IB015 Úvod do funkcionálního programování

A zase ta REKURZE ...

Jiří Barnat
Libor Škarvada

Jiný pohled na rekurzivní funkce

Rekurze připomenutí

Rekurze

- Definice funkce, nebo datové struktury, s využitím sebe sama.

Příklad

- Funkce `length` , která při aplikaci na seznam vrací jeho délku, je definovaná rekurzivně:

```
length :: [a] -> Integer
length [] = 0
length (_:s) = 1 + length s
```

Rekurze a zacyklení výpočtu

Zacyklení výpočtu

- Ne každé použití definovaného objektu na pravé straně definice je smysluplné.
- Nesprávné použití může vést k nekonečnému vyhodnocování, které nemá žádný efekt – **výpočet cyklí**.

Příklad

- Nesprávné použití rekurze ve funkci `length'` :

```
length' :: [a] -> Integer
length' [] = 0
length' x = length' x
```

- Při aplikaci `length'` na neprázdný seznam výpočet cyklí.

Kruh versus spirála

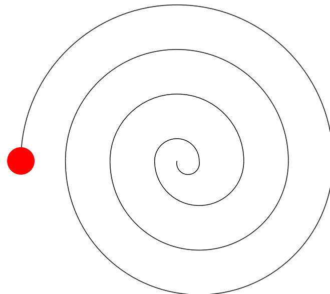
Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu.**
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále"a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

length [4,3,2,1]

```
    = 1 + length [3,2,1]
    = 1 + 1 + length [2,1]
    = 1 + 1 + 1 + length [1]
    = 1 + 1 + 1 + 1 + length []
    = 1 + 1 + 1 + 1 + 0
    = 4
```



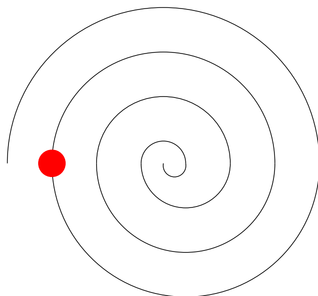
Kruh versus spirála

Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu.**
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále"a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
   = 1 + length [3,2,1]
   = 1 + 1 + length [2,1]
   = 1 + 1 + 1 + length [1]
   = 1 + 1 + 1 + 1 + length []
   = 1 + 1 + 1 + 1 + 0
   = 4
```

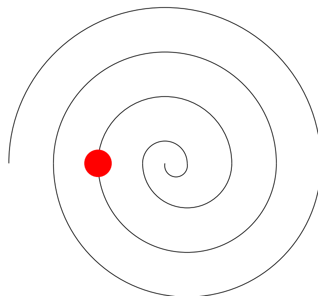


Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu.**
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
   = 1 + length [3,2,1]
   = 1 + 1 + length [2,1]
   = 1 + 1 + 1 + length [1]
   = 1 + 1 + 1 + 1 + length []
   = 1 + 1 + 1 + 1 + 0
   = 4
```



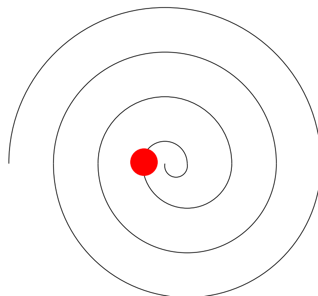
Kruh versus spirála

Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu.**
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále"a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
   = 1 + length [3,2,1]
   = 1 + 1 + length [2,1]
   = 1 + 1 + 1 + length [1]
   = 1 + 1 + 1 + 1 + length []
   = 1 + 1 + 1 + 1 + 0
   = 4
```



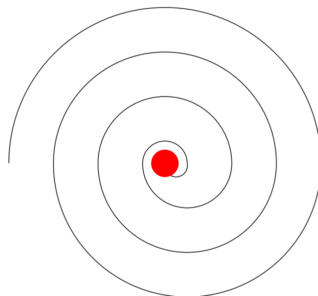
Kruh versus spirála

Pozorování

- Rekurze se někdy představuje jako definice kruhem. Lépe je však představit si rekurzi jako **spirálu.**
- Při výpočtu rekurzivního výrazu, který necyklí, se výpočet pohybuje "po spirále" a **nevyhnutelně** spěje k jejímu konci.

Demonstrace

