lokální definice s `where` lze použít jen v nadřazené definici, zápis s `let` lze použít kdekoliv na místě výrazu, např.

```
( let  fc = flip (:)  in  foldl fc )  "jelen"
```

Lambda abstrakce

Pojem *abstrakce* obecně znamená *oddělení od speciálního (konkrétního) případu.* S funkcionální abstrakcí se setkáváme pokaždé, když zavádíme novou funkci. Například chceme-li každý prvek celočíselného seznamu `[a,b,c,d,e]` vynásobit třemi a přičíst jedničku, můžeme buď výsledný seznam zapsat jako `[3*a+1,3*b+1,3*c+1,3*d+1,3*e+1]`, anebo z jednotlivých výrazů abstrahovat funkci, pojmenovat ji třeba `f`

```
f x = 3 * x + 1
```

a výsledný seznam pak zapsat jako `[f a, f b, f c, f d, f e]`, anebo dokonce jen `map f [a,b,c,d,e]`.

To však není jediná možnost funkcionální abstrakce. V uvedeném příkladu jsme nové funkci dali jméno, `f`, abychom se jím později mohli na funkci odvolávat. Lze však vytvářet i *bezejmenné* funkcionální abstrakce. To znamená zapsat výrazem funkci, která nebude mít jméno – bude vyjádřena právě jen tímto výrazem.

Výraz

```
let  f x = 3 * x + 1  in  map f [a,b,c,d,e]
```

lze ekvivalentně zapsat

```
map (\ x -> 3*x+1) [a,b,c,d,e]
```

Výraz `(\x->3*x+1)` je tzv. bezejmenná funkcionální abstrakce, neboli *lambda abstrakce.*

Obecně, máme-li funkci f jedné proměnné x definovanou pomocí výrazu E

$f\ x = E$

pak funkci f lze vyjádřit lambda abstrakcí $\lambda x \to E$. Lambda abstrakce jsou vedle aplikací základními (a jedinými) způsoby vytváření složených výrazů v tzv. *lambda kalkulu*[21]. V Haskellu je syntax téže lambda abstrakce `\`x`->`E.

Lambda abstrakce `(\x->3*x+1)` je funkcí, která svůj číselný argument vynásobí třemi a přičte jedničku, tj. např. `(\x->3*x+1) 2` $\rightsquigarrow$ 7. Jiné příklady lambda abstrakcí jsou `(\c -> (toLower c,toUpper c))` nebo `(\g -> g . g)`. První z nich vyjadřuje funkci typu `Char`$\to$`(Char,Char)`, která po aplikaci na písmeno vrátí uspořádanou dvojici písmen (malé, velké), např. `(\c -> (toLower c,toUpper c)) 'E'` $\rightsquigarrow$ `('e','E')`. Druhá (typu `(a` $\to$ `a)` $\to$ `(a` $\to$ `a)`) po aplikaci na funkci vrátí tuto funkci složenu samu se sebou, takže např. `(\g -> g . g) not` je identická funkce na logických hodnotách: `(\g -> g . g) not True` $\rightsquigarrow$ `True`.

V kapitole 6 jsme viděli, že funkce vyšší arity jsou vlastně unárními vyššími funkcemi (funkcemi, které po aplikaci na argument vracejí funkce). Takže například funkci „nezáporného odčítání“ z příkladu v kap. 2 lze zapsat výrazem
`\x -> (\y -> if x>y then x-y else 0)` Můžeme však použít jednodušší syntaxe
`\x y -> if x>y then x-y else 0`

Zápis vícenásobných lambda abstrakcí tvaru $\lambda x_1 \to (\lambda x_2 \to (\ldots(\lambda x_n \to E)\ldots))$ zkracujeme na $\lambda x_1\ x_2\ \ldots\ x_n \to E$. Aplikace takové vícenásobné abstrakce na argumenty[22] je

$$\begin{array}{rl}
 & (\lambda x_1 \to (\lambda x_2 \to (\ldots(\lambda x_n \to E[x_1, x_2, \ldots, x_n])\ldots)))\ M_1 M_2\ \ldots M_n \\
\rightsquigarrow & \quad (\lambda x_2 \to (\ldots(\lambda x_n \to E[M_1, x_2, \ldots, x_n])\ldots))\ M_2\ \ldots M_n \\
\rightsquigarrow & \\
\vdots & \\
\rightsquigarrow & \quad (\lambda x_n \to E[M_1, M_2, \ldots, x_n])\ M_n \\
\rightsquigarrow & \quad E[M_1, M_2, \ldots, M_n]
\end{array}$$

čili kratčeji

$$(\lambda x_1\ \ldots\ x_n \to E[x1, \ldots, x_n])\ M_1\ \ldots\ M_n \rightsquigarrow^n E[M_1, \ldots, M_n]$$

Pro vícenásobné abstrakce je odpovídající syntax v Haskellu `\x y z t -> e`.

Shrnutí

- Konstrukce `let` umožňuje lokálně definovat hodnoty uvnitř výrazů.
- Nepojmenované (anonymní) funkce lze vyjádřit lambda abstrakcemi.
- Pomocí lambda abstrakcí můžeme vyjádřit i funkce vyšší arity.

