IB015 Neimperativní programování

A zase ta REKURZE!
(rekurze na typech)

Jiří Barnat
Libor Škarvada

Uživatelem definované typy

(algebraické typy)

Riziko nepoužití uživatelem definovaných typů

Pozorování

- Počítač veškerá data reprezentuje čísly.
- Programátoři jej dobrovolně, či nedobrovolně napodobují.

Riziko

- Mnohdy číselná reprezentace různých hodnot není přímočará a tedy umožňuje nechtěné zadání neplatných hodnot.
- Neplatné hodnoty mohou vzniknout i neopatrnou aplikací číselných operací.
- **Použití neplatných hodnot může být nebezpečné.**

Riziko nepoužití uživatelem definovaných typů – příklad

Příklad

- Chceme reprezentovat den v týdnu a definovat funkce pracující s touto reprezentací.
- Možné číselné kódování, je následující:

  pondělí $= 1$, úterý $= 2$, ..., neděle $= 7$

- Funkce `zitra` (s chybou) a funkce `je_pondeli` :

```
zitra :: Int -> Int
zitra x = x+1                          - nesprávně i (x+1) `mod` 7

je_pondeli :: Int -> Bool
je_pondeli x = if (x==1) then True else False
```

Chyba ve výpočtu

`je_pondeli 8` $\rightsquigarrow$ `False`

`je_pondeli (zitra 7)` $\rightsquigarrow$ `...` $\rightsquigarrow$ `False`

Uživatelem definované typy

Definice typů

- V Haskellu pomocí klíčového slova `data` .
- Obecná šáblona:

  `data` **Název_typu** `=` **Hodnotové_konstruktory**
- Jednotlivé hodnotové konstruktory se oddělují znakem `|`
- Syntaktické omezení Haskellu: nově definovaný typ i hodnotové konstruktory musí začínat velkým písmenem.

Příklad

- Dny v týdnu lze definovat jako nový typ, který má 7 hodnot.

```
data Dny = Po | Ut | St | Ct | Pa | So | Ne
```

- Hodnoty jsou definovány výčtem.
- Jsou použity nulární hodnotové konstruktory – konstanty.

Definice n-árních hodnotových konstruktorů

Uživatelem definované

- Obecná šablona pro n-ární hodnotový konstruktor:

  Jméno Typ_1 ... Typ_n

- Příklad typu s ternárním hodnotovým konstruktorem:

  `data Barva =` **`RGB Int Int Int`**

- Hodnoty typu Barva:

```
RGB 42 42 42
RGB 12 (-23) 45
```

Částečná aplikace hodnotového konstruktoru

```
RGB          :: Int -> Int -> Int -> Barva
RGB 23       ::        Int -> Int -> Barva
RGB 23 23    ::               Int -> Barva
RGB 23 23 23 ::                      Barva
```

Typové konstruktory

Typové konstanty

- Definicí dle šablony:

  `data` **Název_typu** `= Hodnotové_konstruktory`

  zavádíme nový typ s označením `Název_typu`.
- `Název_typu` je nulární typový konstruktor, typová konstanta.

N-ární typové konstruktory

- Typové konstruktory jako například `->` nebo `[]` nedefinují typ, pouze předpis jak nový typ vyrobit.

Tvorba typu

- Každá typová konstanta definuje typ.
- Typ získám také úplnou aplikací n-árních typových konstruktorů na již definované typy.

  `(->) Dny Bool =` **Dny -> Bool**

  `[] Dny =` **[Dny]**

  `(->) (Dny -> Bool) [Dny] =` **(Dny -> Bool) -> [Dny]**

Hodnoty a typy

Tvorba nových hodnot

- Aplikace hodnotových konstruktorů vytváří nové hodnoty.

Tvorba nových typů

- Aplikace typových konstruktorů vytváří nové typy.

Uspořádané n-tice a seznamy

- Používá se stejné označení pro typové i hodnotové konstruktory!

```
'a' :: Char
[('a','a'),('a','a')] :: [(Char,Char)]          - hodnotové
[('a','a'), ('a','a')] :: [(Char,Char)]           - typové
```

Definice polymorfních typových konstruktorů

Polymorfní typové konstruktory

- Seznam prvků typu `a` , strom hodnot typu `a` , ...

Definice polymorfních typových konstruktorů

- Definice s využitím typových proměnných:

  `data` **Název_typu** $\mathbf{a_1}$ **...** $\mathbf{a_n}$ `= ...`

- Typové proměnné lze použít pro definici hodnotových konstruktorů.

Kompletní obecná šablona

$$\begin{aligned}\texttt{data Tcons } a_1 \texttt{ ... } a_n \texttt{ = } & \texttt{ Dcons}_1 \texttt{ typ}_{(1,1)} \texttt{ typ}_{(1,2)} \texttt{ ... typ}_{(1,arita_1)} \\ & \vdots \\ & \texttt{Dcons}_m \texttt{ typ}_{(m,1)} \texttt{ typ}_{(m,2)} \texttt{ ... typ}_{(m,arita_m)}\end{aligned}$$

Typový konstruktor `Maybe`

Maybe

- Předdefinovaný unární polymorfní typový konstruktor.

```
data Maybe a = Nothing | Just a
```

- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`

```
deleni :: Fractional a => a -> a -> Maybe a
deleni x y = if (y==0) then Nothing else Just (x/y)
```

- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?