```
length [4,3,2,1]
   = 1 + length [3,2,1]
   = 1 + 1 + length [2,1]
   = 1 + 1 + 1 + length [1]
   = 1 + 1 + 1 + 1 + length []
   = 1 + 1 + 1 + 1 + 0
   = 4
```



Rekurzivní definice funkcí

Pozorování

- Uvědomění si toho, co udává vzdálenost od středu pomyslné spirály, je klíč k správnému použití rekurze.

Rekurze ve funkci `length`

- Vzdálenost od středu odpovídá délce zbývající části seznamu.
- S každým dalším rekurzivním voláním funkce se seznam, který je argumentem funkce, zkracuje.
- Funkce `length` je tedy jednou nevyhnutelně volána pro prázdný seznam, což je volání, které rekurzi zastaví.

Definice rekurzivní funkce

2 části definice

- Při definice rekurzivní funkce je nutné si uvědomit, co je **středem spirály**, tj. kde se má výpočet rekurzivní funkce zastavit, a **jak se k tomuto středu bude výpočet blížit**.

Příklad – 2 části definice funkce `length`

- Ukončení rekurzivního výpočtu (střed spirály)

```
length [] = 0
```

- Jedno rekurzivní volání (přiblížení se o "jednu otáčku")

```
length (x:s) = 1 + length s
```

Příklad – 2 části definice v jednom výrazu

- Obě části v jednom řádku definice

```
f1 x = if (odd x) then x else f1 (x/2)
```

Rekurzivní funkce a větvení

- V případě, že se výpočet funkce větví, vzdálenost od středu pomyslné spirály musí klesat s každou větví.

Funkce s nekonečnou rekurzí

- Teoreticky je možné použít rekurzi pro realizaci nekonečného cyklu. V praxi však toto řešení nefunguje vzhledem k omezené velikosti paměti, pro uchovávání návratových adres.

Vzdálenost od středu

- To, že pomyslná vzdálenost od středu klesá, nemusí nutně znamenat, že datová struktura, se kterou rekurzivní funkce pracuje, se zmenšuje.

Příklad

- Je-li cílem algoritmu opakovaným dělením celku dosáhnout určitého počtu dílků, počet dílků při každém dělení roste.
- Vzdálenost od středu pomyslné spirály lze v tomto případě identifikovat jako počet dělení, které zbývá k dosažení cílového počtu.
- Všimněme si, že pokud se při každém kroku zdvojnásobí počet dílků, jejich počet roste vzhledem k počtu rekurzivních kroků exponenciálně.

Pozadí rekurze

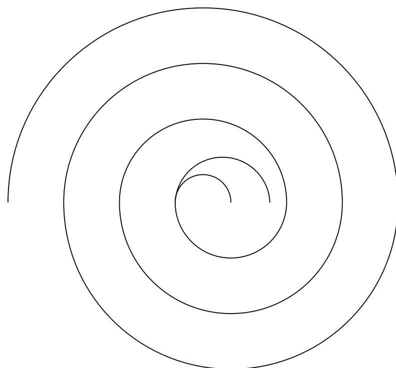
- Struktura, podle níž se řídí rekurze, nemusí být spojena s úplným uspořádáním.
- Musí však být **dobře založená** (well-founded), což znamená, že v ní neexistuje nekonečně dlouhá klesající posloupnost prvků.

Příklad

- Množina všech podmnožin dané množiny je pouze částečně uspořádána vzhledem k inkluzi, avšak postupné odebírání prvků z libovolné podmnožiny nevyhnutelně dospěje k prázdné množině.

Rozeklaná spirála

- Spirála je v místě dosažení středu rozeklaná, tj. končí ve dvou a více bázových případech.



Rozeklaná spirála

- Spirála je v místě dosažení středu rozeklaná, tj. končí ve dvou a více bázových případech.

Příklad rekurzivní funkce s více bázovými případy

Příklad

- Definujte funkci, která pro zadaný seznam vrátí seznam, který vznikne z původního seznamu vynecháním všech prvků na sudých pozicích.
- `oddMembers [1,2,3,4,5,6,7,8]` $\leadsto^*$ `[1,3,5,7]`
- `oddMembers "Trol ej ej schomoula."` $\leadsto^*$ `"To je cool."`

Myšlenka a definice

- Rekurzivní volání zkracuje zadaný seznam vždy o dva prvky.
- Krajními případy jsou **prázdný** a **jednoprvkový** seznam.
-
```
oddMembers :: [a] -> [a]
oddMembers [] = []
oddMembers (x:[]) = [x]
oddMembers (x:y:s) = x : oddMembers s
```

Rekurzivní datové struktury

Stejné principy rekurze

Pozorování

- Pro definici rekurzivních datových struktur (hodnot rekurzivních typů) platí podobná pravidla jako pro definice rekurzivních funkcí.

Opačný směr

- Vytváření hodnot rekurzivního datového typu probíhá od středu pomyslné spirály směrem ven.
- Rekurzivní datová struktura má **základní** (bázovou) **hodnotu**.
- Základní hodnota je rozvíjena **rekurzivním pravidlem**.

Seznam

Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

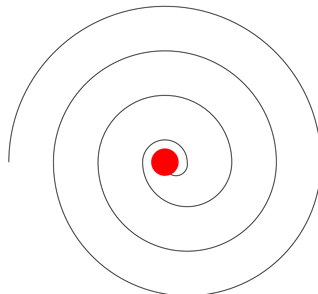
Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `( a : `**seznam**` )` je **seznam**.

Demonstrace

[]