[21] Lambda kalkul je matematický formalismus pro popis vyčíslitelných funkcí, který zavedl Alonzo Church. Je jakýmsi „funkcionálním prajazykem“.
[22] tzv. β-redukce

8 Seznamy

Intenzionální seznamová notace

Jak známo, množiny se v matematice zapisují dvěma odlišnými způsoby. Tzv. *extensionální* zápis je zápis množiny výčtem prvků, například

$$\{1, 4, 9, 16\}, \quad \{1\}, \quad \emptyset$$

Tzv. *intenzionální* zápis množiny ji popisuje jejími vlastnostmi. Ty se obvykle vyjádří obecným tvarem prvku (výrazem) a podmínkami, jež musí prvky splňovat. Intenzionální zápis množin uvedených výše je třeba

$$\{n^2 \mid n \in \{1, 2, 3, 4\}\}$$
$$\{n^2 \mid n \in \{1, 2, 3, 4\},\ n^3 < 8\}$$
$$\{n \mid n \in \{1, 2, 3, 4\},\ n^2 + 1 < 2^n\}$$

Podobným způsobem můžeme zapisovat seznamy. Extensionální zápis seznamů je obvyklý zápis ve tvaru

`1:4:9:16:[]`, `1:[]`, `[]`

nebo

`[1,4,9,16]`, `[1]`, `[]`

tj. pomocí seznamových konstruktorů `[]` a `(:)` (ať už vyjádřených explicitně jako v prvém anebo implicitně jako v druhém případě, oba zápisy jsou ekvivalentní).

Intenzionální zápis těchto seznamů se podobá množinovému intenzionálnímu zápisu.

```
[ n^2 | n<-[1..4] ]
[ n^2 | n<-[1..4], n^3<8 ]
[ n | n<-[1..4], n^2<2^n ]
```

Obecně má intenzionální zápis seznamu tvar

`[` *výraz* `|` *kvalifikátor* `,` ... `,` *kvalifikátor* `]`

a jeho význam je: seznam prvků daných *výraz*em, který může záviset na proměnných určených *kvalifikátor*y.

Kvalifikátory mohou být trojího druhu:

generátory jsou „zdrojem“ pro vytváření seznamu. Mají tvar

vzor `<-` *seznam*

> Ze *seznam*u se postupně vybírají prvky vyhovující *vzor*u. Přitom dojde k vazbě proměnných ze *vzor*u na hodnoty, které se použijí pro vytváření prvků výsledného seznamu.
>
> Například
>
> `[[n] | n <- [1..3]]` $\rightsquigarrow^*$ `[[1],[2],[3]]`

(vzorem je v tomto případě proměnná `n`), nebo

```
[x+1 | [x] <- [[],[1,3],[7],[],[5]] ⇝* [8,6]
```

(vzorem je výraz `[x]`).

Jestliže je v zápisu více generátorů, pak se generátory více vpravo mění rychleji než generátory více vlevo:

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

filtry jsou podmínky – logické výrazy. Prvek vygenerovaný předcházejícími generátory je do výsledného seznamu zařazen pouze tehdy, je-li podmínka splněna. Například

```
[n^2 | n <- [1..4], even n] ⇝* [4,16]
```

neboť pouze čísla 2 a 4 z generujícího seznamu jsou sudá.

lokální definice slouží pro zavedení lokálních proměnných. Mají tvar

`let` *vzor* `=` *výraz*

Lokální definice se vztahuje na kvalikátory, které za ní následují, a na výraz popisující prvky seznamu (výraz před svislou čarou). Příklad:

```
[sqn | n <- [1..99], let sqn = n^2, sqn < 10] ⇝* [1,4,9]
```

Kvalifikátory napravo mohou používat hodnot generovaných levějšími (dřívějšími) kvalifikátory:

```
[ (x,y) | x<-[1..3], y<-[1..x] ] ⇝* [(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

Prvky výsledného seznamu nemusejí záviset na všech (na žádných) generovaných hodnotách, pouze na jejich počtu:

```
[ 0 | n <- [1..4] ]                ⇝*   [0,0,0,0]
[ m | m <- [1..3], n <- [m..3] ]   ⇝*   [1,1,1,2,2,3]
```

Definujme funkci `factors`, která, je-li aplikována na přirozené číslo n, vrátí rostoucí seznam všech kladných dělitelů čísla n.

```
factors   ::  Int -> [Int]
factors n  =  [ d | d <- [1..n], n `mod` d == 0 ]
```

Pak můžeme definovat další funkci `primes`, která počítá seznam všech prvočísel nepřevyšujících její argument.

```
primes   ::  Int -> [Int]
primes n  =  [ k | k <- [2..n], factors k == [1,k] ]
```

Funkce `spaces` po aplikaci na číslo n vrátí řetězec n mezer.

```
spaces   ::  Int -> [Char]
spaces n  =  [ ' ' | i<-[1..n] ]
```

Je-li argument funkce `spaces` nulový, je výsledkem prázdný řetězec `""`.

Dalším příkladem použití intenzionální notace je funkce `quicksort`, jejímž parametrem je číselný seznam a výsledkem je tentýž seznam permutovaný tak, aby byl neklesající.