Typový konstruktor `Maybe`

Maybe

- Předdefinovaný unární polymorfní typový konstruktor.

```
data Maybe a = Nothing | Just a
```

- Zamýšlené použití pro funkce, jejichž hodnota může být nedefinována.

Příklad

- Chceme ošetřit dělení nulou, definujeme novou funkci `deleni`

```
deleni :: Fractional a => a -> a -> Maybe a
deleni x y = if (y==0) then Nothing else Just (x/y)
```

- Jaký je výsledek aplikace `deleni` na argumenty `32` a `8` ?

  Just 4.0

- Proč je následující definice špatně?

```
deleni x y = if (y==0) then Nothing else (x/y)
```

Typové aliasy

type vs. data

- Zatímco s využitím `data` definuji nové typy, pomocí `type` zavádím typové aliasy (synonyma) k již existujícícm typům.
- Používá se pro lepší čitelnost kódu.

Příklady

- ```
  type String = [Char]
  ```
- ```
  type Day = Int
  type Month = Int
  type Year = Int
  type Date = (Day,Month,Year)
  ```

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.
- Chybu neodhalí typová kontrola, definice je typově správně.

Kruh versus spirála

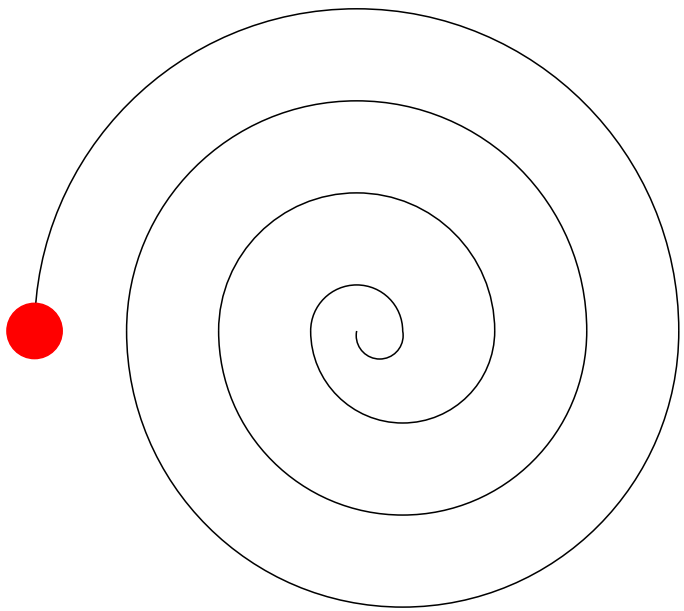
Pozorování

- 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
```



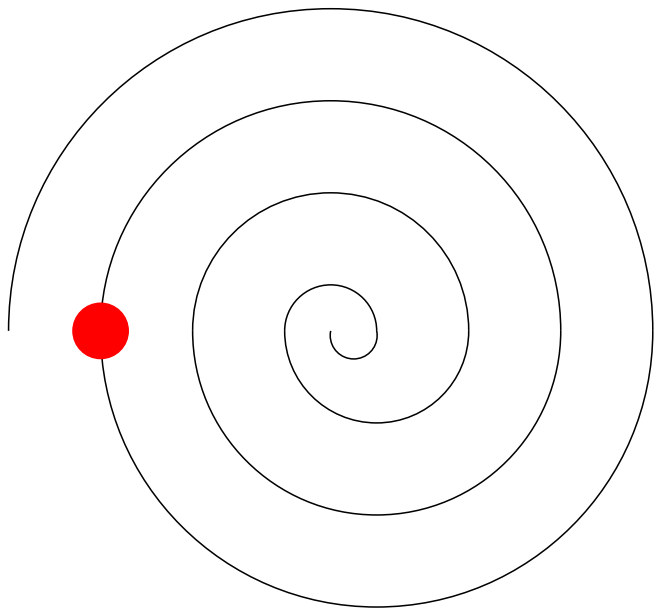
Kruh versus spirála

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
```



