Priority operátorů, prefix/infix, if, definice podle vzoru, lokální definice

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

A Cup of Tea

Nan-in, a Japanese master during the Meiji era (1868-1912), received a university professor who came to inquire about Zen. Nan-in served tea. He poured his visitor's cup full, and then kept on pouring.
The professor watched the overflow until he no longer could restrain himself. "It is overfull. No more will go in!"
"Like this cup," Nan-in said, "you are full of your own opinions and speculations. How can I show you Zen unless you first empty your cup?"

The hardest thing about learning functional programming is forgetting what you think you "know". It might be true, but not in this context. Everything is different, so you need to start from the very beginning.
If you know C++, and you want to learn Java, you compare both languages all the time, and this works fine. It's the same with natural languages: If you know English and want to learn Spanish, it's okay to compare both languages all the time, as they follow the same indo-germanic paradigm. But if you want to learn something fundamental different, like Japanese, or Haskell, you have to "unlearn" first. You won't make real progress until you stop comparing.

Zenový kōan a komentář o výuce funkcionálního programování
přebrán z diskusního fóra *Programmers Stack Exchange* od uživatele *Landei*.

Haskell v praxi

Haskell:

- kompilátor vs. interpret
- Hugs, GHC, Haskell Platform
- GHC a Haskell Platform: `ghc`, `ghci`, `runghc`

Spouštení:

- všechny nástroje jsou spustitelné v shellu
- spuštení shellu např.: `Alt-F2`, `gnome-terminal`
- prompt shellu: `xlogin@nymfe12:/home/xlogin$`
- prompt interpretu: `Prelude>` nebo `Main>`

Práce s interpretem

Do interpretu lze zadávat příkazy a výrazy.
Příkazy:

- `:h[elp]` – nápověda
- `:t[ype] expr` – typ výrazu
- `:i[nfo] ident` – informace o objektech jazyka
- `:l[oad] file.hs` – načtení souboru s Haskellovým kódem
- `:r[eload]` – znovunačtení posledního souboru
- `:q[uit]` – ukončení práce s interpretem
- `:m[odule] Modul` – načtení modulu

Výrazy:

- vše ostatní...

Nekončící výpočty lze přerušit pomocí `Ctrl-C`.

Úvod k operátorům

Příklad 1.1.1: S použitím interpretru jazyka Haskell porovnejte vyhodnocení následujících dvojic výrazů a rozdíl vysvětlete.

a) `5 + 9 * 3` versus `(5 + 9) * 3`

b) `2 ^ 2 ^ 2 == (2 ^ 2) ^ 2` versus
`3 ^ 3 ^ 3 == (3 ^ 3) ^ 3`

c) `3 + 3 + 3` versus `3 == 3 == 3`

d) `(3 == 3) == 3` versus `(4 == 4) == (4 == 4)`

Vlastnosti operátorů

Příklad 1.1.2: S využitím interního příkazu `:info` interpretru `ghci` zjistěte prioritu a směr vyhodnocování následujících operací:

```
., !!, ^, *, /, `div`, `mod`, +, -, :, ++, ==, /=,
>, <, >=, <=, &&, ||
```

Vlastnosti operátorů

<table>
<tr><th>operátor</th><th>priorita</th><th>asociativita</th></tr>
<tr><td>.</td><td>9</td><td>←</td></tr>
<tr><td>!!</td><td>9</td><td>→</td></tr>
<tr><td>^</td><td>8</td><td>←</td></tr>
<tr><td>*, /, `div`, `mod`</td><td>7</td><td>→</td></tr>
<tr><td>+, -</td><td>6</td><td>→</td></tr>
<tr><td>:, ++</td><td>5</td><td>←</td></tr>
<tr><td>==, /=, <, <=, >, >=</td><td>4</td><td>–</td></tr>
<tr><td>&&</td><td>3</td><td>←</td></tr>
<tr><td>||</td><td>2</td><td>←</td></tr>
<tr><td>>>=, >></td><td>1</td><td>→</td></tr>
<tr><td>$</td><td>0</td><td>←</td></tr>
</table>

Pokud to nemá operátor/funkce explicitně definované, jeho priorita je 9 a je asociativní zleva (→).

Podmíněný výraz

Syntax: `if bool_expr then expr1 else expr2`
Výrazy *expr1* a *expr2* musí být stejného typu.

Příklad 1.1.3: Vysvětlete, co je chybné na následujících podmíněných výrazech, a výrazy vhodným způsobem upravte.

a) `if 5 - 4 then False else True`

b) `if 0 < 3 && odd 6 then 1 else "chyba"`

c) `(if even 8 then (&&)) (0 > 7) True`

Prefix, infix, postfix

Binární funkce se dají zapisovat třemi různými způsoby:

1. **prefixový zápis:** `(+) 3 4`, `mod 7 5`
   - operátor/funkce před argumenty
   - je nativně podporovaný v Haskellu (má vyšší prioritu než infix)
   - nealfanumerické funkce musí být v závorkách
2. **infixový zápis:** `3 + 4`, ``7 `mod` 5``
   - operátor/funkce mezi argumenty
   - je nativně podporovaný v Haskellu (má nižší prioritu než prefix)
   - alfanumerické funkce musí být ve zpětných apostrofech
3. **postfixový zápis:** `3 4 (+)`, `7 5 mod`
   - operátor/funkce za argumenty
   - není podporovaný v Haskellu

Prefixový a infixový zápis

Příklad 1.1.4: Přepište infixové zápisy výrazů do syntakticky správných prefixově zapsaných výrazů a naopak:
a) `4 ^ (7 `mod` 5)`
b) `max 3 ((+) 2 3)`

Syntaktická struktura aneb závorkování výrazů

Příklad 1.1.5: Doplňte všechny implicitní závorky do následujících výrazů:

a) `2 ^ mod 9 5`

b) `f . (.) g h . id`

c) `2 + div m 18 * m `mod` 7 == m ^ 2 ^ n - m + 11 && m * n < 20`

d) `flip (.) snd . id const`

e) `f 1 2 g + (+) 3 `const` g f 10`

f) `replicate 8 x ++ filter even (enumFromTo 1 (3 + 9 `mod` x))`

Syntaktická struktura aneb závorkování výrazů

Příklad 1.1.6: Zjistěte (bez použití interpretru), na co se vyhodnotí následující výraz. Poté jej přepište do prefixového tvaru a pomocí interpretru ověřte, že se jeho hodnota nezměnila.

```
5 + 7 * 5 `mod` 3 `div` 2 == 3 * 2 - 1
```

Definice funkce podle vzoru

Definice funkce podle vzoru obsahuje:

- název funkce (`tell`),
- hodnoty argumentů (`1`, `2`), formální parametry (`x`), anonymní parametry (`_`),
- rovnítko oddělující pravou a levou stranu definice (`=`),
- pravou stranu/tělo funkce.

```
tell 1 = "one"
tell 2 = "two"
tell x = if even x then "(even)" else "(odd)"
tell _ = "never happens"
```

- Všechny řádky definující funkci musí být spolu.
- Není možné použít stejný parametr na levé straně víckrát.

Funkce logické konjunkce

Příklad 1.2.1: Definujte funkci `logicalAnd`, která se chová stejně jako funkce logické konjunkce, tak, abyste v definici

a) využili podmíněný výraz.

b) nepoužili podmíněný výraz.

Rekurze

Příklad 1.2.3: Upravte následující kód tak, aby funkce pro záporná čísla necyklila, ale skončila s chybovou hláškou. Použijte k tomu funkci `error :: String -> a`.

```
power :: Double -> Int -> Double
_ `power` 0 = 1
z `power` n = z * (z `power` (n-1))
```

Příklad 1.2.4: Definujte v Haskellu funkci `dfct` (v kombinatorice někdy značenou !!), kde

$$\begin{aligned} 0!! &= 1, \\ (2n)!! &= 2 \cdot 4 \cdots (2n), \\ (2n+1)!! &= 1 \cdot 3 \cdots (2n+1) \end{aligned}$$

Rekurze

Příklad 1.2.11: Napište funkci, která o zadaném přirozeném čísle rozhodne, jestli je mocninou dvojky.

Lokální definice

Lokální definice jsou pojmenované výrazy s lokální platností, jejich primárním účelem je zpřehlednit kód (a nezavádět nová globální jména). Rozeznáváme 2 druhy:

1. s definicí před výrazem (**let-in**)
   Syntax: `let var = subexpr in expr`
2. s definicí za výrazem (**where**)
   Syntax: `expr where var = subexpr`

- Je možné definovat i více výrazů v jedné definici:

```
let a = 25 + b
    b = 5
    in (a + b) * b - 1
```

- Na zarovnání záleží!
- V lokálních definicích je možné používat vzory.

Lokální definice

Příklad 1.3.1: Výraz

```
((3+4)/2) * ((3+4)/2 - 3) * ((3+4)/2 - 4)
```

a) upravte s využitím syntaktické konstrukce pro lokální definici (`let ... in`) tak, aby se v něm neopakovaly stejné složené podvýrazy;

b) upravte stejně, ovšem s využitím globálních definic (uložených v externím souboru).

Datové typy, seznamy a částečná aplikace

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Základní datové typy

Numerické

- celočíselné: `Int` (na platformě závislá velikost) a `Integer` (neomezený)
- s pohyblivou desetinnou čárkou: `Float`, `Double`

Logické hodnoty

- typ `Bool`
- `True` nebo `False`

Funkční typ

- `Parameter -> ... -> Parameter -> ReturnValue`

Nultice

- typ `()`
- jediná hodnota je `()`

Žádné implicitní konverze typů (nelze `Int` namísto `Integer`, ...)

Složené datové typy, n-tice, polymorfismus

Komplikovanější datové typy lze získat skládáním jednodušších:

- n-tice: `(Int, Bool)`, `(Double, Int, Int)`
- seznamy: `[Integer]`, `[Char]`
- vlastní datové typy

Často nezávislé na typu vnitřních hodnot (polymorfismus):

- polymorfní datové typy: `(a, b, c)` (trojice je definovaná pro libovolné typy `a`, `b`, `c`)
- polymorfní funkce: `fst :: (a, b) -> a`
- lze i kombinovat: `(Int, [(a, a)], Bool)`

Datové typy

Příklad 2.1.2: Nalezněte příklady hodnot následujících typů:

a) `Bool`
b) `Integer`
c) `Double`
d) `False`
e) `()`
f) `(Int, Integer)`
g) `(Integer, Double, Bool)`
h) `((), (), ())`

Datové typy

Příklad 2.1.3: Určete typy následujících výrazů, zkontrolujte si řešení pomocí interpretru.

a) `True`

b) `"True"`

c) `not True`

d) `True || False`

e) `True && "1"`

f) `f 1`, kde funkce `f` je definovaná jako

```
f :: Integer -> Integer
f x = x * x + 2
```

g) `f 3.14`, kde `f` je definovaná stejně jako v části f

h) `g 3.14`, kde `g` je definovaná jako

```
g :: Double -> Double
g x = x * x + 2
```

Datové konstruktory

- začínají velkým písmenem nebo dvojtečkou
- vytvářejí hodnoty (daného typu)
- chovají se jako konstanty (nulární konstruktory) nebo funkce
- například: `True`, `False`, seznamové konstruktory `[]` a `(:)`
- lze je použít v definici podle vzoru (obecné výrazy, jako třeba `1 + n`, použít nelze)
- číselné literály se chovají jako konstruktory číselných typů

Částečná aplikace

Příklad 2.2.1: Co vyjadřuje výraz `min 6`? Napište ekvivalentní výraz pomocí `if`.

