

IB015 Neimperativní programování

Ukázky funkcionálně řešených problémů

Jiří Barnat
Libor Škarvada

Prohledávání a prořezávání

Problém n dam

Problém n dam

- Rozestavit n šachových dam na pole čtvercové šachovnice $n \times n$ tak, aby se žádné dvě dámy navzájem neohrožovaly.

Všechna řešení pro $n = 4$

- | | | | |
|---|---|---|---|
| | ♔ | | |
| | | | ♔ |
| ♔ | | | |
| | | ♔ | |

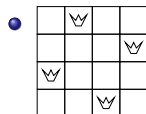
		♔	
♔			
			♔
	♔		

Pozorování

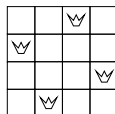
- V každém řádku/sloupku šachovnice může být nejvýše jedna dáma (jinak se ohrožují), a zároveň musí být alespoň jedna dáma, jinak bychom jich neumístili n .

Kódování pozice

- Očíslujeme 1 až n řádky šachovnici směrem shora dolů a sloupce šachovnice směrem zleva doprava.
- Řešení budeme kódovat seznamy čísel a to tak, že čísla v seznamu určují pozice dam v odpovídajících sloupcích.



[3, 1, 4, 2]



[2, 4, 1, 3]

Úkol

- Naprogramujte funkci `damy :: Int -> [[Int]]`, která pro zadané n vrátí seznam všech možných řešení.

Postupné rozšiřování šachovnice

- Zavedeme funkci $da\ m\ n :: Int \rightarrow Int \rightarrow [[Int]]$, která bude počítat všechna řešení pro obdélníkovou matici s m sloupky a n řádky ($m < n$).
- Nová funkce bude řešení počítat rekurzivně, a to s využitím řešení pro šachovnici s $(m - 1)$ sloupky a n řádky.
- Šachovnice s nula sloupky, má jedno triviální řešení: $[]$, tj. $da\ 0\ n = [[]]$.

Cesta k celkovému řešení

- $damy :: Int \rightarrow [[Int]]$
 $damy\ n = da\ n\ n$

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
-

1	2	...	m-1	
				1
				2
				3
				4
				⋮
				n

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- `hrozba :: [Int] -> [Int]`
`hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]`

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- `hrozba` vrátí seznam ohrožených pozic.

0	1	2	...	m-1	
					1
					2
					3
					4
					⋮
					n

Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- [4, , ,] [4, , ,] [4, , ,]

0	1	2	...	m-1	
					1
					2
					3
	♔				4
					⋮
					n

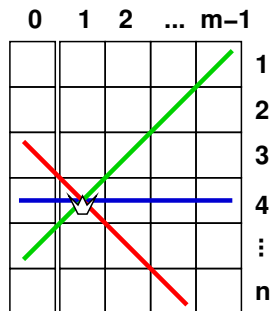
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- $[4, \ , \ , \]$ $[4, \ , \ , \]$ $[4, \ , \ , \]$
 $+ [1, 2, 3, 4]$ $- [1, 2, 3, 4]$

 $[4, \ , \ , \]$ $[5, \ , \ , \]$ $[3, \ , \ , \]$



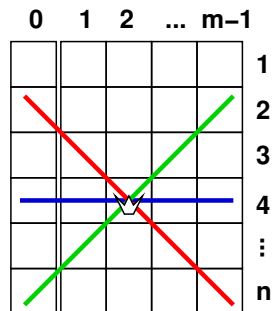
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- $[,4, ,] \quad [,4, ,] \quad [,4, ,]$
 $\quad \quad \quad +[1,2,3,4] \quad -[1,2,3,4]$

 $[,4, ,] \quad [,6, ,] \quad [,2, ,]$



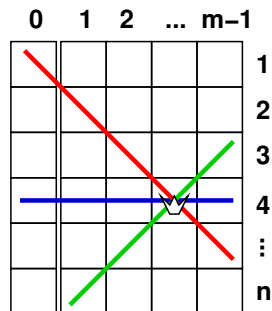
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupec jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- $[, ,4,]$ $[, ,4,]$ $[, ,4,]$
 $+ [1,2,3,4]$ $- [1,2,3,4]$

 $[, ,4,]$ $[, ,7,]$ $[, ,1,]$



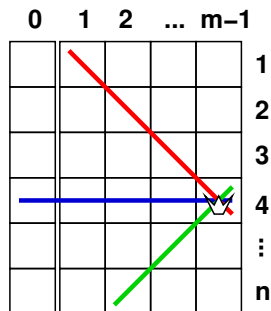
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- $[, , ,4] \quad [, , ,4] \quad [, , ,4]$
 $\quad \quad \quad +[1,2,3,4] \quad -[1,2,3,4]$

