Taust: Curry-Howard vastavus

- C.-H.: loogikad ja tüübisüsteemid on samasuguse struktuuriga.

- Nägime: saame teadmisi ja oskusi üle kanda.
- Nägime: saame kasutada samu vahendeid.

- Kordamine: mida arvutab järgneva väite tõestus?

$$\forall x, y \in \mathbb{N}.\ x > y \to \exists z \in \mathbb{N}.\ x = y + z$$

- Täna:
  - ülekandmine teises suunas;
  - lõpetame range tõestuse ja programmeerimise eristamise;
  - ehk, vaatame sõltuvate tüüpidega programmeerimist.

Probleem

Tahame listi indekseerida funktsiooniga: `Nat → List a → a`

Probleem:

- List ei pruugi sisaldada `n`-ndat elementi!
- List võib olla täiesti tühi! Siis ei saa isegi valet elementi tagastada.

Võime teha nii: `Nat → List a → Maybe a`.

Töötab, kuid pole optimaalne juhul, kui listis on piisavalt elemente.

Listi tüüp võiks sisaldada infot tema pikkuse kohta.

Pikkusega listid

- Tavalised listid (uue süntaksi abil)

```
data List : Type → Type where
    Nil : List a
    (::) : a → List a → List a
```

- Paneme pikkused külge

```
data Vec : Nat → Type → Type where
    Nil : Vec 0 a
    (::) : a → Vec n a → Vec (1+n) a

test5 : Vec 3 Int
test5 = [1,2,3]
```

- selekteerimine

```
nth : (n:Nat) → Vec (1+n+m) a → a
nth 0     (x :: xs) = x
nth (S k) (x :: xs) = nth k xs
```

- konkateneerimine

```
concatVec : Vec n a → Vec m a → Vec (n+m) a
concatVec []       ys = ys
concatVec (x :: y) ys = x :: concatVec y ys
```

- Vektorite map

```
Functor (Vec n) where
 -- map : (a → b) → Vec n a → Vec n b
    map f []       = []
    map f (x :: y) = f x :: map f y
```

NB! Definitsioon sama mis listidel!

Funktsiooni `nth n` tüüp `Vec (1+n+m) a → a` sõltub `n`-i väärtusest.

- `nth 0 : Vec (1+m) a → a`
- `nth 2 : Vec (3+m) a → a`

Sõltuvad tüübid võimaldavad arvutamist ja selle tõestamist koos teha.

Aga see ei tööta alati nii lihtsalt ...

- Ümberpööramine listidel

```
revList : List a → List a
revList [] = []
revList (x :: ys) = (revList ys) ++ [x]
```

- Saame tõestada lihtsa teoreemi:

```
revrev :  (xs:List a) → revList (revList xs) = xs
```

- Ümberpööramine vektoritel

```
idVec : {n:Nat} → Vec (n + 1) a → Vec (1 + n) a
idVec {n = 0}     xs       = xs
idVec {n = (S k)} (x :: ys) = x :: idVec ys

revVec : {n:Nat} → Vec n a → Vec n a
revVec {n = 0  } []       = []
revVec {n = S k} (x :: ys) =
    idVec ((revVec ys) `concatVec` [x])
```

- Ilma `idVec`-ta tuleb veateade. Idris ei tea, et `k + 1` on `S k` ehk `1+k`.
- `idVec` asemel võib kasutada **`rewrite`** ... **`in`** ... väite `1+k = k+1` tõestusega.

Mitte ainult Vec!

- `Vec` kasutatakse palju aga selles pole midagi sisuliselt erilist.
- Näiteks, saame teha ülemise rajaga listi.

```
maks : Nat → Nat → Nat
maks 0 m = m
maks n 0 = n
maks (S k) (S j) = S (maks k j)

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

- Funkstiooniga saame ka teha tüüpe, mis sõltuvad väärtusest:

```
Arv : Bool → Type
Arv False = List Int -- False: mittedeterministlik
Arv True  = Int      -- True: deterministlik
```

- ... aga siis Idris tea, et kui List Int ≡ Arv ?b siis ?b ≡ False.

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 parameetrile.
- Programm peab töötama, kui kõik null-muutujad asendada ()-ga.

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 → a`.

Motivatsioon

Probleem 2:
Pilditöötlusprogramm, mis rakendab pildile (baitide massiiv) mingit filtrit.

- Puhtal FP-l 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 tagastusväätrus 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 : Int) → 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 tagastusväätrus 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: lineaarsed massiivid

```
infix 5 #

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

export
interface Array arr where
  write  : (1 a : arr) → Nat → Double → arr
  read   : (1 a : arr) → Nat → L arr Double
  size   : (1 a : arr) → L arr Nat
  withArray : Nat → ((1 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
```

Näide: kompileerimiseks LinArray-d

```
export
data LinArray = MkLinArray (IOArray Double)

private
newIOArr : List Double → IO (IOArray Double)
newIOArr xs = ...

export
Array LinArray where
    withArray n f =
        unsafePerformIO (do a ← newArray (cast n)
                            let (_ # r) = f (MkLinArray a)
                            pure r )

    size (MkLinArray a) = MkLinArray a # (cast (max a))

    write (MkLinArray a) i v =
        unsafePerformIO (do ok ← writeArray a (cast i) v
                            pure (MkLinArray a))
    read (MkLinArray a) i =
        unsafePerformIO (do r ← readArray a (cast i)
                            case r of
                                Just v  ⇒ pure (MkLinArray a # v)
                                Nothing ⇒ pure (MkLinArray a # 0))
```