```
[1] = 1 : []
[2,1] = 2 : [1]
[3,2,1] = 3 : [2,1]
[4,3,2,1] = 4 : [3,2,1]
```



Seznam

Klasický pohled na seznam

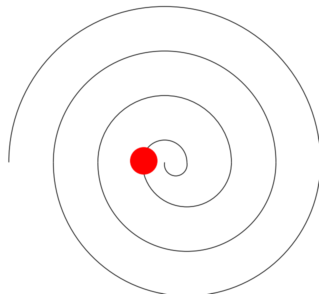
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `( a :` **seznam** `)` je **seznam**.

Demonstrace

```
[]
[1] = 1 : []
[2,1] = 2 : [1]
[3,2,1] = 3 : [2,1]
[4,3,2,1] = 4 : [3,2,1]
```



Seznam

Klasický pohled na seznam

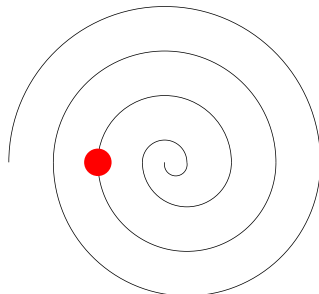
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `( a : ` **seznam** `)` je **seznam**.

Demonstrace

```
[]
[1] = 1 : []
[2,1] = 2 : [1]
[3,2,1] = 3 : [2,1]
[4,3,2,1] = 4 : [3,2,1]
```



Seznam

Klasický pohled na seznam

- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `( a :` **seznam** `)` je **seznam**.

Demonstrace