Částečná aplikace

Částečná aplikace
Umožňuje nám aplikovat funkci na méně parametrů, než očekává, a získat tak funkci, která bude očekávat zbylé parametry.

- `(+) 4` je funkce, která očekává 1 parametr a přičte k němu 4
- `(-) 3` očekává 1 parametr a odečte ho od trojky

Operátorové sekce
Alternativní metoda zápisu částečné aplikace pro binární funkce.

- levá `(2 /)` vydělí 2 svým argumentem
- pravá `(/ 2)` vydělí svůj argument 2
- možno i pro funkce s alfanumerickým názvem: ``(`mod` 2)``

Závorky jsou povinné.

Implicitní závorky

Příklad 2.2.2: Které z následujících výrazů jsou ekvivalentní?

a) `f 1 g 2` $\equiv$ `f 1 (g 2)`
b) `(f 1 g) 2` $\equiv$ `(f 1) g 2`
c) `(+ 2) 3` $\equiv$ `2 + 3`
d) `(+) 2 3` $\equiv$ `(+ 2) 3`
e) `81 * f 2` $\equiv$ `(*) 81 f 2`
f) `fact n` $\equiv$ `n * fact n - 1` (uvažujíce klasickou rekurzivní definici funkce `fact`)
g) `sin (1.43)` $\equiv$ `sin 1.43`
h) `sin 1.43` $\equiv$ `sin 1 . 43`
i) `8 - 7 * 4` $\equiv$ `(-) 8 (* 7 4)`

Seznamy

- homogenní kolekce
- seznam může obsahovat libovolný počet prvků daného typu
- typ seznamu zapisujeme jako `[ElementType]`, je to tedy parametrizovaný typ
- má sekvenční přístup k prvkům – k prvkům lze přistupovat pouze od začátku seznamu

Příklady:

- `[True, False]`
- `[1, 2, 42, 128]`
- `1:2:3:[]`

Datové konstruktory:

- `[] :: [a]` prázdný seznam (nad libovolným typem)
- `(:) :: a -> [a] -> [a]` připojí prvek na začátek seznamu

Příklad 2.3.1: Rozhodněte, které z následujících seznamů jsou správně utvořené. U nesprávných rozhodněte proč, u správně utvořených určete typ. Konzultujte své řešení s interpretrem.

a) `[1, 2, 3]`
b) `(1:2):3:[]`
c) `1:2:3:[]`
d) `1:(2:(3:[]))`
e) `[1, 'a', 2]`
f) `[ [], [1, 2], 1:[] ]`
g) `[ 1, [1, 2], 1:[] ]`
h) `[]:[]`

Seznamy

Příklad 2.3.2: Určete typy seznamů:

a) `["a", "b", "c"]`

b) `['a', 'b', 'c']`

c) `"abc"`

d) `[(True, ()), (False, ())]`

e) `[(++) "abc" "def", "X" ++ "Y" ++ "Z"]`

f) `[(&&), (||)]`

g) `[]`

h) `[[]]`

i) `[[], [""]]`

Vzory seznamů

Příklad 2.3.3: Pro následující vzory a seznamy určete, které vzory mohou reprezentovat které seznamy. Stanovte, jak se navážou proměnné ze vzoru.

vzory:

`[], x, [x], [x,y], (x:s), (x:y:s), [x:s], (x:y):s`

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

Funkce na seznamech

Příklad 2.3.4: Definujte funkce `myHead :: [a] -> a` (která vrátí první prvek seznamu) a `myTail :: [a] -> [a]` (která vrátí seznam bez prvního prvku). Nepoužívejte knihovní funkce `head`, `tail`.

Příklad 2.3.8: Definujte funkci `len :: [a] -> Integer`, která spočítá délku seznamu. Nesmíte použít funkci `length`.

Příklad 2.3.11: Definujte rekurzivní funkci `multiplyN :: Integer -> [Integer] -> [Integer]`, která vrátí seznam, v němž je každý prvek v druhém seznamovém parametru vynásoben číslem, které je prvním parametrem funkce.

Funkce na seznamech: `filter`

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

Vybere ze seznamu ty prvky, které splňují danou podmínku.

`filter odd [1,2,3,4,5,6]` $\leadsto^*$ `[1,3,5]`

Funkce na seznamech: `filter`

Příklad 2.3.15: Definujte funkci
`evens :: [Integer] -> [Integer]`, která ze seznamu vybere sudá čísla. Použijte funkci `filter`.

Funkce na seznamech: `map`

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

Funkce `map` aplikuje zadanou funkci na každý prvek seznamu zvlášť a vrátí výsledný seznam po aplikaci.

`map (+ 2) [1,2,3]` $\leadsto^*$ `[ (+ 2) 1, (+ 2) 2, (+ 2) 3 ]`
$\leadsto^*$ `[3,4,5]`

Funkce na seznamech: `map`

Příklad 2.3.16: S využitím funkce `map` a knihovní funkce `toUpper :: Char -> Char` z modulu `Data.Char` (tj. je třeba použít `import Data.Char`, na začátku souboru, nebo `:m + Data.Char` v interpretru) definujte novou funkci `toUpperStr`, která převádí řetězec písmen na řetězec velkých písmen, tj. `toUpperStr "bob"` $\leadsto^*$ `"BOB"`.

Příklad 2.3.17: Definujte funkci
`multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2. Nepoužívejte rekurzi explicitně.

Příklad: `multiplyEven [2,3,4]` $\leadsto^*$ `[4,8]`,
`multiplyEven [6,6,3]` $\leadsto^*$ `[12,12]`.

Funkce na seznamech: `map`, `filter`

Příklad 2.3.22: Napište funkci `vowels`, která dostane seznam řetězců a vrátí seznam řetězců takových, že v každém řetězci ponechá jenom samohlásky (ale zachová jejich pořadí). Například `vowels ["ABC","DEF"]` se vyhodnotí na `["A","E"]`.

Skládání funkcí, typování II, další funkce na seznamech

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Skládání funkcí

Jednou ze základních operací s funkcemi je jejich skládání.

- v matematice zapisujeme $f \cdot g$, v Haskellu `f . g`, přičemž `f` a `g` bereme jako unární funkce
- aplikace složené funkce `(f . g) x` je ekvivalentní `f (g x)`
  - závorky okolo `(f . g)` jsou nutné jinak by se vyhodnotilo jako `f . (g x)`
- zápisy `h = f . g` nebo také `h = (.) f g` jsou dle definice tečky ekvivalentní zápisu `h x = f (g x)`
- operátor tečka má nejvyšší prioritu a je asociativní zprava
- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

Skládání funkcí v praxi

Příklad 3.1.1: Vyhodnoťte následující výrazy:

a) `((== 42) . (2 +)) 40`

b) `((> 2) . (* 3) . ((-) 4)) 5`

c) `filter ((>= 2) . fst) [(1,"a"), (2,"b"), (3,"c")]`

Skládání funkcí v praxi

Příklad 3.1.2: Uvažme funkci
`negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních funkcí typu `a -> Bool` (tzv. predikátů).

a) Definujte funkci `negp`.
b) Definujte funkci `negp` jako unární funkci (s použítím pouze jedného formálního parametru).
c) Definujte funkci `negp` bez použití formálních parametrů.

Příklad 3.1.3: Určete všechny implicitní závorky v následujících výrazech:

a) `f.g x`
b) `f (.) g (h x) . (.) f g x`

Typování funkcí: teorie

Typovat funkce a výrazy lze v zásadě dvěma způsoby:

- **Intuitivní typování:** Výraz posoudíme z hlediska toho, co dělá, a na základě toho určíme typ. Vhodné pro nepříliš složité výrazy.
- **Algoritmické typování:** Exaktní určení typu typovacím algoritmem. Je zdlouhavější, ale funguje pro všechny výrazy.

Výraz lze před otypováním zjednodušit nebo vyhodnotit. Ale pozor! Polymorfizmus může při intuitivním vyhodnocení kvůli ztrátě informace způsobit změnu typu:

- `head [id, not]`
- `min 2 (3.1 :: Float)`

Typování funkcí: teorie

Typovací algoritmus

- každý výskyt funkce, konstanty, formálního argumentu otypujeme (nezávislé typové proměnné)
- určíme rovnosti vyplývající z aplikací funkcí na argumenty (aplikace vynucuje shodu typu formálního a skutečného parametru)
- rozepíšeme a zjednodušíme rovnosti na co nejjednodušší
- vyjádříme všechny typové proměnné pomocí minimální množiny typových proměnných
- určíme hledaný typ a dosadíme do něj vyjádření
- zohledníme typové kontexty, jsou-li nějaké
- volitelně převedeme typ do kanonického tvaru

Typování funkcí I.

Příklad 3.2.1: Určete typy výrazů:

a) `(&&) True`
b) `id "foo"`
c) `(&& False)`
d) `const True`
e) `const True False`
f) `(:[])`
g) `(:[]) True`
h) `[]:[]:[]`
i) `([]:[]):[]`

Typování funkcí II.

Příklad 3.2.2: Určete typy následujících výrazů:

a) `map fst`

b) `map (filter not)`

c) `const id '!' True`

d) `fst (fst, snd) (snd, fst) (True, False)`

e) `head [head] [tail] [[]]`

(Poslední dva případy jsou pro pokročilé.)

Typování funkcí III.

Příklad 3.2.3: Určete typy funkcí:

a) `swap (x,y) = (y,x)`

b) `cadr = head . tail`

c) `caar = head . head`

d) `twice f = f . f`

e) `comp12 g h x y = g (h x y)`

Typ definice funkce se vzory

Příklad 3.2.4: Určete typy následujících funkcí:

a)
```
sayLength [] = "empty"
sayLength x  = "noempty"
```

b)
```
mswap True  (x, y) = (y, x)
mswap False (x, y) = (x, y)
```

c)
```
gfst (x, _) = x
gfst (x, _, _) = x
gfst (x, _, _, _) = x
```

d)
```
foo True []     = True
foo True (_:_) = False
foo False       = False
```

První pohled na typové třídy

Někdy chceme mít polymorfní funkce, které však nelze aplikovat na zcela libovolné argumenty:

- `(+) :: Num a => a -> a -> a` nám umožňuje sčítat čísla, ale ne jiné věci (protože pro ně není možné dobře definovat sčítání)
- `(==) :: Eq a => a -> a -> Bool` umožňuje porovnat porovnatelné hodnoty
- `(<) :: Ord a => a -> a -> Bool` nám umožňuje uspořádat hodnoty, které jdou uspořádat, ale neumožňuje uspořádat například funkce
- `show :: Show a => a -> String` umožňuje převést hodnoty na `String`, pokud mají smysluplnou textovou reprezentaci (např. funkce nemají)

První pohled na typové třídy

Část typu před => označujeme jako **typový kontext**. Typový kontext může obsahovat i více omezení:
`(^) :: (Num a, Integral b) => a -> b -> a`.