 $[, , ,4] \quad [, , ,8] \quad [, , ,0]$



Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupek jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupek.
- [6,1,3,5] [6,1,3,5] [6,1,3,5]

	0	1	2	...	m-1	
			♔			1
						2
				♔		3
						4
					♔	⋮
	♔					n

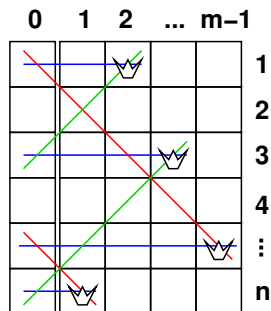
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupec jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupec.
- $[6,1,3,5]$ $[6,1,3,5]$ $[6,1,3,5]$
 $+ [1,2,3,4]$ $- [1,2,3,4]$

 $[6,1,3,5]$ $[-,3,6,-]$ $[5,-,-,1]$



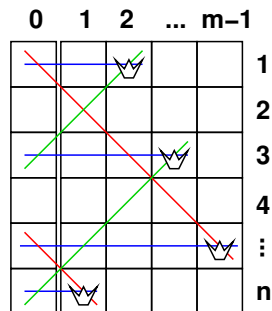
Účel

- Rozhodnout, které pozice při rozšíření o jeden sloupec jsou při stávajícím rozložení dam nevyhovující.
- hrozba :: [Int] -> [Int]
hrozba p = p ++ zipWith (+) p [1..] ++ zipWith (-) p [1..]

Princip

- Předpokládejme stávající šachovnici, kterou rozšíříme zleva o jeden sloupec.
- $[6,1,3,5]$ $[6,1,3,5]$ $[6,1,3,5]$
 $+ [1,2,3,4]$ $- [1,2,3,4]$

 $[6,1,3,5]$ $[-,3,6,-]$ $[5,-,-,1]$
- Platné volné pozice: 2 a 4.



```
type Reseni = [Int]

damy :: Int -> [Reseni]
damy n = da n n

da :: Int -> Int -> [Reseni]
da 0 _ = [[]]
da m n = [ k:p | p <- da (m-1) n, k <- [1..n], nh k p]
          where nh k p = k 'notElem' ( p
                                     ++ zipWith (+) p [1..]
                                     ++ zipWith (-) p [1..] )
```

Backtracking, Prořezávání

- Backtracking – rekurzivní generování všech možných řešení.
- Prořezávání – časná eliminace neplatných řešení (zde realizováno funkcí `nh k p`).

Nejmenší nepoužité přirozené číslo

(Richard Bird: Pearls of Functional Algorithm Design)

Zadání

- Pro konečný seznam přirozených čísel zjistěte nejmenší přirozené číslo, které se v seznamu nevyskytuje.

Řešení 1.

- Ze seznamu přirozených čísel "odečteme" zadaný seznam. Nejmenší číslo výsledného seznamu je hledané číslo.

```
minfree :: [Int] -> Int
minfree s = head ([0..] 'listminus' s)
```

- Pro odečtení jednoho seznamu od druhého si definujeme pomocnou funkci:

```
listminus :: Eq a => [a] -> [a] -> [a]
listminus u v = filter ('notElem' v) u
```

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` 1
 `v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` 2
- `v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` 3
- `v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` `length v`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` `length v+1`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` `length v+2`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` `length v+3`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce listminus

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` `length v + length v-1`
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce `listminus`

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]` ...
`v = [9,8,7,6,5,4,3,2,1,0]`

Funkce `listminus`

- `listminus u v = filter ('notElem' v) u`
- `u = [0,1,2,3,4,5,6,7,8,9,10,...]`
`v = [9,8,7,6,5,4,3,2,1,0]`

Vlastnosti řešení

- Délka výpočtu: $\text{length } v + \text{length } v-1 + \text{length } v-2 + \text{length } v-3 + \dots$
- V nejhorším případě kvadratická časová složitost.

Otázka

- Je možné problém řešit s lineární časovou složitostí?