```
[]
[1] = 1 : []
[2,1] = 2 : [1]
[3,2,1] = 3 : [2,1]
[4,3,2,1] = 4 : [3,2,1]
```

Seznam

Klasický pohled na seznam

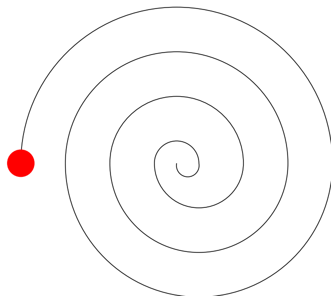
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

Rekurzivní pohled na seznam

- `[]` je **seznam**.
- `( a :` **seznam** `)` je **seznam**.

Demonstrace

```
[]
[1] = 1 : []
[2,1] = 2 : [1]
[3,2,1] = 3 : [2,1]
[4,3,2,1] = 4 : [3,2,1]
```



Rekurzivní datové struktury

Pozorování

- Rekurzivní nahlížení na seznam se může jevit jen jako mentální hříčka.

Stromy jako rekurzivní datové struktury

- Mnoho problémů je přirozené řešit s využitím jiné rekurzivně definované datové struktury – **binárního stromu**.
- Nelineární rekurzivní datová struktura.

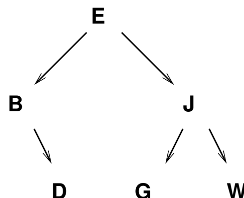
Binární stromy

Rekurzivní definice binárního stromu

- Prázdný strom je **binární strom**
- Hodnota a k ní asociovaný levý a pravý **binární strom** je **binární strom**

Příklad

- Graficky zadaný binární strom.
- E označujeme jako **kořen** stromu
- E, B a J jsou **vnitřní vrcholy** stromu
- D, G a W označujeme jako **listy**
- Levý a pravý binární strom asociovaný s danou hodnotou označujeme jako levý a pravý **podstrom**.



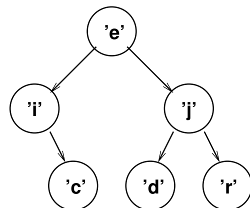
- Binární stromy asociované k hodnotám D, G, W a levý podstrom asociovaný s hodnotou B jsou prázdné stromy.

Definice datového typu `BinTree a`

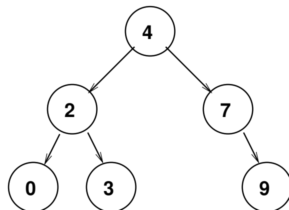
```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

Příklady hodnot definovaného typu

```
tc :: BinTree Char
tc = Node 'e'
       (Node 'i' Empty (Node 'c' Empty Empty))
       (Node 'j' (Node 'd' Empty Empty)
                 (Node 'r' Empty Empty))
```




```
tn :: BinTree Int
tn = Node 4
       (Node 2 (Node 0 Empty Empty)
               (Node 3 Empty Empty))
       (Node 7 Empty (Node 9 Empty Empty))
```



Funkce pro práci s rekurzivními datovými strukturami

Problém

- Chceme definovat funkci, která při aplikaci na hodnotu typu `BinTree Int` zvýší o jedna všechny hodnoty uložené v uzlech stromu.

Jak takovou funkci definovat?

- Výčtem hodnot nelze – možných hodnot je nekonečně mnoho.

```
treeP1‘ :: Num a => BinTree a -> BinTree a
treeP1‘ Empty = Empty
treeP1‘ (Node x) Empty Empty = Node (x+1) Empty Empty
⋮
```

- **Rekurzivně**, rekurzi vedeme podle struktury stromu

```
treeP1 :: Num a => BinTree a -> BinTree a
treeP1 Empty = Empty
treeP1 (Node x left right)
       = Node (x+1) (treeP1 left) (treeP1 right)
```

Popis funkce `treezipwith`

- Funkce `treezipwith` pomocí binární operace `op` vytvoří ze dvou stromů nový strom, jehož struktura bude průnikem obou stromů a v jehož uzlech budou výsledky aplikace operace `op` na hodnoty uzlů ze stejné pozice v obou stromech.