Při typování výrazů, které obsahují takto kvantifikovaně polymorfní typy, je nutné dávat pozor na typový kontext.

Příklad 3.2.5: Určete typy následujících výrazů:
a) `(+ 3)`
b) `(+ 3.0)`
c) `filter (>= 2)`
d) ``(> 2) . (`div` 3)``

Další funkce na seznamech

Funkcí na seznamech je velké množství, mnohé užitečné lze nalézt v modulu `Data.List`[1], základní i přímo z `Prelude`[2].

Příklad 3.3.1: S pomocí interpretru zjistěte typy funkcí `and`, `or`, `all` a `any`. Zkuste je vyhodnotit na nějakých parametrech a přijít na to, co počítají (jejich název je vhodnou nápovědou).

Příklad 3.3.2: Zjistěte, co dělají následující funkce:

```
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

---

[1] http://hackage.haskell.org/package/base/docs/Data-List.html

[2] http://hackage.haskell.org/package/base/docs/Prelude.html#g:11

Funkce `zip`, `unzip`: intuice

Funkce `zip` a `unzip` umožňují převod mezi dvojicí seznamů a seznamem dvojic:

```
zip [1,2,3] ['a','b','c'] ⇝*
    [(1,'a'),(2,'b'),(3,'c')]
unzip [(1,'a'),(2,'b'),(3,'c')] ⇝*
    ([1,2,3],['a','b','c'])
```

Délka výsledku funkce `zip` je určena délkou kratšího seznamu:

```
zip [1,2,3,4] ['a','b'] ⇝* [(1,'a'),(2,'b')]
```

Funkce `zip`, `unzip` I.

Příklad 3.3.3: Funkci `zip :: [a] -> [b] -> [(a,b)]`, lze definovat následovně:

```
zip (x:s) (y:t) = (x,y) : zip s t
zip _     _     = []
```

a) Které dvojice parametrů vyhovují prvnímu řádku definice?
b) Přepište definici tak, aby první klauzule definice (první řádek) byla použita jako poslední klauzule definice.

Funkce `zipWith`: intuice

Funkce `zipWith` je zobecněním funkce `zip`:

- funkce `zip` spájí prvky pomocí datového konstruktoru uspořádané dvojice `(,)`
- funkce `zipWith` má navíc jako argument funkci, kterou určíme způsob spojení prvků ze seznamů

```
zipWith f [x,y] [z,q] ⇝* [f x z, f y q]
zipWith (,) [1,2] ['a','b'] ⇝* [(1,'a'),(2,'b')]
zipWith (*) [5,10,11] [3,4] ⇝* [15,40]
```

Funkce `zipWith` I.

Příklad 3.3.6: Jaká je hodnota následujících výrazů?

a) `zipWith (^) [1..5] [1..5]`

b) `zipWith (:) "MF" ["axipes", "ík"]`

c) `let fibs = [0,1,1,2,3,5,8,13] in zipWith (+) fibs (tail fibs)`

d) `let fibs = [0,1,1,2,3,5] in zipWith (/) (tail (tail fibs)) (tail fibs)`

Funkce `zipWith` II.

Příklad 3.3.8: Napište funkci, která zjistí, jestli jsou v seznamu typu `(Eq a) => [a]` některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

Lambda-abstrakce, eta-redukce, pointfree vs. pointwise zápis, curryfikace

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Lambda-abstrakce (λ-abstrakce) – teorie

Anonymní funkce, umožňuje zapsat funkci přímo v místě volání.

- vhodné pro funkce jako `map`, `filter`
- `map (\x -> x ^ 2 + 2 * x + 1) [1, 2, 3]`
- obecně zapisujeme `\<parameters> -> function_body`
- parametrů může být více
- parametry můžou obsahovat vzory: `\(x, y) -> x + y`
- nelze zapsat ekvivalent víceřádkové definice nebo vyjádřit rekurzi

Lambda-abstrakce I.

Příklad 4.1.1: Které z následujících výrazů jsou korektní?

a) `\x y -> 0`

b) `\f -> f 0`

c) `(\s -> "ahoj, " ++ s, "to: " ++ s)`

d) `\x -> x . \y -> y x`

e) `\[(x, y)] z -> testIt x y z`

f) `\(_) [_] -> ()`

g) `\x y x -> y + 2 * x`

h) `(\x y -> x y) (\x y -> x y) (\x y -> x y)`

i) `\a b -> a (\c d e -> b c (d e))`

j) `\[] -> ()`

Lambda-abstrakce II.

Příklad 4.1.4: V následujících výrazech proveďte aplikace λ-abstrakcí na hodnoty tam, kde to je možné. Funkce přitom nevyhodnocujte – zaměřte se skutečně jenom na aplikaci λ-abstrakcí. Také nepoužívejte η-redukci.

a) `(\t -> map t [1, 2, 3]) (+)`

b) `\t u -> t * u 2 3`

c) `\t u -> (t * u 2) 3`

d) `(\x y -> x + y) 4 0.3`

e) `(\n -> n * 10) (3 + 1)`

f) `(\f -> const map f filter) (head . head)`

g) `(\a b -> zipWith a [1..10] b) (\x y -> x * 10 + y) ((\t -> map (^2) t) [1..5])`

h) `(\x y -> x (map y)) (\s (a, b) -> s [a..b]) (\f -> f - 1)`

Eta-redukce (η-redukce)

Odstraňování formálních parametrů z definice funkce:

```
multiplyN n xs = map (n *) xs
multiplyN n    = map (n *)
multiplyN      = map . (*)
```

- někdy umožňuje funkci zapsat elegantněji
- často umožňuje zbavit se nutnosti použít λ-funkci
- nebezpečí nečitelného kódu:
  `(.) . (.)` vs. `\f g x y -> f (g x y)`

Pointfree vs. pointwise

- pointfree tvar je tvar bez formálních parametrů
- pointwise tvar je tvar se všemi formálními parametry odpovídajícími typu funkce

Převody pointfree ↔ pointwise I.

Příklad 4.2.1: Následující výrazy použijte v lokální definici a vyhodnoťte v interpretru jazyka Haskell na vhodných parametrech. Po úspěšné aplikaci výrazy upravujte tak, abyste se při jejich definici vyhnuli použití λ-abstrakce a formálních parametrů.

a) `\x -> 3 * x`
b) `\x -> x ^ 3`
c) `\x -> 3 + 60 `div` x ^ 2 > 0`
d) `\x -> [x]`
e) `\s -> "<" ++ s ++ ">"`
f) `\x -> 0 < 35 - 3 * 2 ^ x`
g) `\x y -> x ^ y`
h) `\x y -> y ^ x`
i) `\x y -> 2 * x + y`

Pomocné funkce `flip` a `const`

Některé funkce nelze přímo přepsat do pointfree tvaru, potřebujeme pomocné funkce:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

- `flip` předá 2 parametry funkci v opačném pořadí

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

- `const` zapomíná svůj druhý parametr a vrací první

Pozor, vyhodnocení `const` může změnit typ!

```
\x -> const x (x True) :: (Bool -> b) -> Bool -> b
\x -> x                :: a -> a
```

Převody pointfree $\leftrightarrow$ pointwise II.

Příklad 4.2.3: Převeďte následující výrazy do pointwise tvaru:

a) `(^2) . mod 4 . (+1)`

b) `(+) . sum . take 10`

c) `map f . flip zip [1, 2, 3]`

d) `(.)`

e) `flip flip 0`

f) `(.) (+) . (+)`

g) `(.(.))`

Převody pointfree $\leftrightarrow$ pointwise III.

Příklad 4.2.4: Určete typ následujících funkcí. Přepište tyto definice funkcí tak, abyste v jejich definici nepoužili λ-abstrakci a formální parametry (tj. chce se pointfree definice).

a) `f x y = y`

b) `h x y = q y . q x`

Curryfikace[1] – teorie

Umožnění částečné aplikace funkce:

- n-ární funkce v Haskellu je vnímána ne jako funkce, která bere n-tici, ale jako funkce, která očekává jeden parametr a vrací funkci, která bere $(n - 1)$ parametrů
- oblíbený koncept ve funkcionálních jazycích
- `const :: a -> b -> a` lze vnímat jako funkci, která očekává parametr typu `a` a vrací funkci typu `b -> a`
- oproti tomu `fst :: (a, b) -> a` je unární funkce na dvojici a nelze ji částečně aplikovat
- převody pomocí
  `curry :: ((a, b) -> c) -> a -> b -> c`
  `uncurry :: (a -> b -> c) -> (a, b) -> c`
- `const` $\equiv$ `curry fst`, `uncurry const` $\equiv$ `fst`

[1] Pojmenováno po matematikovi a logikovi jménem Haskell Brooks Curry

Curryfikace I.

Příklad 4.3.1: Definujte *unární* funkci `nebo` pro realizaci logické disjunkce a pomocí modifikátorů `curry` a `uncurry` definujte ekvivalenci mezi vámi definovanou funkcí `nebo` a předdefinovanou funkcí `(||)`.

Příklad 4.3.2: Analogicky k funkcím `curry` a `uncurry` definujte funkce

a) `curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d`

b) `uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d`

Curryfikace II.

Příklad 4.3.4: Převeďte funkce do pointfree tvaru:

a) `\(x, y) -> x + y`

b) `\x y -> nebo (x, y)` (`nebo = uncurry (||)`)

c) `\((x, y), z) -> x + y + z` (dodržte asociativitu operátoru +)

Líné vyhodnocování, nekonečné seznamy, akumulační funkce na seznamech

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Líné vyhodnocování

- Vyhodnotíme jenom potřebné a jenom jednou.
- Ovlivňuje i časovou složitost vyhodnocení.