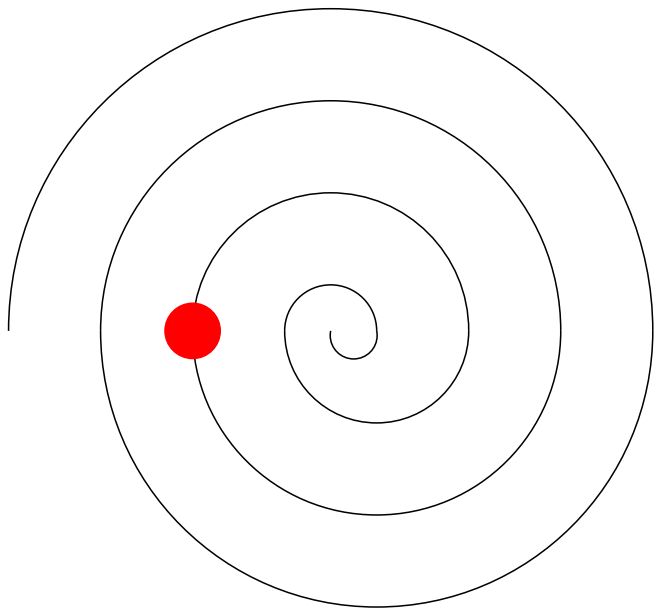
Kruh versus spirála

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
```



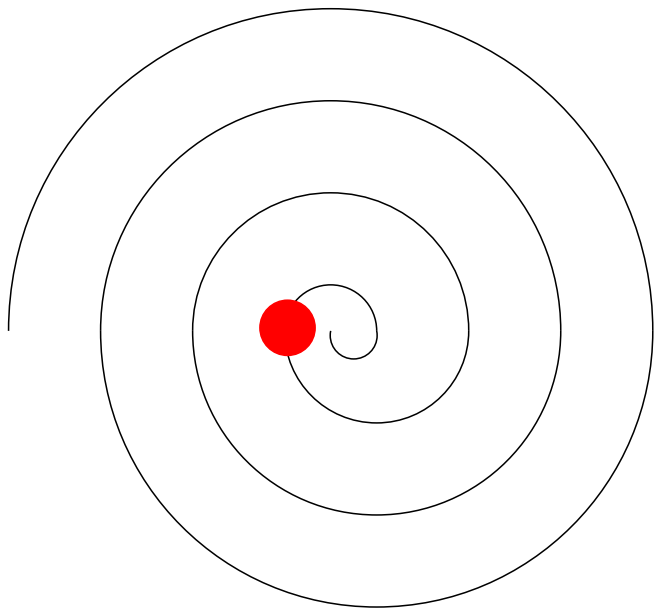
Kruh versus spirála

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
```



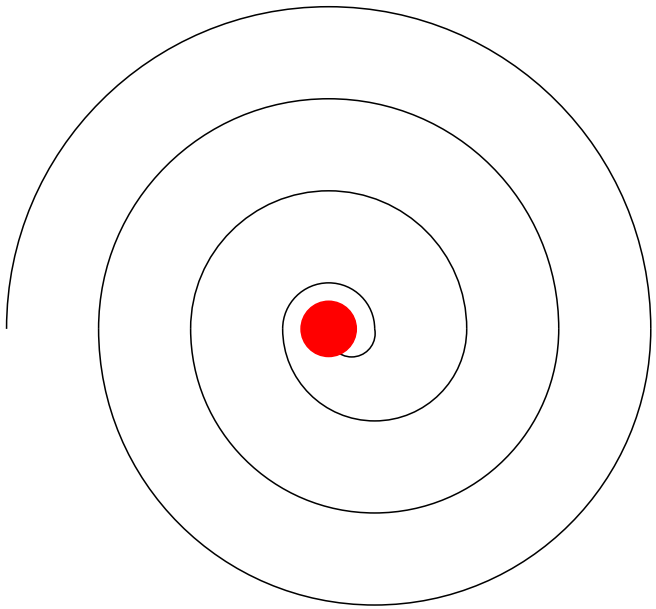
Kruh versus spirála

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
```



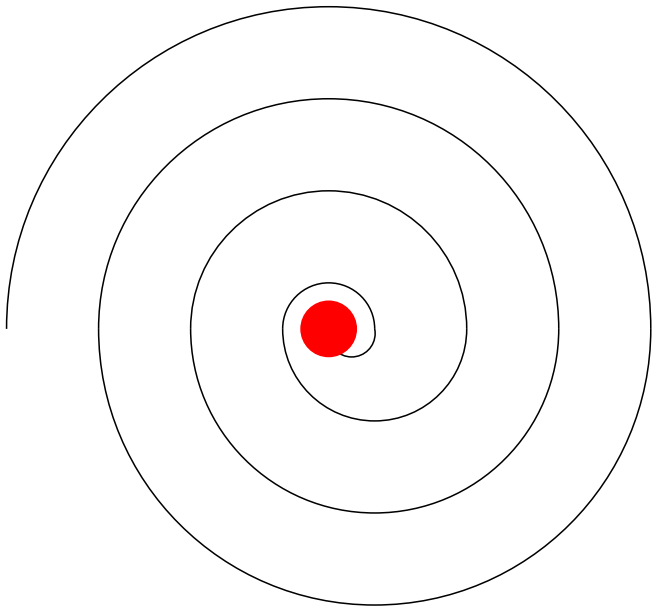
Kruh versus spirála

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
```