```
quicksort       ::  [Int] -> [Int]
quicksort []     =  []
quicksort (p:s)  =  quicksort [ x | x<-s, x<p ]
                    ++ [p]
                    ++ quicksort [ x | x<-s, x>=p ]
```

Nekonečné seznamy

Líné vyhodnocování (normální pořadí redukcí) nám umožňuje pracovat s funkcemi, které jsou definovány na nedefinovaných argumentech. Například máme-li funkci

```
const x y = x
```

pak výraz `const 1 (1/0)` má definovanou hodnotu (rovnou číslu 1), přestože hodnota podvýrazu `(1/0)` není definována. Podvýraz `(1/0)` se totiž vůbec nevyhodnotí, takže jeho nedefinovanost nevadí.

Nedefinovaná hodnota nemusí být způsobena jen dělením nulou, ale obvykle se považuje za význam cyklícího výpočtu. Například zapíšeme-li definici

```
undefb ::  Bool
undefb  =  undefb
```

je výraz `undefb` nedefinován (zacyklí se), ale výraz `False && undefb` je dobře definován (a má hodnotu `False`), neboť operátor `(&&)` zde svůj druhý operand vůbec nevyhodnocuje.

Užitečným příkladem nedefinovaných podvýrazů jsou nekonečné datové struktury, zejména nekonečné seznamy. Například předpis

```
ones  =  1 : ones
```

definuje nekonečný seznam samých jedniček. Vyhodnocení výrazu `ones` se zacyklí vytvářejíc v každém kroku delší a delší seznam jedniček:

$$\texttt{ones} \rightsquigarrow \texttt{1:ones} \rightsquigarrow \texttt{1:1:ones} \rightsquigarrow \texttt{1:1:1:ones} \rightsquigarrow \cdots$$

Avšak vyhodnocení výrazu `take 2 ones` probíhá díky línému vyhodnocování konstruktoru `(:)` takto

```
    take 2 ones
⇝   1 : take (2-1) ones
⇝   1 : take 1 ones
⇝   1 : 1 : take (1-1) ones
⇝   1 : 1 : take 0 ones
⇝   1 : 1 : []   ≡   [1,1]
```

Toto vyhodnocení interpretujeme jako extrakci dvouprvkového seznamu ze začátku *nekonečného seznamu* složeného ze samých jedniček.

Příklady nekonečných seznamů

Seznam všech druhých mocnin přirozených čísel lze vyjádřit zápisem

```
map square [0..]
```

anebo pomocí intenzionální notace

```
[ x^2 | x <- [0..] ]
```

Zápis `[0..]` zde označuje nekonečný seznam $[0,1,2,\ldots]$. Obecně, pro celé číslo m zápis `[`m`..]` znamená nekonečný seznam $[m,m+1,m+2,\ldots]$. Pro dvě celá čísla m, n zápis `[`m`..`n`]` označuje konečný seznam $[m,m+1,m+2,\ldots,n]$. Je-li $m > n$, pak `[`m`..`n`]` je prázdný seznam.

Tabulka mocnin

```
[  [ x^n | x <- [1..] ]  |  n <- [1..]  ]
```

je nekonečný seznam nekonečných seznamů čísel

```
[
   [ 1,  2,   3,    4,    5,    6, ... ],
   [ 1,  4,   9,   16,   25,   36, ... ],
   [ 1,  8,  27,   64,  125,  216, ... ],
   [ 1, 16,  81,  256,  625, 1296, ... ],
   [ 1, 32, 243, 1024, 3125, 7776, ... ],
                 ⋮
]
```

Seznam

```
[ n | n <- [1..],  sum [ d | d<-[1..n-1], n`mod`d==0 ] == n ]
```

je uspořádaný seznam všech dokonalých čísel, tj. přirozených čísel n, která jsou rovna součtu všech svých kladných dělitelů menších než n.

Seznam Fibonacciho čísel

```
fib  =  1 : 1 : zipWith (+) fib (tail fib)
```

je definován rekursivně pomocí funkce `zipWith`. Korektnost definice lze dokázat indukcí přes pozici prvku v seznamu.

Na podobném principu je definován Pascalův trojúhelník

```
pascal  =  [1] : [ zipWith (+) r (0:r) ++ [1] | r <- pascal ]
```

což je nekonečný seznam konečných seznamů čísel tvaru

```
[
                [ 1 ] ,
              [ 1 , 1 ] ,
            [ 1 , 2 , 1 ] ,
          [ 1 , 3 , 3 , 1 ] ,
        [ 1 , 4 , 6 , 4 , 1 ] ,
      [ 1 , 5 ,10 ,10 , 5 , 1 ] ,
                  ⋮
]
```

Uspořádaný seznam všech prvočísel

```
sieve [2..]
  where  sieve (p:s) = p : sieve [ n | n <- s, n `mod` p /= 0 ]
```

je definován pomocí Eratosthenova síta: funkce `sieve` má argument seznam čísel, z nichž první je vždy prvočíslem, řekněme p. Pro zjištění dalšího prvočísla je nutno vyškrtat všechny násobky čísla p.

Užitečnými funkcemi pro vytváření nekonečných seznamů jsou `repeat`, `cycle` a `iterate`:

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

cycle   ::  [a] -> [a]
cycle t  =  s  where s = t ++ s

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

Funkce `iterate` vytvoří z funkce f a prvku x seznam $[x, f\ x, f(f\ x), f(f(f\ x)), \ldots]$.

Shrnutí

- Intenzionální seznamová notace dovoluje přehledně definovat seznamy pomocí kvalifikátorů – generátorů, filtrů a lokálních definic.
- V líných jazycích můžeme pracovat s nekonečnými seznamy. Konstruktor `(:)` nevyhodnocuje své argumenty, když nemusí. Proto se při výpočtu vyhodnotí z nekonečného seznamu vždy jen tolik, kolik je třeba.

9 Definice datových typů

Datové a typové konstruktory

V kap. 5 jsme viděli, že datové struktury (seznamy, n-tice) se sestavují z konstruktorů (`[]`, `(:)`, `(,)`,...). Konstruktory jsou zvláštní funkce, pro které neexistují žádná pravidla pro zjednodušení. Skládá-li se výraz pouze z konstruktorů, pak ho již nelze zjednodušit – je svou vlastní hodnotou.[23] Například výrazy `reverse "abc"` nebo `6*7` lze zjednodušit na `"cba"` resp. na `42` podle definice funkce `reverse` resp. interně definovaného násobení. Avšak výraz `(True,0,1):[]` již zjednodušit nelze, neboť `True`, `0`, `1`, `(,,)`, `(:)`, `[]` jsou všechno konstruktory:

```
True :: Bool
0    :: Int
1    :: Int
[]   :: [a]
```

jsou nulární konstruktory (konstanty),

```
(:)  :: a -> [a] -> [a]
```

je binární seznamový konstruktor, a

```
(,,) :: a -> b -> c -> (a,b,c)
```

je ternární konstruktor uspořádaných trojic.

Konstruktorům také říkáme podrobněji *datové konstruktory*, pro odlišení od typových konstruktorů. *Typové konstruktory* slouží pro vyjádření typů v typových výrazech. Ve výše uvedeném příkladě jsou `Bool`, `Int` nulárními typovými konstruktory (typovými konstantami) – nezávisí na jiných typech. Symbol `[]` v typovém výrazu `[a]` je unárním typovým konstruktorem vyjadřujícím typ všech seznamů nad daným typem. (V případě `[a]`, kdy parametrem typového konstruktoru `[]` je typová proměnná, je typ `[a]` polymorfní a znamená „typ všech seznamů“.) Symbol `->` je binárním typovým konstruktorem pro vyjádření typu všech funkcí. Symbol `(,,)` v typovém výrazu `(a,b,c)` je ternárním typovým konstruktorem pro vyjádření typu všech uspořádaných trojic – tj. kartézského součinu.[24]

Definice nových typů

Je možno definovat nové typové a datové konstruktory. Zápisem

`data` *Typcons* a_1 ... a_n `=` *Dcons*$_1$ $t_{1,1} \ldots t_{1,r_1}$ `|` ... `|` *Dcons*$_m$ $t_{m,1} \ldots t_{m,r_m}$

definujeme nový n-ární typový konstruktor *Typcons*. Přitom $a_1, \ldots a_n$ jsou typové proměnné, *Dcons*$_1$ ... *Dcons*$_m$ jsou nové datové konstruktory; $t_{i,j}$ jsou typové výrazy,

[23] Výrazy skládající se pouze z konstruktorů a proměnných se nazývají *vzory* a mohou se vyskytovat na místě parametrů na levých stranách definic funkcí.

[24] Místo „mixfixové“ notace `(a,b,c)` lze použít i prefixové `(,,) a b c`.

v nichž se nevyskytují jiné typové proměnné než $a_1, \ldots a_n$. Pro snazší syntaktické odlišení se datové i typové konstruktory zapisují s velkým počátečním písmenem.

Hodnoty typu $Typcons\ a_1\ \ldots\ a_n$ se tvoří pomocí konstruktorů

$$\begin{array}{lll} Dcons_1 & :: & t_{1,1} \to t_{1,2} \to \cdots \to t_{1,r_1} \to Typcons\ a_1\ \ldots\ a_n \\ Dcons_2 & :: & t_{2,1} \to t_{2,2} \to \cdots \to t_{2,r_2} \to Typcons\ a_1\ \ldots\ a_n \\ & \vdots & \\ Dcons_m & :: & t_{m,1} \to t_{m,2} \to \cdots \to t_{m,r_m} \to Typcons\ a_1\ \ldots\ a_n \end{array}$$

Příklady

Předdefinovaný nulární typový konstruktor `Bool` má dva nulární datové konstruktory: `True` a `False`. Jeho definice by tedy vypadala takto:

```
data Bool  =  True  |  False
```

Podobně by bylo možno definovat nulární typový konstruktor `Day` se sedmi nulárními datovými konstruktory:

```
data Day  =  Su  |  Mo  |  Tu  |  We  |  Th  |  Fr  |  Sa
```

a na novém typu `Day` pak definovat třeba funkci

```
weekend    ::  Day -> Bool
weekend Sa  =  True
weekend Su  =  True
weekend _   =  False
```

Nulární typový konstruktor

```
data Colour  =  Red  |  Green  |  Blue  |  Grey Float
```

má tři nulární a jeden unární datový konstruktor. Hodnoty typu `Colour` representují červenou, zelenou či modrou barvu, anebo odstín šedi parametrizovaný desetinným číslem.

Definice datových typů mohou být rekursivní. Například předpis

```
data Nat  =  Zero  |  Succ Nat
```

definuje nulární typový konstruktor `Nat` a nulární datový konstruktor `Zero` a unární datový konstruktor `Succ`. Konstruktor `Succ` má však parametr typu `Nat`. Jeho typ je tedy `Nat` $\to$ `Nat`. Hodnoty typu `Nat` tedy jsou `Zero`, `Succ Zero`, `Succ (Succ Zero)`, `Succ (Succ (Succ Zero))`, ... a představují všechna přirozená čísla.[25]

Podobně vypadá definice předdefinovaného typu všech seznamů nad hodnotami daného typu,

[25] Ponechme stranou skutečnost, že počítání s takto zavedenými čísly by bylo méně efektivní než počítání s hodnotami typu `Int`.

