

## Kvantitatiivne tüübiteooria

Probleem 1: Listid, mis tunnevad oma ülemist raja

Definitsioon üsna lihtne:

```
data BList : Nat → Type where
  Nil : BList 0
  (::) : (n:Nat) → {m:Nat} → BList m → BList (maks n m)

testblist : BList 3
testblist = [1,2,3,2,2,3]
```

Funktsioonid keerukamad, kui listidel, aga tehtavad.

```
Main> makeBList 10
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

... aga arvutamise kiirus väga aeglane

```
Main> :exec time $ sumBList $ revBList (makeBList 2000)
duration: 6.724873000s
Main> :exec time $ sumList $ revList (makeList 2000)
duration: 0.009214000s
```

Lahendus 1: märgendame väärtused, mida vaja vaid tüübikontrolliks.

```
data BList0 : Nat → Type where
  Nil : BList0 0
  (::) : (n:Nat) → {0 m:Nat} → BList0 m → BList0 (maks n m)

testblist0 : BList0 3
testblist0 = [1,2,3,2,2,3]
```

Kompileerimisel asendatakse märgendatud parameetrid ()-ga.

... nüüd arvutamise kiirus väga kiire

```
Main> :exec time $ sumBList $ revBList (makeBList 2000)
duration: 6.724873000s
```

```
Main> :exec time $ sumList $ revList (makeList 2000)
duration: 0.009214000s
```

```
Main> :exec time $ sumBList0 $ revBList0 (makeBList0 2000)
duration: 0.007729000s
```

## Tehniliselt: Null

- Null arvuga muutujate ei tohi/saa "kasutada".
- S.t. seda saab kasutada vaid argumendis null arvuga parameetritele.
- Null arvuga parameetrid kustutatakse kompileerimisel.

Näide:

```
test0 : (0 _ : Int) → Int
test0 x =
  let 0 y : Int = x + 10 in
  100
```

NB! Implitsiitsed (tüübi)parameetrid on vaikimisi arvuga null.  
Näiteks a tüübis  $a \rightarrow a$ .

## Motivatsioon

Probleem 2:

Pilditöötlusprogramm, mis rakendab pildile (baitide massiiv) mingit filtrit.

- Puhtal FP-1 on eeliseid (andmete sõltuvused selged) aga peame muutmisel tegema koopiaid (aeglane, kasutab liigselt mälu).
- Lahendus 1: Modelleerime programmi puhaste arvutuste ja massiivimuudatuste jadana. (sarnaselt IO-le)
  - Puhta FP eelised kaovad ära.
- Lahendus 2: Kitsendame keelt nii, et "vana" väärtust ei tohi kasutada ja kopeerimise asemel hoopis muudame väärtust.

Tehniliselt: märgendame muutujad, mida võib vaid üks kord kasutada.

## Tehniliselt: Üks

- Üks arvuga muutujat peab täpselt üks kord kasutama: tagastama või andma parameetriks.
- S.t. lisaks ühele kasutuskorrale suvaline arv null-kasutuskordi.
- Üks arvuga argumendi puhul on ka tagastusväärtus üks-arvuga. Muidu saaks kaudselt ikka parameetrit mitu korda kasutada.

Näide:

```
test1 : (1 _: Int) → Int
test1 x =
  let 0 y : Int = test0 x + 10 in
  let 1 z : Int = test1 x in -- lõpmatu tsükkel
  let 0 y : Int = z in
  z
```

## Kvantitatiivne tüübiteooria

- Kolm arvu: *null*, *üks* ja *suva*.
- Arve saab kirjutada let-i muutuja ja funktsiooni tüübi parameetri ette.  
Näiteks  $(1\ x : \text{Int}) \rightarrow \text{Bool}$ .
- let-i muutuja arvu püüab Idris tuletada.
- Vaikimisi on arvuks *suva*.
- Arvud on ainult vahend. API looja vastutab turvalisuse eest.

## Tehniliselt: Suva

- Suva arvuga muutujat saab suvaline arv kordi kasutada.
- S.h. null või üks parameetri argumendina.
- Suva arvuga argumentide puhul on ka tagastusväärus suva-arvuga.

Näide:

```
tests : Int → Int
tests x =
  let y : Int = test0 x in
  let z : Int = test1 x in
  z + z
```

## Lineaarsus

Kirjanduses kasutatakse *lineaarsuse* mõistet.

Meie kontekstis:

- $x$ -ga tehakse kõigepealt teisendus  $f$ , siis  $g$  ja siis  $h$ . Ehk  $h(g(f(x)))$ .
- Välistab, et väärtust  $x$  ja  $f(x)$  tulemust peaks korraga mälus hoidma.



## Turvalisus

- Märgendi 1 lisamine ei taga, et seda parameetrit ka sisuliselt täpselt üks kord kasutatakse.
- Mustrisobitus on üks "kasutus", ehk kui andmestruktuuri sees pole lineaarset resurssi, siis saame lineaarsust rikkuda.

```
testb : (1 _ : List Int) → (List Int, List Int)
testb x =
  case x of
    []   ⇒ ([], [])
    x::xs ⇒ (x::xs, x::xs)
```

- API looja ülesanne on, et lineaarset resurssi saaks turvaliselt kasutada.
- Enamasti on arvuga üks:
  - abstraktsed tüübid või liidesed,
  - spetsiaalsed lineaarsed konteinerid.

## Näide: linearsed massiivid

```
infix 5 #

public export
data L : Type → Type → Type where
  (#) : (l _ : a) → b → L a b

export
interface Array arr where
  write : (l a : arr) → Nat → Double → arr
  read  : (l a : arr) → Nat → L arr Double
  size  : (l a : arr) → L arr Nat
  withArray : Nat → ((l a : arr) → L arr c) → c

export
Array (List Double) where
  ...

export
Array LinArray where      -- LinArray "private"
  ...
```

## Näide: lineaarsed massiivid

```

mapArrayHelper : Array arr ⇒ (Double → Double) → Nat → Nat → (1 _ : arr) → arr
mapArrayHelper f i len a =
  if i==len then
    a
  else
    let a # ov = read a i
        a      = write a i (f ov)
    in
      mapArrayHelper f (i+1) len a

mapArray : Array arr ⇒ (Double → Double) → (1 _ : arr) → arr
mapArray f a =
  let a # len = size a
  in
    mapArrayHelper f 0 len a

```

## Näide: testimiseks listid

**export**

```
Array (List Double) where
  write [] n d = []
  write (x :: xs) 0 d = d :: xs
  write (x :: xs) (S k) d = x :: write xs k d

  read [] n = [] # 0.0
  read (x :: xs) 0 = (x::xs) # x
  read (x :: xs) (S n) =
    let _ # r = read xs n in
      x::xs # r

  size [] = [] # 0
  size (x :: xs) = (x :: xs) # (length (x :: xs))

  withArray l f =
    let _ # r = f (replicate l 0.0) in
      r
```