```
   head [sum [] + sum [], product [1..100]]
⇝ sum [] + sum []
⇝ 0 + 0
⇝ 0
```

Příklad 5.2.1: Uvažte význam líného vyhodnocování v následujících výrazech:

a) `take 10 [1..]`
b) `let f = f in fst (2, f)`
c) `let f [] = 3 in const True (f [1])`
d) `0 * div 2 0`
e) `snd ("a" * 10, id)`

Líné vyhodnocování – úskalí

http://ro-che.info/ccc/11

Nekonečné seznamy

Užitečné seznamové funkce:

- `take`, `(!!)`, `repeat`, `replicate`, `cycle`, `iterate`
- notace `[a..c]`, `[a,b..c]`, `[a..]`, `[a,b..]` (i nečíselné seznamy)

Příklad 5.2.3: Pomocí některé z funkcí `iterate`, `repeat`, `replicate`, `cycle` vyjádřete nekonečné seznamy:

a) Seznam sestávající z hodnot `True`.
b) Rostoucí seznam všech mocnin čísla 2.
c) Rostoucí seznam všech sudých mocnin čísla 3.
d) Rostoucí seznam všech lichých mocnin čísla 3.
e) Alternující seznam `-1` a `1`: `[1,-1,1,-1, ...]`.
f) Seznam řetězců `["", "*", "**", "***", "****", ...]`.
g) Seznam zbytků po dělení 4 pro seznam `[1..]`: `[1,2,3,0,1,2,3,0,...]`.

Akumulační funkce na seznamech – motivace

Příklad 5.1.1: Definujte následující funkce rekurzivně:

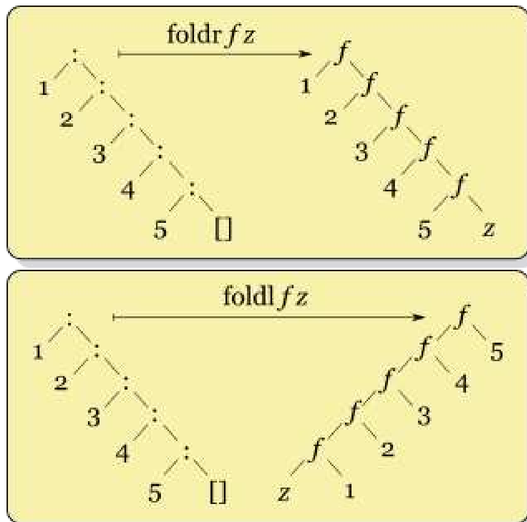
a) `product'` – součin prvků seznamu
b) `length'` – počet prvků seznamu
c) `map'` – funkci `map`

Co mají tyto definice společné? Jak by vypadalo jejich zobecnění?

Akumulační funkce:

- zpracují seznam na jednu hodnotu
- transformace seznamu dle struktury
- `foldr`, `foldr1`, `foldl`, `foldl1`

Akumulační funkce na seznamech – ilustrace



https://en.wikipedia.org/wiki/Fold_%28higher-order_function%29

Příklad 5.1.2: Určete, co dělají akumulační funkce s uvedenými argumenty. Najděte hodnoty, na které je lze tyto výrazy aplikovat, a ověřte pomocí interpretu.

a) `foldr (+) 0`
b) `foldr1 (\x s -> x + 10 * s)`
c) `foldl1 (\s x -> 10 * s + x)`

Příklad 5.1.3: Definujte funkci `subtractlist`, která odečte druhý a všechny další prvky *neprázdného* seznamu od jeho prvního prvku, tj. `subtractlist` [x_1, ..., x_n] = x_1 - x_2 - ... - x_n.

Příklad 5.1.5: Uvažme funkci: `foldr (.) id`

a) Jaký je význam uvažované funkce?
b) Jaký je její typ?
c) Uveďte příklad částečné aplikace této funkce na jeden argument.
d) Uveďte příklad úplné aplikace této funkce na kompletní seznam argumentů.

Akumulační funkce na seznamech III

Zhluboka se nadechněte, ve sbírce si otevřete příklad 5.1.8 (implementace funkcí pomocí akumulačních funkcí) a řešte jednotlivé podpříklady.

Práce se vstupem a výstupem, vlastní datové typy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Vstup a výstup v Haskellu

Běžné funkce v Haskellu nemají vedlejší efekty ani vnitřní stav.

- nesmí modifikovat soubory, vypisovat na obrazovku, načítat vstup…
- **zavolání funkce se stejnými parametry vrátí vždy stejný výsledek**

Občas nepraktické – znemožňuje to komunikaci s vnějším světem.

- speciální `IO` funkce, mají povoleny vedlejší efekty
- obecně typu `Type1 -> Type2 -> ... -> IO ReturnType`
- například `putStrLn :: String -> IO ()`, `getLine :: IO String`
- hodnota typu `IO a`: (vstupně-výstupní) **akce**, má **vnitřní výsledek** typu `a`
- k vnitřnímu výsledku nelze přistupovat přímo, používáme speciální jazykové konstrukce

IO: úvod

Příklad 6.1.1: Do interpretru zapište nejprve výraz `"**\n**\n"` a poté výraz `putStr "**\n**\n"`. Rozdíl chování interpretru vysvětlete.

IO: do-notace

Speciální konstrukce jazyka pro práci se `IO`: `do`-blok.

```
echo :: IO ()
echo = do
    putStrLn "Write something"
    line <- getLine
    let out = "You wrote: " ++ line
    putStrLn out
```

- návratovou hodnotu akce `getLine` extrahujeme pomocí `<-`
- blok musí končit akcí, její návratová hodnota je návratovou hodnotou bloku
- `let` uvnitř `do` platí od své definice až po konec bloku

IO: do-notace

Příklad 6.2.1: Definujte akci `getInt :: IO Int`, která ze standardního vstupu načte celé číslo. Využijte knihovní funkci `read :: (Read a) => String -> a`.
Můžete využít funkce `return :: a -> IO a`, která zabalí hodnotu do `IO` akce, která vrací tuto hodnotu.

Pozor: Funkce `return` neukončuje `do`-blok:

```
getSomething :: IO Int
getSomething = do
    return "ahoj"
    return 1
```

je akce s vnitřním výsledkem `1`.

IO: do-notace

S pomocí `IO` lze vytvářet spustitelné programy v Haskellu:

- stačí definovat funkci `main` typu `IO ()`, použije se jako vstupní bod

Příklad 6.2.2: Upravte a doplňte následující zdrojový kód tak, aby program vyžadoval a načetl postupně tři celá čísla a o nich určoval, zda mohou být délkami hran trojúhelníku. Hotový program přeložte do samostatného spustitelného souboru a otestujte.

```
main :: IO ()
main = do putStrLn "Dej mi jedno číslo:"
          x <- getLine
          print ((+) 1 (read x :: Int))
```

Spouštění a kompilace:

- kompilace: `ghc File.hs`, vytvoří binárku `File/File.exe`
- přímé spuštění (interpret): `runhaskell File.hs`

IO pomocí operátoru >>=

do-notace se překládá na použití operátoru:

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

- umožňuje přistoupit k vnitřnímu výsledku akce z funkce, která vrací akci

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

- pro řetězení akcí (bez použití výsledku první)

```
echo =
    putStrLn "Write something" >>
    getLine >>= \line ->
    let out = "You wrote: " ++ line in putStrLn out
```

IO pomocí operátoru >>=

Příklad 6.3.1: Uvažme následující program:

```
import Data.Char
main :: IO ()
main = getLine >>= putStr . filter isAlpha
```

a) Co program dělá?
b) Přepište program do `do`-notace.

IO

Příklad 6.3.2: Převeďte následující program v `do`-notaci na notaci s použitím >>=.

```
main = do
    f <- getLine
    s <- getLine
    appendFile f (s ++ "\n")
```

Vlastní datové typy

Můžeme si definovat vlastní datové typy:

```
data Shape = Circle Double
           | Rectangle Double Double
           | Point
           deriving (Eq, Show)
```

- definuje typ `Shape`
- hodnoty například
  - `Circle 42`
  - `Rectangle 3.5 2`
  - `Point`
- `deriving` část říká, do kterých typových tříd má typ patřit

Datové a typové konstruktory

```
data Shape = Circle Double
           | Rectangle Double Double
           | Point
           deriving (Eq, Show)
```

- Shape je (nulární) typový konstruktor
- tři datové konstruktory („funkce“ vytvářející typ):
  - unární: `Circle :: Double -> Shape`
  - binární: `Rectangle :: Double -> Double -> Shape`
  - nulární: `Point :: Shape`
- obojí začínají velkým písmenem nebo dvojtečkou
- lze je použít v definici podle vzoru

```
area :: Shape -> Double
area Point           = 0
area (Circle r)      = pi * r * r
area (Rectangle a b) = a * b
```

Datové a typové konstruktory

Příklad 6.4.3: Identifikujte nově vytvořené typové a datové konstruktory a určete jejich aritu.

a) `data X = X`

b) `data A = X | Y String | Z Int Int`

c) `data B a = A | B a | C a`

d) `data C = D C`

e) `data E = E (E, E)`

f) `type String = [Char]`

Vlastní datové typy

Příklad 6.4.1: Mějme datový typ `Day` představující dny v týdnu definovaný níže. Definujte funkci `weekend :: Day -> Bool`, která o zadaném dni určí, jestli je to víkendový den.

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving (Show, Eq, Ord)
```

Příklad 6.4.2: Vytvořte nový datový typ `Jar` představující sklenici ve spíži. Každá sklenice je v jednom z následujících stavů:

- je prázdná (`EmptyJar`);
- je v ní ovocná marmeláda (`Jam`), pamatujeme si typ ovoce, ze kterého byla vyrobena (`String`);
- jsou v ní okurky (`Cucumbers`), o nich si nemusíme nic pamatovat, stejně se hned snědí;
- je v ní kompot (`Compote`), pamatujeme si rok výroby (`Int`).

Vaší úlohou je pak nadefinovat funkci `stale :: Jar -> Bool`, která určí, jestli je obsah dané sklenice již zkažený. Prázdné sklenice, okurky ani marmelády se nekazí (možná je to tím, že se příliš rychle snědí), kompoty se pokazí za 10 let od zavaření (zadefinujte si celočíselnou konstantu `today`, ve které budete mít aktuální rok).

Rekurzivní datové typy

Příklad 6.4.6: Uvažme následující definici typu `Expr`:

```
data Expr = Con Float
          | Add Expr Expr | Sub Expr Expr
          | Mul Expr Expr | Div Expr Expr
```

a) Uveďte výraz typu `Expr`, který představuje hodnotu 3.14.

b) Definujte funkci `eval :: Expr -> Float`, která vrátí hodnotu daného výrazu.