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True` .
- Pro každé číslo v v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False` , poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[True ,True ,True ,True ,True]`
`[0,3,1,4]`

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True`.
- Pro každé číslo v v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False`, poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- [**False**, True, True, True, True]
[**0**, 3, 1, 4]

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True` .
- Pro každé číslo v v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False` , poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- [`False`, `True` , `True` , **`False`**, `True`]
[0, **3**, 1, 4]

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True` .
- Pro každé číslo v v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False` , poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- [`False`, **`False`**, `True` , `False`, `True`]
[0, 3, **1**, 4]

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True` .
- Pro každé číslo v v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False` , poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False, False, True , False, False]`
`[0, 3, 1, 4]`

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True` .
- Pro každé číslo v v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False` , poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False, False, True, False, False]`
`[0, 3, 1, 4]`

Pozorování

- Nejmenší číslo bude vždy v rozsahu $\langle 0, \text{length}(v) \rangle$.

Idea lepšího řešení

- Uvažme tabulku, ve které je $(\text{length } v + 1)$ hodnot `True` .
- Pro každé číslo v zadaném seznamu přepíšeme na pozici určené tímto číslem hodnotu `True` na `False` , poté pozice prvního nepřepsaného `True` určuje hledané číslo.
- `[False, False, True, False, False]`
`[0, 3, 1, 4]`

Předpoklady

- Konstantní operace pro přístup k hodnotám omezeně dlouhého seznamu.

Haskell efektivně, aneb od seznamů k polím

Pole

- Seznam/tabulka adresovatelných míst pro uložení dat.
- Klíčovou vlastností je časově konstantní operace pro přístup k hodnotě na zadané adrese.
- Důležitá datová struktura v imperativním světě.

Poznámka:

- Přístup k n -tému prvku seznamu je možné realizovat funkcí
 - $(!!) :: [a] \rightarrow \text{Int} \rightarrow a$
 - $(!!) (x:s) 0 = x$
 - $(!!) (_ :s) n = (!!)\ s (n-1)$
- Časová složitost operace $(!!)$ ale není konstantní.

Použití

- Pole jsou definována v modulu `Data.Array`.
- `import Data.Array`

Typový konstruktor `Array`

- Binární typový konstruktor: `Array :: *->*->*`
- `Array I A` je typ všech polí, která jsou indexovaná (adresovaná) hodnotami typu `I` a obsahují prvky typu `A`.
- Typ použitý pro indexaci musí být instancí typové třídy `Ix`.

Příklad hodnoty a typu

- `array (1,4) [(1,'a'),(2,'b'),(3,'a'),(4,'b')]`
`:: (Num i, Ix i) => Array i Char`

Typová třída `Ix`

- Instance typové třídy `Ix` umožňují efektivní implementaci pole.
- ```
class (Ord a) => Ix a where
 range :: (a,a) -> [a]
 index :: (a,a) -> a -> Int
 inRange :: (a,a) -> a -> Bool
```

## Meze pole

- Uspořádaná dvojice indexů tvoří meze pole.
- Všechny hodnoty uvnitř mezí pole jsou platné pro indexaci.
- `range` : Seznam platných hodnot daného rozsahu.
- `inRange` : Test zda je daný index v uvedených mezích.
- `index` : Pozice daného indexu v rozsahu uvedených mezí.

## Instance třídy `Ix`

- Předdefinovanými instancemi třídy `Ix` jsou typy `Int`, `Integer`, `Char`, `Bool` a jejich kartézské součiny (se sebou samým).

## Příklad 1

- `array ('D', 'F') [( 'D', 2014), ('E', 1977), ('F', 2001)]`  
:: `Num e => Array Char e`

## Příklad 2

- `range (5,9) ~\~* [5,6,7,8,9]`
- `range ('a', 'd') ~\~* "abcd"`
- `range ((1,1), (3,3)) ~\~*`  
`[(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)]`

## Přístup k prvkům pole

- Uvažme pole `arr :: Array I A` a index do pole `i :: I`, pak `arr!i` je prvek uložený v poli `arr` pod adresou `i`.
- `(!) :: Ix i => Array i e -> i -> e`
- Například:

```
array (5,6) [(5,"ANO!"),(6,"NE!")] ! 5 ~>* "ANO!"
array ('D','E') [('D',2014),('E',1977)] ! 'D' ~>* 2014
```

## Aktualizace hodnot v poli

- Operace `//` zkopíruje původního pole a při kopírování aktualizuje uvedený seznam dvojic (index, hodnota).
- `(//) :: Ix i => Array i e -> [(i, e)] -> Array i e`
- Například:

```
array (1,2) [(1,1),(2,2)] // [(1,3),(2,3)]
~>* array (1,2) [(1,3),(2,3)]
```