```
data List a  =  Nil  |  Cons a (List a)
```

Ve skutečnosti používáme zvláštní syntax: místo `Nil` se píše `[]`, místo `Cons` se píše `(:)`, a místo `List a` se typový konstruktor zapisuje ve tvaru `[a]`.

Typový konstruktor `Tree` slouží k popisu typu všech binárních stromů, jejichž uzly jsou ohodnoceny hodnotami stejného typu (který je to typ je určeno parametrem typového konstruktoru `Tree`, například `Tree Bool` je typ všech binárních stromů s uzly ohodnocenými logickými hodnotami).

```
data Tree a  =  Empty  |  Node a (Tree a) (Tree a)
```

Pak lze definovat třeba funkci

```
preorder              ::  Tree a -> [a]
preorder Empty         =  []
preorder (Node v l r)  =  v : preorder l ++ preorder r
```

která seřadí hodnoty uzlů do seznamu v pořadí *preorder* (u stromů se rozlišují levé a pravé následníky).

10 Typové třídy

Dosud jsme se setkali s *monomorfními* typy, k jejichž vyjádření stačí pouze typové konstruktory, tj. v jejich zápisu nejsou potřeba typové proměnné.

```
False    ::  Bool
not      ::  Bool -> Bool
"abc"    ::  [Char]
toUpper  ::  Char -> Char
```

Dále jsme používali *polymorfní* typy. V jejich zápisu vystupovaly typové proměnné.

```
length  ::  [a] -> Int
(.)     ::  (b->c) -> (a->b) -> a->c
fst     ::  (a,b) -> a
```

Typové proměnné v zápisu polymorfních typů jsou implicitně universálně kvantifikovány, tedy jako by typy z předchozího příkladu byly

```
length  ::  Pro všechny typy a. [a] -> Int
(.)     ::  Pro všechny typy a, b, c. (b->c) -> (a->b) -> a->c
fst     ::  Pro všechny typy a, b. (a,b) -> a
```

To znamená, že například v typu funkce `fst` mohou být za typové proměnné `a`, `b` dosazeny *libovolné* typy.

Existují však také typy, u nichž je vhodné jejich polymorfismus jistým způsobem omezit. Jaký je třeba typ operátoru rovnosti `(==)` ? Kdybychom rovnosti definovali jako monomorfní funkce, pak bychom museli mít zvláštní operátor pro každý typ:

```
eqB   ::  Bool -> Bool -> Bool
eqI   ::  Int -> Int -> Bool
eqF   ::  Float -> Float -> Bool
eqC   ::  Char -> Char -> Bool
eqFC  ::  (Float,Char) -> (Float,Char) -> Bool
```

To by však bylo nepohodlné a nepřehledné. Jednak typů, jejichž hodnoty chceme testovat na rovnost, je mnoho, jednak bychom pro ně všechny rádi používali stejný operátor, `(==)` . Zdálo by se, že právě zde je vhodné definovat polymorfní typ,

```
(==)  ::  a -> a -> Bool
```

Avšak takový polymorfismus je příliš obecný. Říká, že za typovou proměnnou `a` můžeme dosadit *libovolný* typ. Ale testovat rovnost třeba dvou funkcí typu `Int->Int` neumíme.[26] Rádi bychom vyjádřili, že v typu `a -> a -> Bool` může typová proměnná `a` zastupovat jen *některé* typy – ty, jejichž hodnoty testovat na rovnost umíme.

Řešením je zavedení tzv. *typových tříd*. Zavedeme typovou třídu (množinu typů) `Eq`, do níž budou patřit všechny typy, na jejichž hodnoty lze operaci `(==)` aplikovat. Typ rovnosti se pak zapíše

[26] Rovnost dvou funkcí není obecně algoritmicky rozhodnutelná.

```
(==)  ::  Eq a  =>  a -> a -> Bool
```

a stejná situace je s nerovností:

```
(/=)  ::  Eq a  =>  a -> a -> Bool
```

Tyto zápisy čteme „typ rovnosti (nerovnosti) je `a` → `a` → `Bool` pro všechny typy `a` ze třídy `Eq`“. Za typovou proměnnou `a` můžeme tedy dosadit jakýkoliv typ patřící do třídy `Eq`, ale žádný jiný. Typům patřícím do nějaké třídy se říká *instance* této třídy. Typy jako `Bool`, `Int`, `Char`, `(Int,Char)` apod. jsou instancemi třídy `Eq`, typy jako `Int->Int` instancemi třídy `Eq` nejsou.

Funkce `elem` zjišťující příslušnost prvku do seznamu je definována

```
elem        ::  Eq a  =>  a -> [a] -> Bool
elem _ []    =  False
elem x (y:t) =  x==y || elem x t
```

V jejím typu je vyjádřeno, že typová proměnná `a` může nabývat pouze typů ze třídy `Eq` (jinak bychom ve druhé klausuli definice nemohli testovat rovnost).

Části „`Eq a =>`“ v zápisu typu říkáme *typový kontext*. Ten se může skládat i z více typových omezení, která jsou pak uzavřena v kulatých závorkách a oddělena čárkami:

```
paireq  ::  (Eq a, Eq b) => (a,b) -> (a,b) -> Bool
paireq (x,y) (u,v)  =  x==u && y==v
```

Prázdný typový kontext `() =>` se v zápisech typů obvykle vynechává: místo `id :: ()=>a->a` se píše `id :: a->a`. Prázdný typový kontext odpovídá „neomezenému“ parametrickému polymorfismu.

Instance typových tříd

Tím, co danou typovou třídu charakterizuje, jsou právě operace, které můžeme s hodnotami instancí této třídy provádět. Například u třídy `Eq` to jsou operace `(==)` a `(/=)`.

Definice typové třídy `Eq` v Haskellu vypadá takto:

```
class Eq a
      where (==), (/=) ::  a -> a -> Bool
```

Tato definice říká, že `Eq` je třída, do níž patří (jejíž instancí je) každý takový typ `a`, pro který jsou definovány operace `(==), (/=) :: a -> a -> Bool`.

Když máme definovánu typovou třídu, je třeba ještě určit, které instance tato třída má (tj. které typy do ní patří). K tomu slouží deklarace instance. Například ve standardní knihovně Haskellu je deklarace

```
instance Eq Char  where  (==)    =  primEqChar
                         x /= y  =  not (x == y)
```

Tento zápis říká, že typ `Char` je instancí třídy `Eq`, přičemž pokud je operace `(==)` aplikována na argumenty typu `Char`, je tato operace definována pomocí funkce `primEqChar`. Funkce `primEqChar` je interní (zabudovaná) a má monomorfní typ `Char->Char->Bool`, na rozdíl od polymofního typu operace `(==)`.

Definici operace v deklaraci typové instance se někdy říká *implementace metody* deklarované v typové třídě.[27] Stejně pojmenovaná operace (metoda) může být v jiném typu (jiné instanci) téže typové třídy definována (implementována) jinak:

```
instance Eq Int  where  (==)   = primEqInt
                        x /= y = not (x == y)
instance Eq Float  where  (==)   = primEqFloat
                          x /= y = not (x == y)
```

V instancích `Char`, `Int`, `Float` byla operace `(/=)` definována vždy jako negace operace `(==)`. Tato situce, kdy instance nějaké typové třídy se liší pouze některými operacemi, a ostatní operace jsou od nich odvozeny, je častá. Proto již v definici typové třídy lze uvést definice *implicitních operací*[28]. V případě třídy `Eq` je implicitní operací operace `(/=)`, která je definovaná pomocí operace `(==)`.

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

Může být typ `(a,b)` instancí třídy `Eq`? To záleží na tom, zda jsou jejími instancemi i typy a, b. Umíme-li otestovat rovnost levých i pravých složek dvou uspořádaných dvojic, pak rovnost celých dvojic získáme jako konjunkci rovností odpovídajících si složek. Jestliže však u některé složky testovat rovnost neumíme, nemůžeme rozhodnout ani o rovnosti celých dvojic. Takže například typ `(Char,Bool)` je instancí třídy `Eq`, ale typy `(Int->Int,Bool)` nebo `(Char,Float->Int)` nikoliv.

Tuto závislost lze vyjádřit deklarací instance

```
instance (Eq a, Eq b) => Eq (a,b)
         where (x,y) == (u,v)  =  x==u && y==v
```

Deklarace říká, že typ všech uspořádaných dvojic `(a,b)` je instancí třídy `Eq`, pokud jsou jejími instancemi typy `a`, `b`. Jinak řečeno, instancí třídy `Eq` je polymorfní typ `(a,b)`, jehož polymorfismus je ovšem omezen typovým kontextem `(Eq a, Eq b)`.

Podobně seznamový typ `[a]` je instancí třídy `Eq`, pokud je její instancí i typ `a`.[29]

```
instance  Eq a  =>  Eq [a]
          where  [] == []        =  True
                 (_:_) == []     =  False
                 [] == (_:_)     =  False
                 (x:s) == (y:t)  =  x == y && s == t
```

[27] Tato terminologie pochází z objektově orientovaného programování.

[28] angl. *default methods*

[29] Tato vlastnost seznamových typů v Haskellu je poněkud diskutabilní, uvážíme-li, že nám dovoluje testovat na rovnost i nekonečné seznamy.

Podtřídy

V kapitole 8 jsme se setkali s definicí funkce `quicksort`, která seřadila prvky celočíselného seznamu podle velikosti. Nemusíme se však omezovat jen na celá čísla, protože řadit podle velikosti lze hodnoty všech typů, pro něž jsou definovány operace `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, `min`. Takové typy patří do typové třídy `Ord`. Ale pro tyto typy je definováno nejen zmíněných šest operací, ale i operace `(==)` a `(/=)` z typové třídy `Eq`. Říkáme, že třída `Ord` je *podtřídou* třídy `Eq`. Definice třídy `Ord` může vypadat takto:[30]

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

Tato definice říká, že třída `Ord` je podtřídou třídy `Eq`, to znamená, že na hodnoty typů třídy `Ord` můžeme aplikovat nejen operace `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, `min`, ale rovněž operace `(==)`, `(/=)`. Někdy se též říká, že podtřída *dědí* metody své nadtřídy, např. `Ord` zdědila metody `(==)` a `(/=)` ze třídy `Eq`.

Typová třída může být podtřídou i více než jedné třídy. Například ve standardním Haskellu je třída `Real`, která je podtřídou tříd `Num` a `Ord`:

```
class (Num a, Ord a) => Real a
      where ...
```

Třída `Num` je třída číselných typů – jejími standardními instancemi jsou typy `Int`, `Integer`, `Float`, ale další její instancí může být třeba typ komplexních čísel. Instancemi třídy `Real` jsou opět typy `Int`, `Integer`, `Float`, ale typ `Complex` její instancí být nemůže. Třída `Real` je totiž také podtřídou třídy `Ord`, takže na hodnotách jejích typů musí být definováno uspořádání.

Přetížení

Polymorfismus rovnosti není parametrický. Parametrický polymorfismus je totiž takový, kdy hodnota je na všech typech definována jediným způsobem. Například funkce `length` počítá prvky seznamu vždy stejným způsobem bez ohledu na to, jaký mají typ. Jinými příklady parametrického polymorfismu jsou funkce `fst`, `snd`, `head`, `tail`, `foldr`, `zip`, `flip`, atd. Naproti tomu rovnost celých čísel musíme testovat jinak definovanou funkcí `(==)` než rovnost seznamů (rovnost čísel se dá testovat přímo, rovnost seznamů je definována rekursivně). Tomuto druhu polymorfismu, kdy na různých typech se funkce může chovat různě, se říká *přetížení*. Operace `(==)`, `(/=)` jsou tedy přetíženy.

[30] Definice v `Prelude.hs` je o něco podrobnější

Také aritmetické operace `(+)`, `(-)`, `(*)`, `(/)`, `(^)` jsou přetíženy. Tyto operace jsou na hodnotách typu `Int` definovány jinak než na hodnotách typu `Float` – operace se na nich provádějí pomocí různých strojových instrukcí. A na hodnotách typu `Integer` jsou aritmetické operace zase zavedeny pomocí knihovních procedur pro „dlouhou aritmetiku“.

Podobně jako existuje typová třída `Eq`, na jejíchž instancích jsou definovány operace `(==)`, `(/=)`, existuje typová třída `Num`, na jejíchž instancích jsou definovány například operace `(+)`, `(-)`, `(*)`. Instancemi třídy `Num` jsou typy `Int`, `Float`.

Shrnutí

- Typové třídy jsou množiny typů, nad jejichž hodnotami lze provádět určité operace. Těmito operacemi je typová třída definována.
- Prvky typové třídy se nazývají instance.
- Polymorfismus typu může být zúžen typovým kontextem. Typový kontext omezuje některé typové proměnné tím, že stanoví, kterých typových tříd musí být instancemi.
- Typy operací (metod) jsou definovány v typové třídě, jejich hodnoty (implementace) až v deklaracích instancí třídy. Definice oprací, které mají být ve všech instancích stejné, mohou být uvedeny už v definici typové třídy jako tzv. implicitní metody.
- Typové třídy lze definovat jako podtřídy tříd jiných. Podtřídy pak dědí operace z nadtříd.
- Typová třída může být podtřídou více tříd současně.
- Pomocí typových tříd a jejich instancí lze formálně popsat přetížení operací.

11 Vstup a výstup

Monadické operátory vstupu a výstupu

V Haskellu existuje zabudovaný typový konstruktor `IO`. Je-li a nějaký typ, pak `IO` a je typ tzv. *akcí*, které mají tzv. *vnitřní výsledek* typu a. Akce vyjadřuje nějaký děj při běhu programu, nejčastěji to je vstup nebo výstup. Akce, při nichž se získá nějaká hodnota, typicky to jsou vstupní akce, mají tuto hodnotu jako svůj vnitřní výsledek. Akce, které svým proběhnutím žádnou hodnotu neposkytnou (třeba výstupní akce), mají za vnitřní výsledek triviální hodnotu – uspořádanou nultici `()`.

Například `getChar` je akce typu `IO Char`. Tato akce, je-li vyhodnocena, způsobí přečtení znaku ze standardního vstupu. Přečtený znak je vnitřním výsledkem akce. Podobně `getLine` je akce typu `IO String`. Je-li vyhodnocena (provedena), přečte ze standardního vstupu řetězec znaků zakončený oddělovačem řádku. Přečtený řetězec je opět vnitřním výsledkem akce.

Funkce `putChar :: Char->IO ()` slouží pro výpis jednoho znaku na výstup. Je-li aplikována na znak, vrátí akci, která tento výstup zařídí. Například `putChar '*'` je akce typu `IO ()`, tedy s nezajímavým vnitřním výsledkem, ale s účinkem na prostředí, v němž program běží: při jejím provedení se na standardní výstup zapíše hvězdička.

Podobně funkce `putStr :: String->IO ()` slouží pro výpis řetězce znaků na standardní výstup. Její aplikace na řetězec je akcí typu `IO ()`, která řetězec zapíše na výstup.

Představme si, že chceme ze vstupu přečíst písmeno, převést ho na velké a zapsat na výstup. Přečtení zařídí akce `getChar` a přečtený znak bude jejím vnitřním výsledkem. Jak se však dostat k tomuto vnitřnímu výsledku? Nelze to přímo. Kdyby existovala funkce, řekněme `upIO::IO a->a`, která by odhalovala vnitřní výsledky akcí, porušila by se tzv. *referenční transparentnost* jazyka: výraz `upIO getChar` by měl pokaždé jinou hodnotu, podle znaku na vstupu.

Existuje však operátor