Typové třídy II, `Maybe`, rekurzivní datové typy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Typové třídy

- vyjadřují nějakou vlastnost hodnot (porovnatelnost, převeditelnost na text, ...)
- obsahuje funkce, které implementují tuto vlastnost (porovnávání, ...)
- typ patří do třídy $\sim$ instance typové třídy
  - automatické instance (`deriving`)
  - ruční instance

```
data Shape = P | C Double | R Double Double
instance Eq Shape where
    P == P                      = True
    (C r1) == (C r2)            = r1 == r2
    (R x1 y1) == (R x2 y2) = x1 == x2 && y1 == y2
    _ == _                      = False
```

Typové třídy

- často lze některé funkce odvodit z jiných
  - `(/=)` z `(==)`
  - `(>)`, `(<=)`, `(>=)` z `(<=)` a `(==)`
- *minimální kompletní implementace*: skupina funkcí, které musíme implementovat, aby typová třída fungovala
  - pro `Eq`: `(==)`
  - pro `Ord`: `(<=)`

Příklad 7.1.2: Uvažte datový typ představující semafor zadefinovaný níže.

```
data TrafficLight = Red | Orange | Green
```

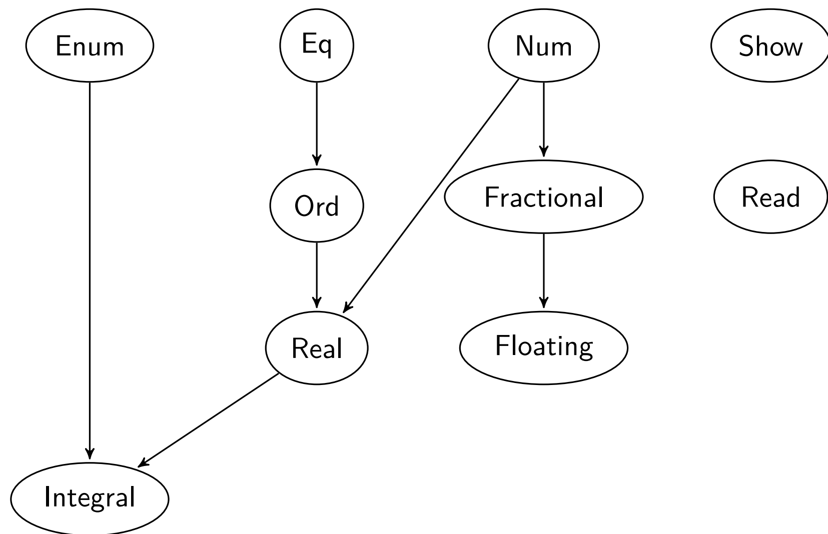
Umožněte zobrazování hodnot tohoto typu, jejich vzájemné porovnávání a řazení (zelená $<$ oranžová $<$ červená). Řečeno jinak, napište instanci `TrafficLight` pro typové třídy `Show`, `Eq` a `Ord`.

Příklad 7.1.3: Zadefinujme vlastní typ uspořádaných dvojic s názvem `PairT`. Tento typ bude mít pouze jeden binární datový konstruktor `PairD` (viz definice níže).

```
data PairT a b = PairD a b
```

Vytvořte instanci `PairT` pro typové třídy `Show`, `Eq` a `Ord`. Ať jsou si dvě dvojice rovny právě tehdy, pokud jsou si rovny po složkách. Uspořádání definujte jako lexikografické po složkách. Zobrazování hodnot tohoto typu nechť je slovní (tedy namísto obligátního `(1,2)` vypište třeba `"pair of 1 and 2"`).

Základní typové třídy



Definice typové třídy

```
class Boolable a where
    getBool :: a -> Bool
```

(typ `getBool :: Boolable a => a -> Bool`)

Do třídy přidáme datové typy obvyklým způsobem:

```
instance Boolable Bool where
    getBool x = x

instance Boolable Int where
    getBool 0 = False
    getBool _ = True
```

Použití:

```
myIf :: Boolable a => a -> b -> b -> b
myIf x t f = if getBool x then t else f
```

Datový typ Maybe

Definice (z `Prelude`):

```
data Maybe a = Nothing | Just a
             deriving (Eq, Ord, Show, Read)
```

- parametrizovaný datový typ `Maybe a` → konkrétní typy: `Maybe Int`, `Maybe [String]`, `Maybe (Int, [Bool])`,...
- reprezentuje hodnotu, jejíž výpočet může selhat
- hodnota může být přítomna (`Just 1`) nebo chybět (`Nothing`)
- hodnoty extrahujeme pomocí vzorů

```
justOrVal :: a -> Maybe a -> a
justOrVal x Nothing  = x
justOrVal _ (Just y) = y
```

Datový typ Maybe

Příklad 7.2.2: S využitím typového konstruktoru `Maybe` definujte funkci
`divlist :: Integral a => [a] -> [a] -> [Maybe a]`, která celočíselně podělí dva celočíselné seznamy „po složkách“, tj.

$$\texttt{divlist [x}_1\texttt{, ..., x}_n\texttt{] [y}_1\texttt{, ..., y}_n\texttt{]} \rightsquigarrow^* \texttt{[div x}_1\texttt{ y}_1\texttt{, ..., div x}_n\texttt{ y}_n\texttt{]}$$

a ošetří případy dělení nulou.

Příklad 7.2.3: Uvažte následující datový typ:

```
data MyMaybe a = MyNothing
               | MyJust a
               deriving (Show, Read, Eq)
```

Deklarujte typ `MyMaybe` jako instanci typové třídy `Ord`. Za jakých podmínek může být typ `MyMaybe a` instancí třídy `Ord`?

Příklad 7.3.3: Uvažte následující rekurzivní datový typ představující binární strom s ohodnocenými uzly:

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

Definujte následující funkce nad binárními stromy:

a) `treeSize :: BinTree a -> Integer`, která spočítá počet uzlů ve stromě
b) `treeMax :: Ord a => BinTree a -> a`, která najde maximální hodnotu v uzlech stromu
c) `listTree :: BinTree a -> [a]`, která převede všechny hodnoty uzlů ve stromu do seznamu
d) `longestPath :: BinTree a -> [a]`, která najde nejdelší cestu ve stromě a vrátí ohodnocení na ní

Příklad 7.3.4: Pro datový typ `BinTree` označíme *výškou stromu* počet uzlů na cestě z kořene do nejvzdálenějšího listu.

a) Definujte funkci `fullTree :: Int -> a -> BinTree a`, která pro volání `fullTree n v` vytvoří binární strom výšky `n`, ve kterém jsou všechny větve stejně dlouhé a všechny uzly ohodnocené hodnotou `v`.

b) Definujte funkci `height :: BinTree a -> Int`, která určí výšku stromu.

c) Definujte funkci
`treeZip :: BinTree a -> BinTree b -> BinTree (a,b)`
jako analogii seznamové funkce `zip`.

Příklad 7.3.5: Uvažme datový typ `BinTree`.

a) Definujte funkci `treeRepeat :: a -> BinTree a` jako analogii seznamové funkce `repeat`. Funkce tedy vytvoří nekonečný strom, který má zadanou hodnotu v každém uzlu.

b) Pomocí funkce `treeRepeat` vyjádřete nekonečný binární strom `nilTree`, který má v každém uzlu prázdný seznam.

c) Definujte funkci
`treeIterate :: (a->a) -> (a->a) -> a -> BinTree a`
jako analogii seznamové funkce `iterate`. Levý potomek každého uzlu bude mít hodnotu vzniklou aplikací první zadané funkce a pravý aplikací druhé zadané funkce.

Rekurzivní datové typy

Příklad 7.3.8: Uvažte typ *n*-árních stromů definovaný následovně:

```
data NTree a = NTree a [NTree a]
             deriving (Show, Read)
```

Definujte následující:

a) funkci `ntreeSize :: NTree a -> Integer`, která spočítá počet uzlů ve stromě
b) funkci `ntreeSum :: Num a => NTree a -> a`, která sečte ohodnocení všech uzlů stromu
c) instance `Eq` a `Ord` pro `NTree a`
d) funkci `ntreeMap :: (a -> b) -> NTree a -> NTree b`, která bere funkci a strom, a aplikuje danou funkci na ohodnocení v každém uzlu:

```
ntreeMap (+1) (NTree 0 [NTree 1 [], NTree 41 []])
    ⇝* NTree 1 [NTree 2 [], NTree 42 []]
```

Pokročilé funkce nad rekurzivními datovými typy, intensionální seznamy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Intenzionální definice seznamů

```
[ 2*n | n <- [10..15], odd n ]
    ⇝* [22, 26, 30]
```

Prvky seznamu generovány společnými pravidly:

- zkouší se všechny prvky z generátoru
- pro prvky vyhovující kvalifikátoru (podmínce) se do výsledného seznamu vloží výraz na levé straně

Intenzionální definice seznamů

Generátory:

- může jich být i více
- vyhodnocují se zleva doprava
- můžou používat vzory

Kvalifikátory:

- musí být výrazy typu `Bool`
- může jich být i více
- výraz na levé straně se přidá pouze pokud platí všechny podmínky zároveň

Lokální definice:

- ve tvaru `let x = ...`, může jich být i více

Intensionální seznamy

Příklad 8.1.1: Pomocí intensionálních seznamů definujte funkci `divisors`, která k zadanému přirozenému číslu vrátí seznam jeho kladných dělitelů.

Příklad 8.1.2: Intensionálním způsobem zapište následující výrazy:

a) `map f s`
b) `filter p s`
c) `map f (filter p s)`
d) `repeat x`
e) `replicate n x`
f) `filter p (map f s)`

Příklad 8.1.4: Intensionálním způsobem zapište následující seznamy nebo funkce:

a) `[1,4,9,...,k^2]` (pro pevně dané `k`)
b) funkci `f`, která ze seznamu seznamů vybere jenom ty delší než 3 prvky
c) `"*****"`
d) `["","*","**","***",...]`
e) seznam seznamů `[[1],[1,2],[1,2,3],...]`
f) `[[1],[2,2,2],[3,3,3,3,3],[4,4,4,4,4,4,4],...]` (hledejte vztah mezi číslem a počtem jeho výskytů)
g) `["z","yy","xxx",...,"aaa...aaa"]` (znak a se v posledním členu vyskytuje přesně 26krát)
h) následující seznam „2D matic“

```
[ [[1]],
  [[1,1],[1,1]],
  [[1,1,1],[1,1,1],[1,1,1]], ... ]
```

Foldy na různých datových typech

Foldy na datových strukturách (odpovídá pojmu *katamorfismus*):

- „projdou“ rekurzivně celou danou strukturu
- nahradí všechny datové konstruktory zadanými funkcemi
  - arita musí souhlasit
  - typová struktura musí souhlasit
- výsledkem je nová struktura

Příklad ze standardní knihovny: `foldr`

- náhrada `(:)` za první argument `foldr`
- náhrada `[]` za druhý argument `foldr`

Pozor: `foldl` není katamorfismem (foldem) v tomhle smyslu!