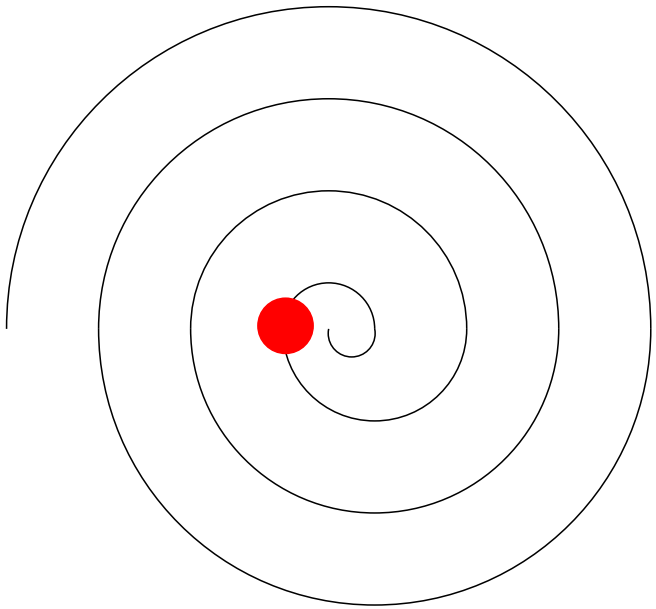
Kruh versus spirála

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
```



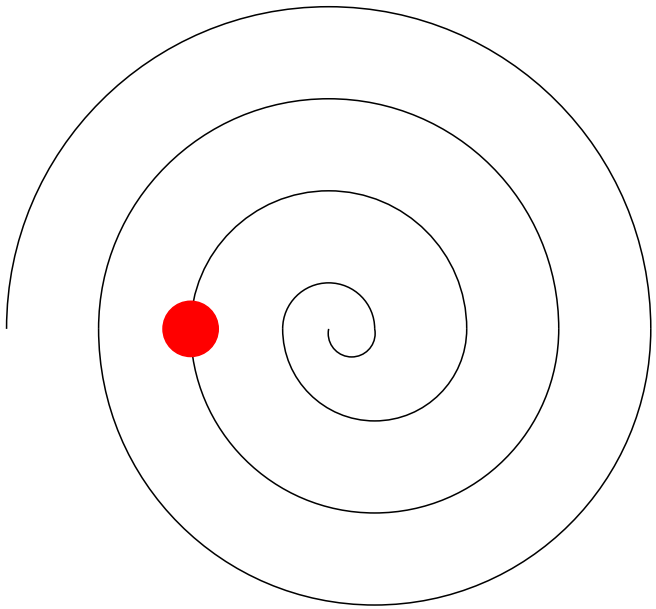
Kruh versus spirála

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
```



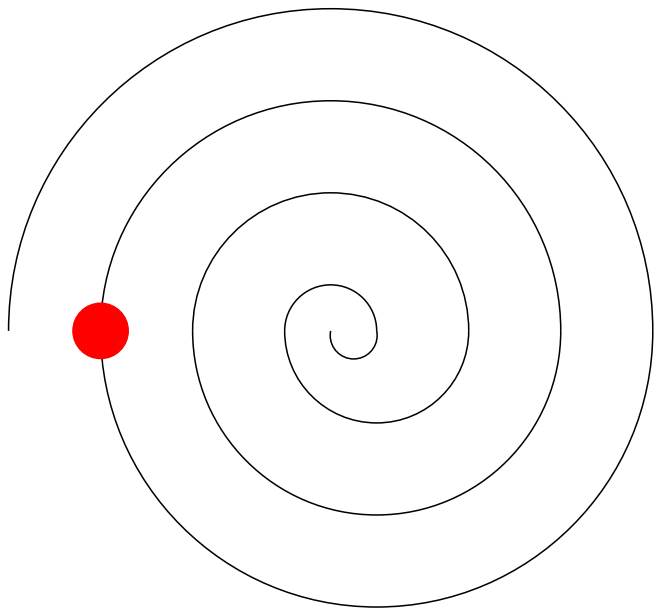
Kruh versus spirála

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
```



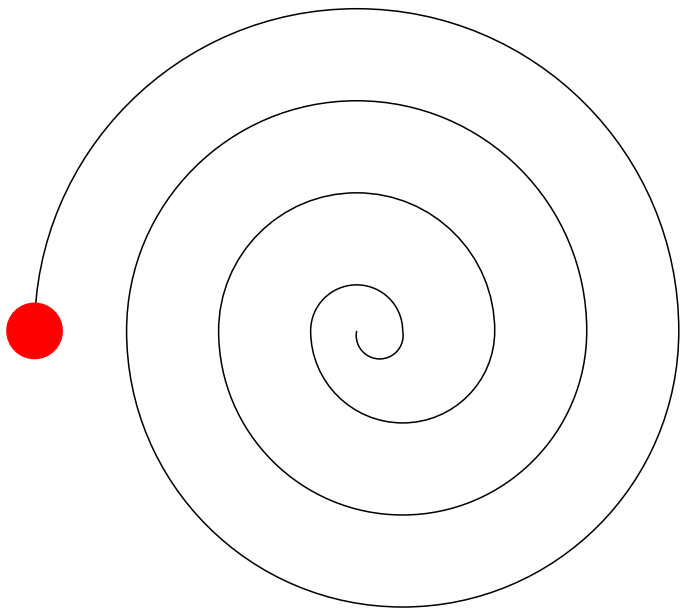
Kruh versus spirála

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 definici 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 :: Integer -> Integer
f1 x = if (odd x) then x else f1 (x `div` 2)
```