Definice funkce `treezipwith`

- ```
  treezipwith :: (a -> b -> c) -> BinTree a -> BinTree b -> BinTree c
  treezipwith op (Node v1 l1 r1) (Node v2 l2 r2)
              = Node (v1 `op` v2) (treezipwith op l1 l2)
                                  (treezipwith op r1 r2)
  treezipwith _ _ _ = Empty
  ```

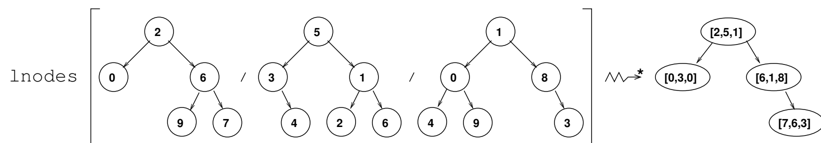
Příklad

`treezipwith (+)`
  ```

Popis funkce `lnodes`

- Funkce `lnodes` vytvoří ze seznamu stromů jeden strom seznamů, tj. strom, jehož struktura bude průnikem všech stromů ze seznamu a v jehož uzlech budou seznamy hodnot z uzlů na odpovídajících pozicích.

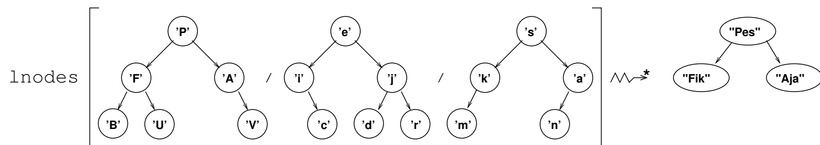
Příklad



Definice funkce `lnodes`

```
lnodes :: [ BinTree a ] -> BinTree [a]
lnodes = foldr (treezipwith (:)) niltree
   where niltree = Node [] niltree niltree
```

Příklad



Dokazování rekurzivních programů

Dokazování správnosti programů

Fakta

- Ověřování správnosti navržených algoritmů je součást práce programátora.
- Testování je nedokonalé.
- Správnost algoritmu můžeme prokázat například tím, že ji formálně (= s matematickou přesností) dokážeme.

Důkaz koreknosti algoritmu

- Dokazujeme, že pokud výpočet algoritmu na platných vstupech skončí, tak algoritmus vrací korektní výsledek. O algoritmu, který má tuto vlastnost říkáme, že je **částečně správný**.
- Pokud je algoritmus částečně správný a dokážeme, že na platných vstupech svůj výpočet vždy skončí, pak říkáme, že algoritmus je **úplně správný**.

Rekurze a matematická indukce

Pozorování

- Pro důkazy částečné správnosti i terminace rekurzivních funkcí se používá **matematická indukce**.

Matematická indukce

- Matematická indukce je metoda dokazování tvrzení, která se používá, pokud chceme ukázat, že dané tvrzení platí pro všechny prvky dobře založené rekurzivně definované nekonečné posloupnosti. (Jako jsou například přirozená čísla.)

Princip matematické indukce

- Ukážeme platnost tvrzení pro bázovou hodnotu.
- Ukážeme, že tvrzení se přenáší při aplikaci rekurzivního kroku.

$$\underline{\mathbf{T(0)}} \text{ a } \underline{\mathbf{T(i)} \Rightarrow \mathbf{T(i+1)}} \implies \mathrm{T}(0), \mathrm{T}(1), \mathrm{T}(2), \ldots$$

Příklad

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurze a matematická indukce spolu úzce souvisí.

Příklad

- Dokažte, že pro každá dvě přirozená čísla x a y taková, že $x > 0$ platí, že funkce `fpow` aplikovaná na argumenty x a y vrátí hodnotu x^y.
-
```
fpow :: Integer -> Integer -> Integer
fpow x 0 = 1
fpow x y = x * fpow x (y-1)
```

Příklad

Pozorování

- Klíčovým problémem při použití matematické indukce je identifikace toho, podle čeho má být indukce vedena.
- Napovědět může místo rekurzivního volání funkce, neboť rekurze a matematická indukce spolu úzce souvisí.

Příklad

- Dokažte, že pro každá dvě přirozená čísla x a y taková, že $x > 0$ platí, že funkce `fpow` aplikovaná na argumenty x a y vrátí hodnotu x^y.
-
```
fpow :: Integer -> Integer -> Integer
fpow x 0 = 1
fpow x y = x * fpow x (y-1)
```
- **Důkaz povedeme indukcí vzhledem k hodnotě y.**

Příklad – Bázový krok

Bázový krok, T(0)

- Nechť $y=0$, a nechť x je libovolné.
- `fpow x y` se redukuje dle `fpow x 0 = 1` na hodnotu `1` .
- $x^0 = 1$, pro libovolné x.
- Tudíž pro $y = 0$ tvrzení platí.

Příklad – Indukční krok

Indukční krok, T(i)=>T(i+1)

- Dokazujeme, že pokud tvrzení platí pro hodnotu i, pak tvrzení platí i pro hodnotu $i+1$. Platnost tvrzení pro hodnotu i se označuje jako **indukční předpoklad**.
- Platnost tvrzení pro hodnotu i říká, že `fpow x i` $\leadsto^* x^i$ pro libovolnou hodnotu x.
- `fpow x (i+1)`

  $\leadsto$ `x * fpow x (i+1-1)`

  $\leadsto$ `x * fpow x i` $\overset{dleIP}{=}$ `x *` x^i

  $\leadsto$ x^{i+1}
- Ukázali jsme, že pokud tvrzení platí pro i, pak platí i pro $i+1$.
- Z platnoti bázového kroku a vlastností matematické indukce plyne, že pro libovolnou hodnotu x tvrzení platí pro všechny hodnoty y.

Další příklady

Věta 1

- Jsou-li s, t dva konečné seznamy stejného typu a délek, a označíme-li $m =$ `length` s, $n =$ `length` t, pak