Fold na seznamech (definice)

Datový typ seznam:

```
data [a] = (:) a [a] | []
```

Typy jeho datových konstruktorů:

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

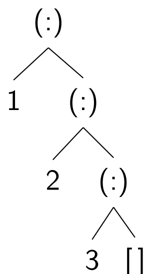
Fold na seznamech:

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

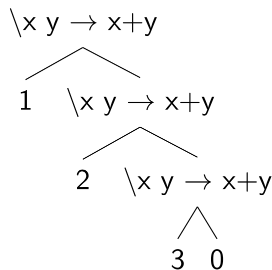
Fold na seznamech (příklad)

```
data [a] = (:) a [a] | []

foldr (+) 0 [1,2,3]
    ⇝ foldr (\x y -> x+y) 0 (1:(2:(3:[])))
    ⇝* 1+2+3+0 ⇝* 6
```



foldr (+) 0 ⟶



⇝* 6

Fold na Peanových číslech (definice)

Datový typ Peanove čísla:

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

Typy jeho datových konstruktorů:

```
Succ :: Nat -> Nat
Zero :: Nat
```

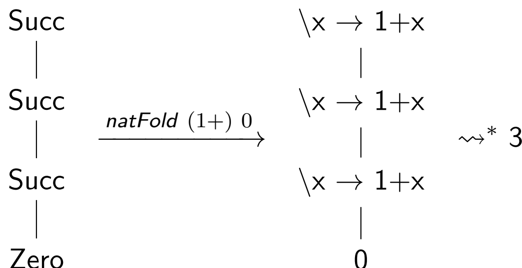
Fold na Peanových číslech:

```
natFold :: (a -> a) -> a -> Nat -> a
natFold s z (Succ n) = s (natFold s z n)
natFold s z Zero     = z
```

Fold na Peanových číslech (příklad)

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

```
natFold (1+) 0 (Succ(Succ(Succ(Zero))))
    ⇝ natFold (\x -> 1+x) 0 (Succ(Succ(Succ(Zero))))
   ⇝* 1+(1+(1+(0))) ⇝* 3
```



Fold na binárních stromech (definice)

Datový typ binární strom:

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

Typy jeho datových konstruktorů:

```
Node  :: a -> BinTree a -> BinTree a -> BinTree a
Empty :: BinTree a
```

Fold na binárních stromech:

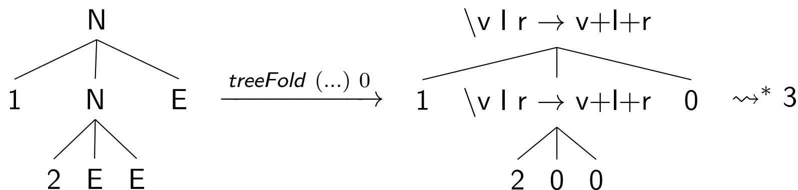
```
treeFold :: (a -> b -> b -> b) -> b -> BinTree a -> b
treeFold n e (Node v l r) = n v (treeFold n e l)
                                (treeFold n e r)
treeFold n e Empty        = e
```

Fold na binárních stromech (příklad)

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

treeFold (\v l r -> v+l+r) 0 (N 1 (N 2 E E) E)
```

$\rightsquigarrow^*$ `1+(2+0+0)+0` $\rightsquigarrow^*$ `3`



Foldy na binárních stromech

Zhluboka se nadechněte, ve sbírce si otevřete příklad 8.2.2 (implementace funkcí na binárních stromech pomocí foldů) a řešte jednotlivé podpříklady.

Soubor s definicí datového typu `BinTree a`, funkce `treeFold` a ukázkovými stromy ze zadání najdete v ISu:

`https://is.muni.cz/el/1433/podzim2014/IB015/um/seminars/code/treeFold.hs`

Opakování

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Příklad 9.1.1: Otypujte následující výrazy:

a) `head [id, not]`

b) `\f -> f 42`

c) `\t x -> x + x > t x`

d) `\xs -> filter (> 2) xs`

e) `\f -> map f [1,2,3]`

f)
```
foo f True  = map f [1,2,3]
foo f False = filter f [1,2,3]
```

g) `\(p,q) z  ->  q (tail z) : p (head z)`

Části a, b, d, e případně f

Příklad 9.1.3: Definujte funkci
`minmax :: Ord a => [a] -> (a, a)`, která pro daný neprázdný seznam *v jednom průchodu* spočítá minimum i maximum.
Funkci definujte jednou rekurzivně a jednou pomocí `foldr` nebo `foldl` (ne `foldr1`, `foldl1`).
Dále implementujte funkci
`minmaxBounded :: (Ord a, Bounded a) => [a] -> (a, a)`,
která funguje i na prázdných seznamech. Využijte konstant
`minBound :: Bounded a => a`,
`maxBound :: Bounded a => a`.
Najděte datový typ, pro který `minmaxBounded` nefunguje.
Poznámka: Může se stát, že interpretr bude zmatený z požadavku `Bounded a` a nebude schopen sám vyhodnotit výrazy jako `minmaxBounded [1,2,3]`. V takovém případě explicitně určete typ prvků seznamu, například:
`minmaxBounded [1,2,3::Int]`.

Příklad 9.1.4: Převeďte následující funkce do pointwise tvaru a přepište je s pomocí intensionálních seznamů a bez použití funkcí `map`, `filter`, `curry`, `uncurry`, `zip`, `zipWith`.

a) `map . uncurry`

b) `\f xs -> zipWith (curry f) xs xs`

c) ``map (* 2) . filter odd . map (* 3) . map (`div` 2)``

d) `map (\f -> f 5) . map (+)`

Různé funkce

Příklad 9.1.13: Co dělají následující funkce?

a) `f1 = flip id 0`

b) `f2 = flip (:) []`

c) `f3 = zipWith const`

d) `f4 p = if p then ('/':) else id`

e) `f5 = foldr id 0`

f) `f6 = foldr (const not) True`

Pointfree/pointwise

Příklad 4.2.2: Převeďte následující funkce do pointfree tvaru:

a) `\x -> (f . g) x`

b) `\x -> f . g x`

c) `\x -> f x . g`

Příklad 4.3.4: Převeďte funkce do pointfree tvaru:

a) `\(x, y) -> x + y`

b) `\x y -> nebo (x, y)` (`nebo = uncurry (||)`)

c) `\((x, y), z) -> x + y + z` (dodržte asociativitu operátoru +)

Příklad 9.1.2: Uvažte následující datový typ:

```
data Foo a = Bar [a]
           | Baz a
```

Určete typy následujících výrazů:

a)
```
getlist (Bar xs) = xs
getlist (Baz x ) = x : []
```

b)
```
\foo -> foldr (+) 0 (getlist foo)
```

Vstup a výstup

Příklad 9.1.5: Otypujte následující `IO` výrazy a převeďte je do do-notace: Uvažujte při tom následující typy `(>>=)`, `(>>)`, `return`:

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

a) `readFile "/etc/passwd" >> putStrLn "bla"`

b) `\f -> putStrLn "bla" >>= f`

c) `getLine >>= \x -> return (read x)`

d)
```
foo :: Integer
foo = getLine >>= \x -> read x
```

Úvod do Prologu, backtracking, unifikace, SLD stromy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Interpret jazyka Prolog

Interpret/kompilátor: *SWI-Prolog* (příkaz `swipl`)
Základní příkazy/predikáty (musí končit tečkou):

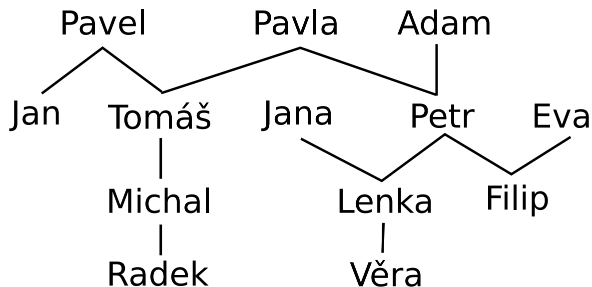
- **`help/0`** Zobrazí základní nápovědu o použití nápovědy.
- **`help/1`** Zobrazí nápovědu k predikátu v argumentu.
- **`apropos/1`** Zobrazí predikáty, které mají v názvu nebo popisu zadaný výraz.
- **`consult/1`** Načte (zkompiluje) zdrojový kód ze zadaného souboru. Nebo také **`['file.pl']`** a **`[file]`**.
- **`make/0`** Znovu načte zdrojový kód ze změněných souborů.
- **`halt/0`** Ukončí interpret *SWI-Prolog*.

Další řešení si vyžádáme středníkem (`;`),
výpočet ukončíme tečkou (`.`).

Příklad 10.1.2: Načtěte rodokmen ze souboru *pedigree.pl* do prostředí Prologu. Soubor naleznete v ISu a v příloze sbírky. Formulujte vhodné dotazy a pomocí interpretru zjistěte odpovědi na následující otázky:

a) Je Petr rodičem Lenky?
b) Je Petr rodičem Jana?
c) Jaké děti má Pavla?
d) Má Petr dceru?
e) Kdo je otcem Petra?
f) Které dvojice otec-syn známe?

Rodokmen – obrázek



Názvosloví

```
human(petr).
dog(nero).
friends(X, Y) :- human(X), dog(Y).

?- friends(petr, nero).
```

predikát = seznam faktů a pravidel se stejným názvem a aritou identifikátoru v hlavě

- **pravidlo** = hlava, jeden nebo více cílů v těle
- **fakt** = jenom hlava, tělo je prázdné

program = množina predikátů

- **dotaz** = prázdná hlava, jeden nebo více cílů v těle

Pokročilejší rodinné vztahy I.

Příklad 10.1.3: Pro rodokmen ze souboru *pedigree.pl* naprogramujte následující predikáty:

a) `child/2`, který uspěje, jestliže první argument je dítětem druhého.

b) `grandmother/2`, který uspěje, jestliže první argument je babičkou druhého.

c) `stepBrother/2`, který uspěje, jestliže první argument je nevlastním bratrem druhého argumentu (mají tedy právě jednoho společného rodiče).

Pokročilejší rodinné vztahy II.

Příklad 10.1.4: Pro rodokmen ze souboru *pedigree.pl* napište predikát `descendant/2`, který uspěje, když je první argument potomkem druhého (ne nutně přímý). Bez použití interpretru určete, v jakém pořadí budou nalezeni potomci Pavly, když použijeme dotaz `?- descendant(X,pavla)`. Jaký vliv má pořadí klauzulí a cílů v predikátu `descendant` na jeho funkci?

Tipy pro psaní rekurzivních predikátů

- Fakta pište dříve než pravidla.
- Jednoduché podmínky pište co nejdřív.
- Nerekurzivní volání napište dříve než rekurzivní.
- Netvořte levorekurzivní pravidla.
- Pokuste se využít optimalizaci posledního volání.

Unifikace

Dva termy jsou unifikovatelné, pokud jsou identické, anebo je možné zvolit hodnoty proměnných použitých v unifikovaných termech tak, aby po dosazení těchto hodnot byly termy identické.

- `a(X) = a(petr)`
  unifikovatelné, substituce [X/petr]
- `a(X,Y) = a(petr)`
  neunifikovatelné, různá arita funktorů
- `a(X,Y) = b(petr, Z)`
  neunifikovatelné, různé jména funktorů
- `a(X,X) = a(petr, Z)`
  unifikovatelné, substituce [X/Z, Z/petr]
- `a(X,eva) = a(petr, X)`
  neunifikovatelné

Unifikace v SWI-Prologu nedělá test na sebevýskyt (*occurs check*)!

Unifikace – příklad

Příklad 10.2.1: Které unifikace uspějí, které ne a proč? Jaká substituce je výsledkem provedených unifikací?

a) `a(X) = b(X)`
b) `X = a(Y)`
c) `a(X) = a(X,X)`
d) `X = a(X)`
e) `jmeno(X,X) = jmeno(Petr,plus)`
f) `s(1,a(X,q(w))) = s(Y,a(2,Z))`
g) `s(1,a(X,q(X))) = s(W,a(Z,Z))`
h) `X = Y, P = R, s(1,a(P,q(R))) = s(Z,a(X,Y))`

(Bez částí g, h).

SLD stromy – teorie

SLD stromy = způsob vizualizace výpočtu v Prologu

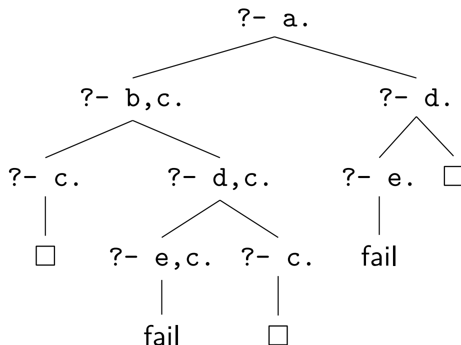
- v kořenu je dotaz
- jednotlivé podcíle se vyhodnocují zleva doprava
- při více možnostech se strom větví
- hrany jsou anotované provedenými unifikacemi
- prázdné listy značí úspěch
- neúspěšné větve označeny výrazem **fail**

SLD stromy – jednoduchá ukázka

Příklad 10.3.1: Uvažme následující program a dotaz `?- a.` Nakreslete odpovídající výpočetní SLD strom.