Rekurzivní funkce a větvení

- V případě, že se výpočet funkce větví, vzdálenost od středu pomyslné spirály musí klesat s každou větví.
- Musí existovat větev, která rekurzi ukončuje a je proveditelná, pokud jsme ve středu pomyslné spirály.
- ```
  f2 :: Integer -> Integer
  f2 x = if (x==0) then 0          - chyba, případ nemusí nastat
                   else if (odd x) then f2 (x-2)
                                   else f2 (x-1)
  ```

Funkce s nekonečnou rekurzí

- Teoreticky je možné použít rekurzi pro realizaci nekonečného cyklu. V praxi však toto řešení nemusí fungovat 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

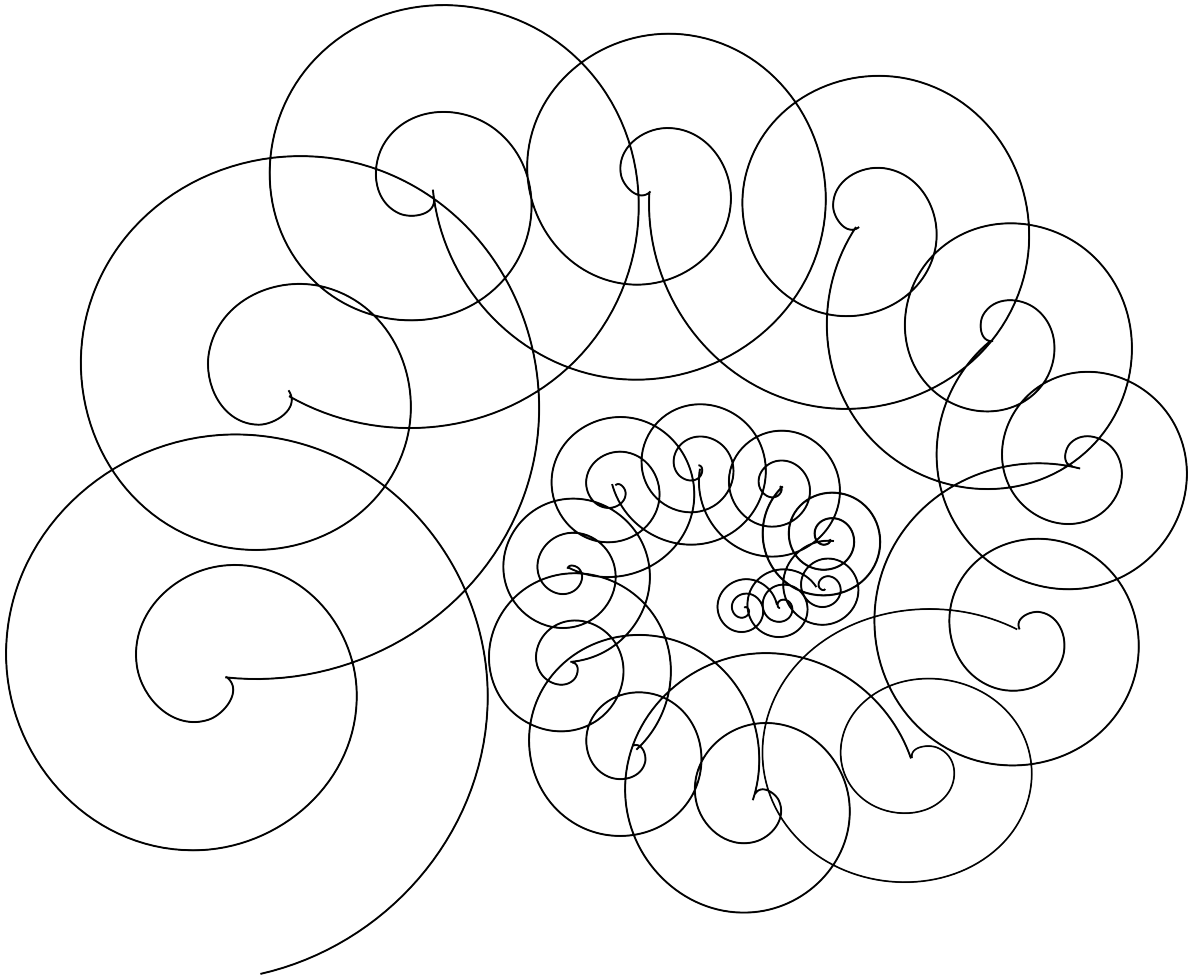
- Struktura, podle níž se řídí rekurze, nemusí být spojena s úplným uspořádáním.
- Musí však být **dobře založená** (well-founded), což znamená, že v ní neexistuje nekonečně dlouhá klesající posloupnost prvků.

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.