```
length (s ++ t) = m + n.
```

- Důkaz veden indukcí podle délky seznamu s.

Věta 2

- Pro každé tři seznamy s, t, u platí rovnost

```
(s ++ t) ++ u = s ++ (t ++ u).
```

- Důkaz veden indukcí podle délky seznamu s.

Věta 3

- Pro každý seznam s a celé číslo $m \geq 0$ platí

```
take m s ++ drop m s = s.
```

- Důkaz veden indukcí podle m.

Mentální cvičení

Měřítko "vzdálenosti" rekurze

- Jaká vlastnost čísla `x` určuje hloubku rekurze při volání následující funkce?

```
f1 x = if (odd x) then x else f1 (x/2)
```

A to je konec ...

Závěrem

Co jsme se dozvěděli ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.

Čím ještě nám byl kurz prospěšný ...

- Správné funkcionální návyky pro programování použijeme v imperativních programovacích jazycích.
- Mentální posilovna.

Závěrem

Co jsme se dozvěděli ...

- Funkcionální výpočetní paradigma.
- Solidní základy programovacího jazyka Haskell.

Čím ještě nám byl kurz prospěšný ...

- Správné funkcionální návyky pro programování použijeme v imperativních programovacích jazycích.
- Mentální posilovna.

Kudy dál?

IB016 Seminář z funkcionálního programování

- Programování v Haskellu

IA014 Funkcionální programování

- Teoretické základy funkcionálního programování
- Implementace funkcionálních jazyků
- Vybrané techniky funkcionálního programování

Poděkování

Přednášejícího studentům

- Za vzornou docházku a přípravu jak na přednášky, tak i na vnitrosemestrální a zkouškové písemky a za celkově poctivý přístup ke studiu.

Studentů přednášejícímu

- Formou zpětné vazby například vyplněním studentské ankety a upozorněním na zásadní, ale i okrajové nedostatky jak přednášejícího, tak i jím připravených studijních materiálů.