```
a :- b,c.
a :- d.
b.
b :- d.
c.
d :- e.
d.
```

SLD stromy – jednoduchá ukázka (řešení)



SLD stromy – složitější případy

Příklad 10.3.2: Uvažme následující databázi faktů:

```
r(a,b).
r(a,c).
r(b,d).

f1(a).
f1(X) :- f1(Y), r(Y,X).

f2(X) :- f2(Y), r(Y,X).
f2(a).

g1(a).
g1(X) :- r(Y,X), g1(Y).

g2(X) :- r(Y,X), g2(Y).
g2(a).
```

Zdůvodněte chování interpretru pro následující dotazy a nakreslete odpovídající SLD stromy.

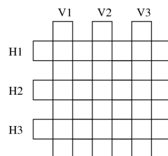
SLD stromy – složitější případy

Zdůvodněte chování interpretru pro následující dotazy a nakreslete odpovídající SLD stromy.

```
?- f1(X).
?- f2(X).
?- g1(X).
?- g2(X).
```

Příklad 10.2.4: Pro níže uvedenou databázi slov napište predikát `crossword/6`, který vypočítá, jak vyplnit uvedenou křížovku. První tři argumenty představují slova uváděná vertikálně shora dolů, druhé tři argumenty představují slova uváděná horizontálně zleva doprava.

```
word(astante,  a,s,t,a,n,t,e).
word(astoria,  a,s,t,o,r,i,a).
word(baratto,  b,a,r,a,t,t,o).
word(cobalto,  c,o,b,a,l,t,o).
word(pistola,  p,i,s,t,o,l,a).
word(statale,  s,t,a,t,a,l,e).
```



Číselné operace, práce se seznamy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Aritmetické operátory

Příklad 11.1.1: Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 = 1 + 1`
b) `1 + 1 = 1 + 1`
c) `2 is 1 + 1`
d) `1 + 1 is 1 + 1`
e) `X = 1 + 1`
f) `X is 1 + 1`
g) `2 is 1 + X`
h) `X = 1, 2 is 1 + X`

Aritmetické operátory

Příklad 11.1.1: Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 = 1 + 1`
b) `1 + 1 = 1 + 1`
c) `2 is 1 + 1`
d) `1 + 1 is 1 + 1`
e) `X = 1 + 1`
f) `X is 1 + 1`
g) `2 is 1 + X`
h) `X = 1, 2 is 1 + X`

- při unifikaci nemají aritmetické operátory speciální význam
  - `1 + 1` odpovídá termu `+(1, 1)`
- predikát `is` aritmeticky vyhodnotí pravou stranu, výsledek unifikuje s levou
  - pravá strana nesmí obsahovat neinstanciované proměnné

Aritmetické operátory

Příklad 11.1.2: Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `X = 2`
b) `X == 2`
c) `X = 2, X == 2`
d) `X =:= 2`
e) `1 + 1 == 2`
f) `1 + 1 =:= 2`
g) `2 =:= 1 + 1`

Aritmetické operátory

Příklad 11.1.2: Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `X = 2`
b) `X == 2`
c) `X = 2, X == 2`
d) `X =:= 2`
e) `1 + 1 == 2`
f) `1 + 1 =:= 2`
g) `2 =:= 1 + 1`

- `==` je test na identitu termů (bez unifikace)
- `=:=` je test na aritmetickou shodu (vyhodnocuje obě strany)

Aritmetické operátory

Příklad 11.1.3: Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 < 2 + 1`
b) `1 + 2 =< 2 + 1`
c) `1 + 2 >= 1`
d) `2 * 3 > 3 * 1.5`
e) `1 + 1 \== 2`
f) `1 + 1 =\= 2`
g) `1 + 2 =\= 2`

Aritmetické operátory

Příklad 11.1.3: Vyhodnoťte následující výrazy a vysvětlete chování jednotlivých operátorů.

a) `2 < 2 + 1`
b) `1 + 2 =< 2 + 1`
c) `1 + 2 >= 1`
d) `2 * 3 > 3 * 1.5`
e) `1 + 1 \== 2`
f) `1 + 1 =\= 2`
g) `1 + 2 =\= 2`

- relační operátory <, >, =<, >= aritmeticky vyhodnocují obě strany
- \== je test na neidentitu
- =\= je aritmetická nerovnost (vyhodnotí obě strany)

Aritmetika: funkce

Příklad 11.1.9: Napište predikát `powertwo/1`, který uspěje, pouze když číslo zadané jako argument je mocninou dvou. Využijte fakt, že mocniny dvou lze opakovaně beze zbytku dělit dvěma, dokud nedostaneme jedna. Můžete využít binární predikáty `mod` pro modulo a `//` pro celočíselné dělení.

Příklad 11.1.8: Implementujte predikát `fact/2`, který při dotazu `fact(`m`, `n`)` uspěje, pokud $m > 0$ a $n = m!$. V ostatních případech může interpret cyklit.
Predikát by měl rovněž fungovat při volání ve tvaru `fact(`n`, Res)`, kde `Res` je volná proměnná, a unifikovat tuto proměnnou s $n!$.

Seznamy

- výčtem prvků: `[1,2,3]`, `[a, 1, [3, b]]`
- ve tvaru `[ {vycet zacatku} | {seznam} ]`:
  `[1,2,3 | [5] ]`, `[ a, [1], 1 | [[]] ]`
- predikát `is_list/1` testuje, zda je argument seznam

Příklad 11.2.1: Které z následujících zápisů představují korektní zápis seznamu? Pokud je zápis korektní, určete počet prvků v daném seznamu.

a) `[1|[2,3,4]]`
b) `[1,2,3|[]]`
c) `[1|2,3,4]`
d) `[1|[2|[3|[4]]]]`
e) `[1,2,3,4|[]]`
f) `[[]|[]]`
g) `[[1,2]|4]`
h) `[[1,2],[3,4]|[5,6,7]]`

Funkce na seznamech

V hlavě můžeme používat unifikaci v seznamu:
`split([X|XS], X, XS).`

Příklad 11.2.2: Implementujte vlastní verze predikátů, které pro seznamy počítají standardní seznamové funkce, které znáte z Haskellu. Konkrétně imitujte funkce `head`, `tail`, `last` a `init`.

Příklad 11.2.11: Napište predikát `nth/3`, který vrátí *n*-tý prvek seznamu. Například dotaz `?- nth(4, [5,2,7,8,0], X).` uspěje se substitucí `X = 8`.

Funkce na seznamech

Zabudované seznamové funkce:

- `length(List, Length)` zjistí délku seznamu,
- `member(Elem, List)` zjistí, zda je `Elem` obsažen v seznamu `List`,
- `append(XS, YS, ZS)` spojí seznamy `XS` a `YS` do seznamu `ZS`.

Funkce na seznamech

Definice `append/3` z knihovny:

```
append([], YS, YS).
append([X|XS], YS, [X|ZS]) :- append(XS, YS, ZS).
```

Příklad 11.2.3: Napište následující predikáty pro práci se seznamy za pomoci vestavěného predikátu `append/3`.

a) `prefix/2` uspěje, jestliže je první argument prefixem seznamu ve druhém argumentu.
b) `suffix/2` uspěje, jestliže je první argument sufixem seznamu ve druhém argumentu.
c) `element/2` uspěje, jestliže je první argument členem seznamu ve druhém argumentu.
d) `adjacent/3` uspěje, jestliže jsou první dva prvky členy seznamu ve třetím argumentu a jsou vedle sebe (v daném pořadí).
e) `sublist/2` uspěje, jestliže je seznam v prvním argumentu podseznamem seznamu v druhém argumentu.

Koncová rekurze: akumulátory

Výpočet lze zrychlit, pokud je nejvýše jedno rekurzivní volání, a to jako poslední podcíl:

- pomocí pomocného predikátu a **akumulátoru**
- extra parametr, v němž střádáme mezivýsledek

```
myLength([], 0).
myLength([_|XS], L) :- myLength(XS, L1), L is L1 + 1.

lengthAcc(XS, L) :- lengthAcc2(XS, 0, L).

% druhy parametr je akumulator
lengthAcc2([], L, L).
lengthAcc2([_|XS], A, L) :- A1 is A + 1,
                            lengthAcc2(XS, A1, L).
```

Koncová rekurze: akumulátory

Příklad 11.2.12: Definujte následující predikáty s pomocí akumulátoru:

a) `listSum/2`, který sečte seznam čísel a na nečíselném seznamu neuspěje (použijte `number/1`),
b) `fact/2`, který spočítá faktoriál zadaného čísla,
c) `fib/2`, který (efektivně) spočítá n-tý člen Fibonacciho posloupnosti.

Řezy, negace, predikáty pro všechna řešení, základy I/O

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Operátor řezu: !

- podcíl, který vždy uspěje
- eliminuje další volby ve výpočetním stromu: od okamžiku unifikace hlavy klauzule po výskyt !

```
?- member(X, [1,2,3]).
X = 1 ;
X = 2 ;
X = 3.
```

```
?- member(X, [1,2,3]), !.
X = 1.

% zadna dalsi reseni
```

Použití:

- zefektivnění výpočtu
- omezení duplicitních řešení
- negace

Operátor řezu: příklad

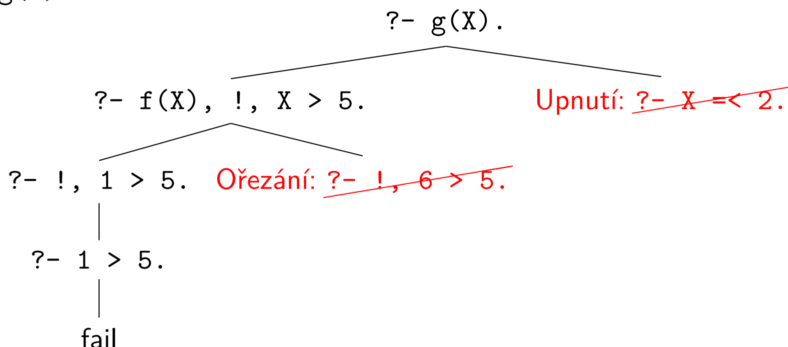
Odstranění zbytečné možnosti dotazu na další řešení:

```
powertwo(1) :- !.
powertwo(X) :- X mod 2 =:= 0, Y is X // 2,
    powertwo(Y).
```

Efekty řezu na výpočetní strom

Upnutí: Další pravidla a fakta pro aktuální cíl se ignorují.
Ořezání: Fixování voleb provedených při vyhodnocování těla pravidla po operátor řezu.