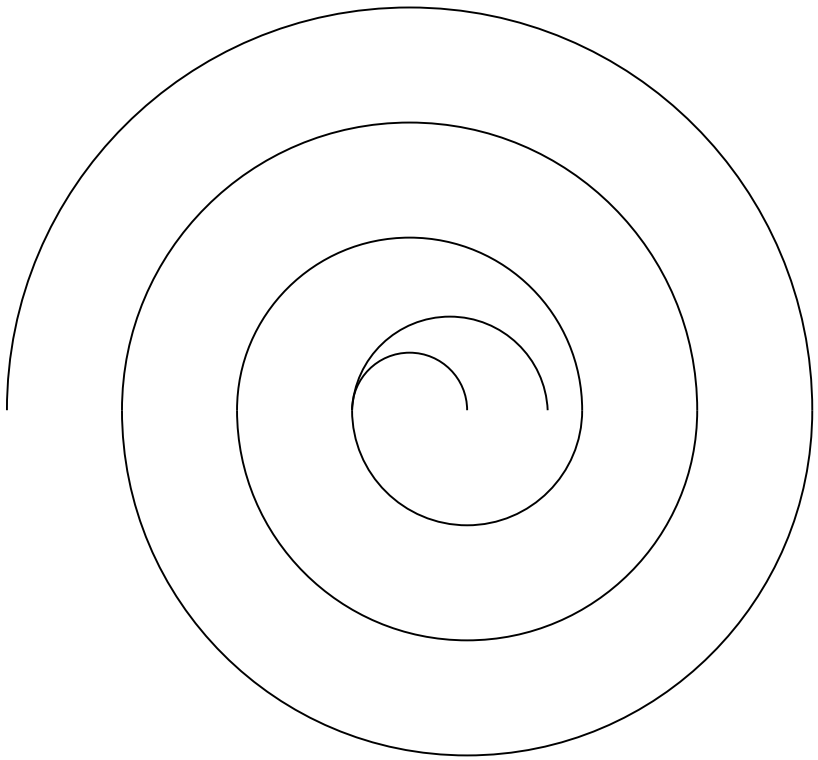
Vnořená rekurze

- Rekurzivně volané funkce se mohou vnořovat.
- "Spirála spirál".



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]` $\rightsquigarrow^*$ `[1,3,5,7]`
- `oddMembers "Trol ej ej schomoula."` $\rightsquigarrow^*$ `"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.

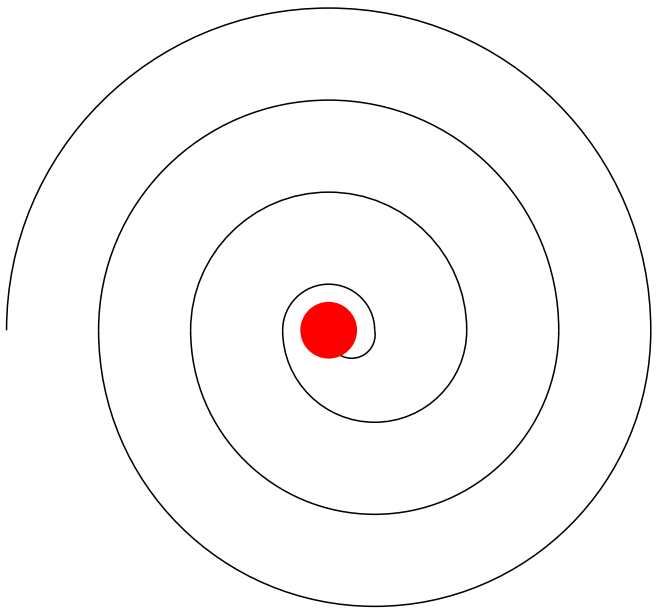
Rekurzivní pohled na seznam

- `[]` 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

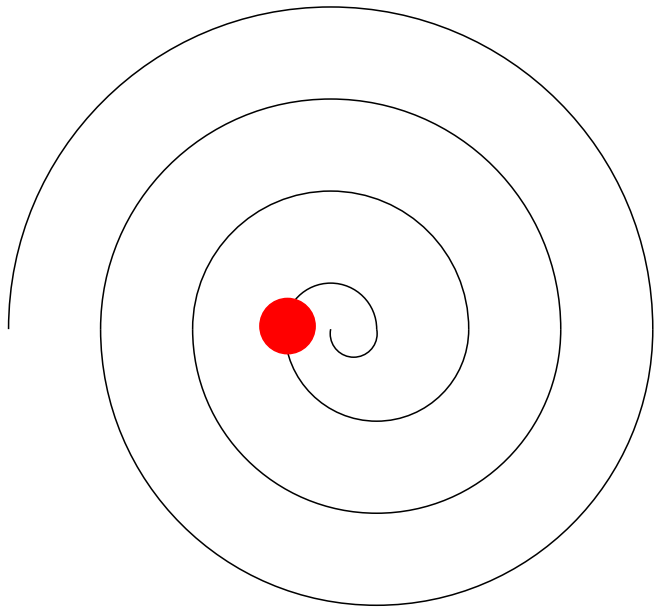
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