```
(>>=)  ::  IO a  ->  (a -> IO b)  ->  IO b
```

který aplikuje funkci, jež je jeho druhým argumentem, na vnitřní výsledek akce, jež je jeho prvním argumentem. Touto aplikací vznikne nová akce.

Výše zmíněné přečtení, převod a zápis písmene lze provést akcí[31]

```
getChar  >>=  putChar . toUpper
```

Operátor `(>>=)` řadí akci a funkci vracející akci za sebe a zajišťuje, aby akce vlevo byla provedena vždy dříve. Proto se mu říká *sekvencializační operátor*. Sekvencializační operátor `(>>=)` je jediným operátorem, který může využít vnitřního výsledku předcházející akce, a to jen jediným způsobem: předat ho jako argument funkci a vytvořit další akci. Z toho plyne, že posloupnost akcí závislých na vnitřních výsledcích předcházejících akcí se nemůže větvit. Říkáme, že tok akcí je *jednovláknový*. Díky tomu je zachována referenční transparentnost.

[31] Operátor `(>>=)` má nízkou prioritu, nižší než `(.)`.

Druhým argumentem operátoru `(>>=)` není akce, ale funkce vracející akci, aby bylo možno využít vnitřní výsledek první akce. V případě některých akcí, zejména výstupních, nás tento vnitřní výsledek nezajímá (je jím uspořádané nultice `()`) a chceme ho ignorovat. Místo sekvencializačního operátoru `(>>=)` lze použít jednoduššího sekvencializačního operátoru `(>>)`, jehož oba argumenty jsou akce. Operátor `(>>)` ignoruje vnitřní výsledek první akce (jímž ostatně obvykle je jen hodnota `()`) a po jejím provedení provede akci druhou.

Definice operátoru `(>>)` je

```
(>>)  ::  IO a  ->  IO b  ->  IO b
act1 >> act2   =   act1  >>=  \ _ -> act2
```

Pak například

```
putStr "aaa"  >>  putStr "bbb"
```

je akce, která na výstup vypíše řetězec `"aaabbb"` (a její vnitřní výsledek je `()`).

Tento zápis je jednodušš alternativou k zápisům

```
putStr "aaa"  >>=  \ x-> putStr "bbb"
```

nebo

```
putStr "aaa"  >>=  const (putStr "bbb")
```

a je s nimi ekvivalentní.

Dalším užitečným operátorem je `return :: a->IO a`, který vytváří prázdnou akci, tj. akci, která nemá žádný účinek na vstupy, výstupy ani prostředí operačního systému (tj. nic nedělá), ale má nějaký vnitřní výsledek. Například `return ()` je prázdná akce typu `IO ()` s vnitřním výsledkem `()`, `return 0` je prázdná akce typu `IO Int` s vnitřním výsledkem 0 atd. Máme-li třeba seznam akcí, z nichž každá je typu `IO ()`, lze z nich vytvořit jednu akci pomocí funkce `sequence_` definované

```
sequence_  ::  [IO a] -> IO ()
sequence_  =  foldr (>>) (return ())
```

Funkcionální programy

Víme, že každý funkcionální program je výraz. Například výraz `2+3` je programem typu `Int`. Vyhodnocování takového „programu“ nemá žádné účinky na okolí. Od programů však obvykle požadujeme vstup a výstup.

Akce nazvaná `main`, jejíž typ je `IO ()`, se nazývá *proveditelný program*. Proveditelný program lze například zkompilovat a pak spustit. Při tomto spuštění (provedení programu) se projeví vedlejší účinky akcí.

Příklad 4 Proveditelný program, který při spuštění načte řádek, převede ho na velká písmena a výsledek vypíše na výstup:

```
main  ::  IO ()
main  =  putStr "Vstupní řádek: "                          >>
         getLine                                           >>=
         \  vstup -> putStr "Totéž velkými písmeny: "       >>
                    putStr (map toUpper vstup)
```

Složené akce (vzniklé řazením menších akcí) lze zapsat též pomocí konstrukce `do`. Uvedený příklad pak vypadá takto:

```
main  ::  IO ()
main  =  do putStr "Vstupní řádek: "
            vstup <- getLine
            putStr "Totéž velkými písmeny: "
            putStr (map toUpper vstup)
```

Obecně, akci `m >> dalsiakce` zapíšeme

```
do m
   dalsiakce
```

Akci `m >>= \ x -> dalsiakce` zapíšeme

```
do v <- m
   dalsiakce
```

Podrobnější popis akcí v Haskellu a přehled vstupních a výstupních akcí a funkcí lze najít v [4].

Shrnutí

- Vstup a výstup lze ve funkcionálních programech provádět pomocí akcí – hodnot zvláštního typu `IO a`.
- Akce lze řadit za sebe pomocí sekvencializačního operátoru `(>>)`.
- Akce, z nichž chceme použít vnitřní výsledek, lze řadit s funkcemi pomocí sekvencializačního operátoru `(>>=)`.
- `return` v je prázdná akce s vnitřním výsledkem v.
- Proveditelný program je hodnota `main` typu `IO ()`.
- Složené akce lze vyjádřit konstrukcí `do`.

Reference

[1] Richard Bird: *Introduction to Functional Programming using Haskell*, Prentice Hall Europe, 1998

[2] Paul Hudak: *The Haskell School of Expression*, Cambridge University Press, 2000

[3] Simon Thompson: *Haskell—The Craft of Functional Programming*, Addison Wesley, 1996

[4] Simon Peyton Jones et al.: *Report on the Programing Language Haskell 98*, available at `http://haskell.org/definition/`