```
f(1).
f(6).
g(X) :- f(X), !, X > 5.
g(X) :- X =< 2.
```



Příklad 12.1.2: Jaký je rozdíl mezi následujícími definicemi predikátů `member/2`? Ve kterých odpovědích se budou lišit?

a)
```
mem1(H, [H|_]).
mem1(H, [_|T]) :- mem1(H, T).
```

b)
```
mem2(H, [H|_]) :- !.
mem2(H, [_|T]) :- mem2(H, T).
```

c)
```
mem3(H, [K|_]) :- H == K.
mem3(H, [K|T]) :- H \== K, mem3(H, T).
```

Příklady: řez

Příklad 12.1.1: Navrhněte predikát `max/3`, který uspěje, jestliže je číslo ve třetím argumentu maximem čísel z prvních dvou argumentů. Uveďte řešení bez použití řezu i s ním.

Příklady: řez

Příklad 12.1.5: Napište predikát `remove/3`, který odstraní všechny výskyty prvního argumentu ze seznamu ve druhém argumentu a výsledný seznam unifikuje do třetího argumentu.

Predikát `fail`, negace

`fail`

- cíl, který vždy selže
- spolu s řezem: selhání při splnění podmínky
- lze jím implementovat negaci

```
notnum(X) :- number(X), !, fail.
notnum(_).
```

Negace (\+)

- negace jako neúspěch
- jen zkratka pro předchozí konstrukci
- doporučuje se používat jen na cíle s plně instanciovanými argumenty
- `\+ P` vyžaduje konečné odvození `P`

```
notnum(X) :- \+ number(X).
```

Negace predikátu s proměnnými

```
zakladni(cervena).
zakladni(zelena).
zakladni(modra).
slozena(X) :- \+ zakladni(X).
pekna(zluta).
pekna(zelena).
```

Zpoza negace nelze vrátit substituci:

```
?- slozena(X).
false.
```

```
?- slozena(X).
      |
?- \+ zakladni(X).
      |
?- zakladni(X), !, fail.
      | X = cervena
?- !, fail.
      |
?- fail.
      |
    fail
```

Kombinace negace a proměnných:

```
?- pekna(X), slozena(X).
X = zluta.
```

```
?- slozena(X), pekna(X).
false.
```

Příklad 12.2.1: Definujte predikát `nd35/1`, který je pravdivý, pokud jako parametr dostane číslo, které není beze zbytku dělitelné čísly 3 ani 5. Úlohu vyřešte bez použití řezu a negace, s použitím řezu a s použitím negace.

Predikáty pro všechna řešení

- `findall(+Template, :Goal, -List)`: všechna řešení volání `Goal` unifikovaná s `Template`
  - volné proměnné v `Goal` existenčně kvantifikovány (hodnoty se ignorují)
  - při žádném řešení uspěje s `List = []`
- `bagof(+Template, :Goal, -List)`: všechna řešení rozlišena podle volných proměnných v `Goal`
  - při žádném řešení selže
  - existenční kvantifikace: `bagof(X, Y ^ foo(X, Y), List)`
- `setof(+Template, :Goal, -List)`: jako `bagof`, ale odstraňuje duplicity

Predikáty pro všechna řešení

Příklad 12.3.2: Uvažme následující databázi faktů:

```
f(a, b).
f(a, c).
f(a, d).
f(e, c).
f(g, h).
f(g, b).
f(i, a).
```

Bez použití interpretru zjistěte, s jakou substitucí uspějí následující dotazy:

a) `?- findall(X, f(a, X), List).`
b) `?- findall(X, f(X, b), List).`
c) `?- findall(X, f(X, Y), List).`
d) `?- bagof(X, f(X, Y), List).`
e) `?- setof(X, Y ^ f(X, Y), List).`

Predikáty pro všechna řešení

Příklad 12.3.3: Napište predikát `subsets/2`, který pro danou množinu vygeneruje všechny její podmnožiny. Množinou rozumíme seznam, ve kterém se neopakují prvky. Na pořadí vygenerovaných množin nezáleží. Napište i predikát `isset/1`, který uspěje, když zadaný seznam korektně reprezentuje množinu (tedy neobsahuje duplicitní prvky).

Vstup a výstup

Založen na proudech, obvykle jeden proud pro vstup a jeden pro výstup

- `see(+File)`/`tell(+File)` otevře soubor `File` pro čtení/zápis, nastaví ho jako aktuální vstupní/výstupní proud
- `seeing(-Stream)`/`telling(-Stream)` uloží aktuální vstupní/výstupní proud
- `seen`/`told` zavře vstupní/výstupní proud
- `read(-Term)` čte term ze vstupu (ukončený tečkou)
  - provádí unifikaci (`read(foo(X, Y))`)
  - na konci vrací `end_of_file`
- `write(+Term)` zapíše term do aktuálního proudu

Vstup a výstup

Predikát neomezeného backtrackování:

```
repeat.
repeat :- repeat.
```

Typické čtení souboru:

```
readFile(File) :- seeing(Old), seen, see(File),
    repeat,
        read(X),
        process(X),
        % selhani spusti backtrackovani k repeat:
        X == end_of_file,
    % ukonci dalsi backtrackovani:
    !,
    seen, see(Old).
```

Vstup a výstup

Příklad 12.4.1: Napište predikát `fileSum(+FileName, -Sum)`, který bude číst zadaný soubor po termech a spočítá sumu čísel v termech tvaru `s(X)`. Ostatní termy bude ignorovat.

Programování s omezujícími podmínkami, opakování

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2014

Programování s omezujícími podmínkami (CLP)

Použitelné pro řešení některých problémů s implicitním zadáním:

- proměnné
- domény proměnných
- (omezující) podmínky určující vztahy proměnných

Omezíme se na celočíselné domény – modul `clpfd`:
`use_module(library(clpfd))`

Nové prvky jazyka:

- aritmetické výrazy, relační operátory: `#<, #=, ...`
- logické operátory: `0, 1, #\, #\/, #==>, ...`
- doménové predikáty: `in/2, ins/2, all_different/1`, domény: např. `1..10`
- řešící predikáty: `label/1, labeling/2`

Magický čtverec

2	7	6
9	5	1
4	3	8


```
:- use_module(library(clpfd)).

magic_square(X) :-
    X = [A,B,C,D,E,F,G,H,I],
    X ins 1..9, all_different(X), N = 15,
    A+B+C #= N, D+E+F #= N, G+H+I #= N,
    A+D+G #= N, B+E+H #= N, C+F+I #= N,
    A+E+I #= N, C+E+G #= N,
    label(X).
```

Faktoriál

Příklad 13.1.4: Napište predikát `fact/2`, který počítá faktoriál pomocí CLP. V čem je lepší než klasicky implementovaný faktoriál?

Problém mincí

Příklad 13.1.5: S využitím knihovny `clpfd` implementujte program, který spočítá, kolika a jakými mincemi lze vyskládat zadanou částku. Snažte se, aby řešení s menším počtem mincí byla preferována (nalezena dříve).

Problém osmi dam

Příklad 13.1.6: Uvažte problém osmi dam. Cílem je umístit na šachovnici 8×8 osm dam tak, aby se žádné dvě neohrožovali. Dámy se ohrožují, pokud jsou ve stejném řádku, sloupci nebo na stejné diagonále.

S pomocí knihovny `clpfd` napište predikát `queens/3`, který dostane v prvním argumentu rozměr šachovnice, ve druhém výstupním argumentu bude jako výsledek seznam s pozicemi sloupců, do kterých třeba v jednotlivých řádcích umístit dámy, a ve třetím argumentu bude možné ovlivnit způsob hledání hodnot (predikát `labeling/2`).

Příklad výstupu:

```
?- queens(8, L, [up]).
L = [1,5,8,6,3,7,2,4]
...
```

Příklad 13.1.7: Napište program, který nalezne řešení zmenšené verze Sudoku. Hrací pole má rozměry 4×4 a je rozdělené na 4 čtverce 2×2. V každém řádku, sloupci a čtverci se musí každé z čísel 1 až 4 nacházet právě jednou. Jako rozšíření můžete přidat formátovaný výpis nalezeného řešení.

Sudoku

Příklad 13.1.8: Stáhněte si program *sudoku.pl*. Soubor naleznete v ISu a v příloze sbírky.

a) Program spusťte a naučte se jej ovládat. Zamyslete se, jak byste funkcionalitu programu sami implementovali.
b) Prohlédněte si zdrojový kód programu a pochopte, jak funguje.
c) Modifikujte program tak, aby nalezená řešení splňovala podmínku, že na všech políčkách hlavní diagonály se vyskytuje pouze jedna hodnota.

Opakování

Příklad 13.2.1: Určete, které dotazy uspějí:

a) `?- X = 1.`
b) `?- X == 1.`
c) `?- X =:= 1.`
d) `?- X is 1 + 1.`
e) `?- X = 1, X == 1.`
f) `?- X = 1 + 1, X == 2.`
g) `?- X = 1 + 1, X =:= 2.`
h) `?- g(X, z(X)) = g(Y, Z).`
i) `?- a(a, a) = X(a, a).`
j) `?- a(1, 2) = b(1, 2).`

Příklad 13.2.2: Opravte chyby v následujícím programu a vylepšete jeho nevhodné chování (bez použití CLP). Měl by fungovat správně pro celá čísla a můžete předpokládat, že první argument je vždy plně instanciován.

```
fact(N, Fact) :-
    M #= N - 1, fact(M, FactP), Fact #= N * FactP.
fact(0, 1).
```

Opakování

Příklad 13.2.3: Napište predikát `merge/3`, který spojí 2 vzestupně uspořádané seznamy čísel z prvního a druhého argumentu. Výsledný vzestupně uspořádaný seznam pak unifikuje do třetího argumentu. Například dotaz `merge([1,2,4], [0,3,5], X)` uspěje se substitucí `X = [0,1,2,3,4,5]`.
Poté naprogramujte predikát `mergesort/2`, který seřadí seznam čísel z prvního argumentu podle velikosti s využitím techniky *merge sort* a výsledek unifikuje s druhým argumentem. Může se vám hodit i pomocný predikát `split/3`, který rozdělí zadaný seznam na 2 seznamy stejné délky.

Opakování

Příklad 13.2.6: Napište predikát `equals/2`, který uspěje, pokud se dva zadané seznamy rovnají. V opačném případě vypíše důvod, proč je tomu tak (jeden seznam je kratší, výpis prvků, které se nerovnají, …). Můžete předpokládat, že seznamy neobsahují proměnné. Příklady použití najdete níže.

```
?- equals([5,3,7], [5,3,7]).
true.
?- equals([5,3,7], [5,3,7,2]).
1st list is shorter
false.
?- equals([5,3,7], [5,2,3,7]).
3 does not equal 2
false.
```