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

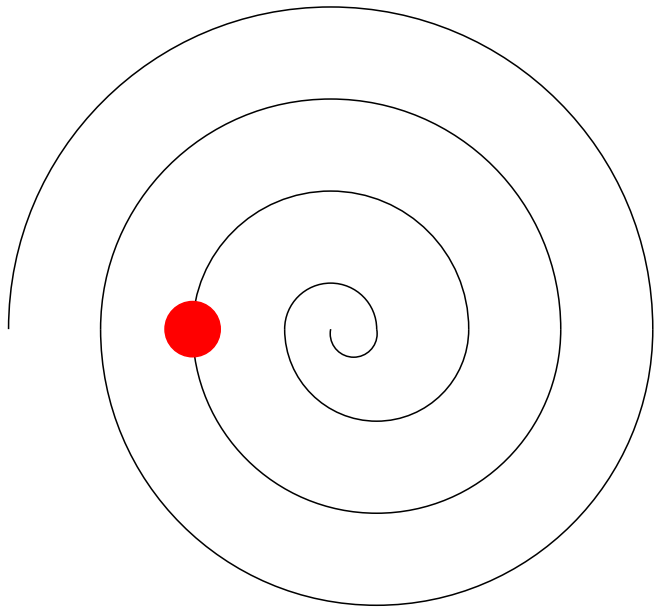
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

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

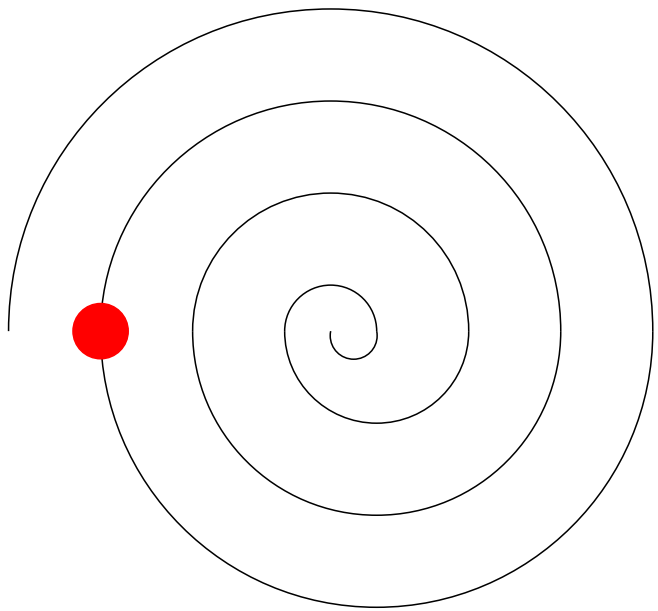
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

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

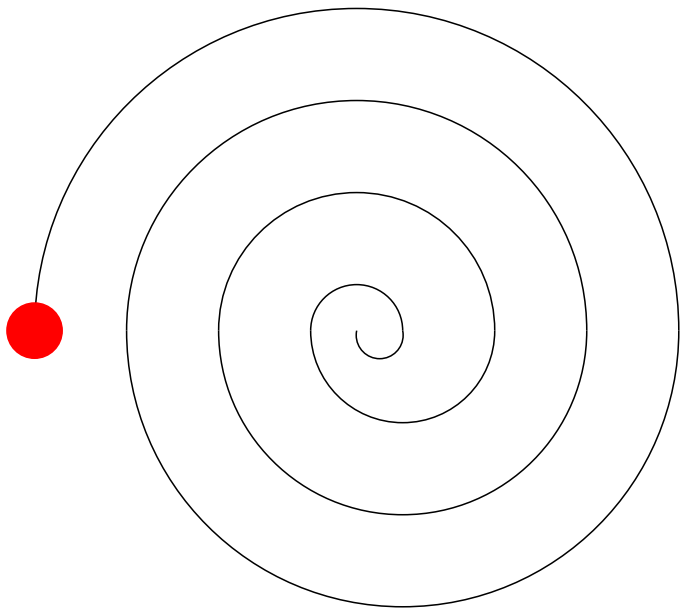
- Prázdná, konečná, případně nekonečná posloupnost prvků stejného typu.

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.

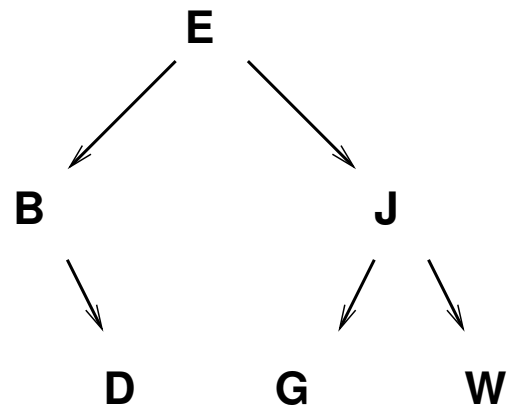
Binární stromy

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.