## Meze pole

- `bounds :: Ix i => Array i e -> (i,i)`

## Seznam indexů pole

- `indices :: Ix i => Array i e -> [i]`

## Převody mezi poli a seznamy

- `elems :: Ix i => Array i e -> [e]`
- `listArray :: Ix i => (i,i) -> [e] -> Array i e`

## Převody mezi poli a seznamy dvojic

- `assocs :: Ix i => Array i e -> [(i,e)]`
- `array :: Ix i => (i,i) -> [(i,e)] -> Array i e`

## accumArray

- Knihovná funkce Haskellu.
- Vytváří pole ze seznamu dvojic (index, hodnota) a navíc akumuluje (funkcí foldl) prvky seznamu se stejným indexem.
- $\text{accumArray} :: \text{Ix } i \Rightarrow (\text{e} \rightarrow \text{a} \rightarrow \text{e}) \rightarrow \text{e} \rightarrow (i, i) \rightarrow [(i, \text{a})] \rightarrow \text{Array } i \text{ e}$
- Je-li akumulární funkce konstantní, pracuje v lineárním čase.

## Příklady

- $\text{accumArray } (+) \ 0 \ (1,2) \ [(1,1), (1,1), (1,1)]$   
 $\rightsquigarrow^* \text{array } (1,2) \ [(1,3), (2,0)]$
- $\text{accumArray } (\text{flip}(:)) \ [] \ (1,2) \ [(2, 'e'), (1, 'l'), (1, 'B'), (2, 'b')]$   
 $\rightsquigarrow^* \text{array } (1,2) \ [(1, "Bl"), (2, "be")]$

## Nejmenší nepoužité přirozené číslo II

(Richard Bird: Pearls of Functional Algorithm Design)

## Postup

- `minfree s ∈ ⟨0, length(s)⟩`  
`s = [1,2,6,2,0]`
- Nebudeme uvažovat čísla mimo očekávaný rozsah.  
`(filter (<=n) s) where n = length s ∼* [1,2,2,0]`
- Vytvoříme asociační seznam (příprava na konverzi do pole).  
`t s = (zip (filter (<=n) s) (repeat True))`  
`where n = length s`  
`t s ∼* [(1,True),(2,True),(2,True),(0,True)]`
- Vytvoříme pole a odstraníme duplicity tak, aby nepoužité indexy ukazovaly na `False`.  
`checklist :: [Int] -> Array Int Bool`  
`checklist s = accumArray (||) False (0,length s) (t s)`

## Mezivýsledek

- `checklist s ∼*`  
`array (0,5) [(0,True),(1,True),(2,True),(3,False),(4,False),(5,False)]`



## Postup pokračování

- V seznamu typu `Array Int Bool` najdeme nejmenší index odkazující na `False`.

```
search :: Array Int Bool -> Int
```

- Nejprve převedeme pole na seznam hodnot typu `Bool`

```
elems (checklist s)
```

```
↪* [True,True,True,False,False,False]
```

- Ze seznamu ponecháme pouze jeho prefix s hodnotami `True`

```
(takeWhile id . elems) (checklist s)
```

```
↪* [True,True,True]
```

- Index prvního `False` určíme jako délku tohoto prefixu.

```
search = length . takeWhile id . elems
```

```
search (checklist [1,2,6,2,0]) ↪* 3
```

## Řešení

- `minfree = search . checklist`

```
search :: Array Int Bool -> Int
```

```
search = length.takeWhile id . elems
```

```
checklist :: [Int] -> Array Int Bool
```

```
checklist s = accumArray (||) False (0,n)
```

```
 (zip (filter (<=n) s) (repeat True))
```

```
 where n = length s
```

## Složitost algoritmu

- `checklist` , `search` – lineární
- `minfree` – lineární

## Lineární řazení některých seznamů

(Richard Bird: Pearls of Functional Algorithm Design)

## Pozorování

- Jsou-li čísla v seznamu z omezeného rozsahu, je možné tento seznam setřídít v **lineárním čase**.

## Algoritmus

- `max = 1000`

```
countlist :: [Int] -> Array Int Int
countlist s = accumArray (+) 0 (0,max) (zip s (repeat 1))

sortBoundedList s =
 concat [replicate k x | (x,k) <- assocs (countlist s)]
```

## Příklad

- `sortBoundedList [3,3,4,1,0,3] ~>* [0,1,3,3,3,4]`