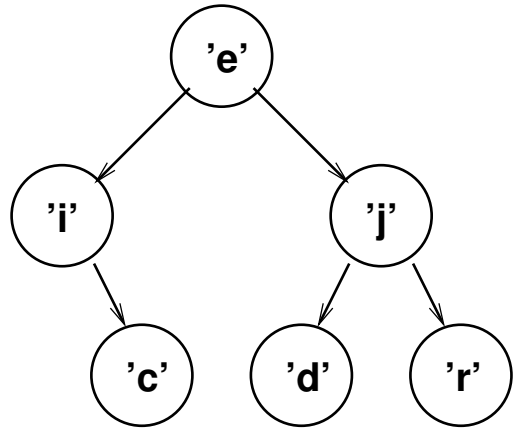
Binární stromy v Haskellu

Definice datového typu `BinTree a`

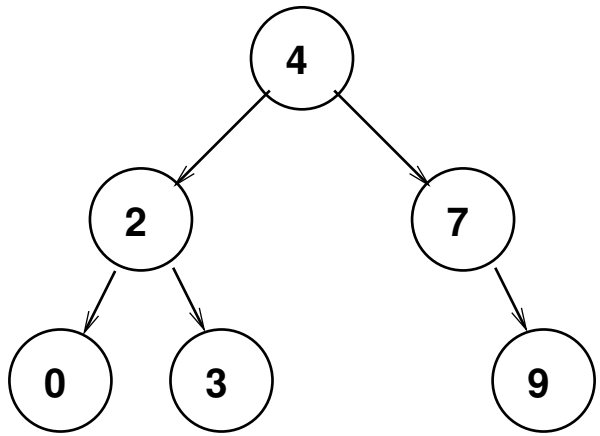
```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

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`

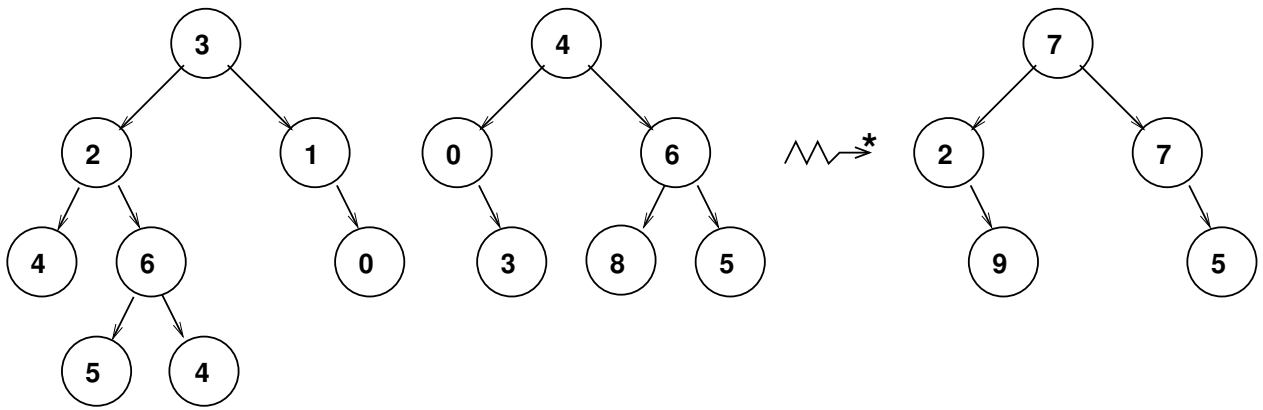
- Funkce `treezipwith` pomocí binární operace `op` vytvoří ze dvou stromů nový strom, jehož struktura bude průnikem obou stromů a v jehož uzlech budou výsledky aplikace operace `op` na hodnoty uzlů ze stejné pozice v obou stromech.

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 (+)`


  ```

Příklad: Stromová struktura bez fixní arity

Pozorování

- V následujícím typu, nemá rekurzivní sestup pevnou aritu.

Příklad

```
data Policie =  Hlidka (String,String) | Oddeleni [Policie]
                deriving Show

h1 = Hlidka ("Pepa", "Emil")
h2 = Hlidka ("Jason", "Drson")
o1 = Oddeleni [h1, h2]

jmena :: Policie -> [String]
jmena (Hlidka (a,b)) = a:b:[]
jmena (Oddeleni []) = []
jmena (Oddeleni (x:s)) = jmena x ++ jmena (Oddeleni s)
```

Dokazování rekurzivních programů

(pro chytré hlavičky)

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 korektnosti 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.

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

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` $\rightsquigarrow^* x^i$ pro libovolnou hodnotu x.
- `fpow x (i+1)`

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

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

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

Další příklady na matematickou indukci

Věta 1

- Jsou-li s, t dva konečné seznamy stejného typu a délek, pak

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

- 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.

Checkpoint

Měřítko „vzdálenosti“ rekurze

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

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

Ternární stromy

- Zadefinujte nový datový typ, který odpovídá ternárním stromům a naprogramujte funkci, která pro instanci takového stromu zjistí délku nejlevější větve.