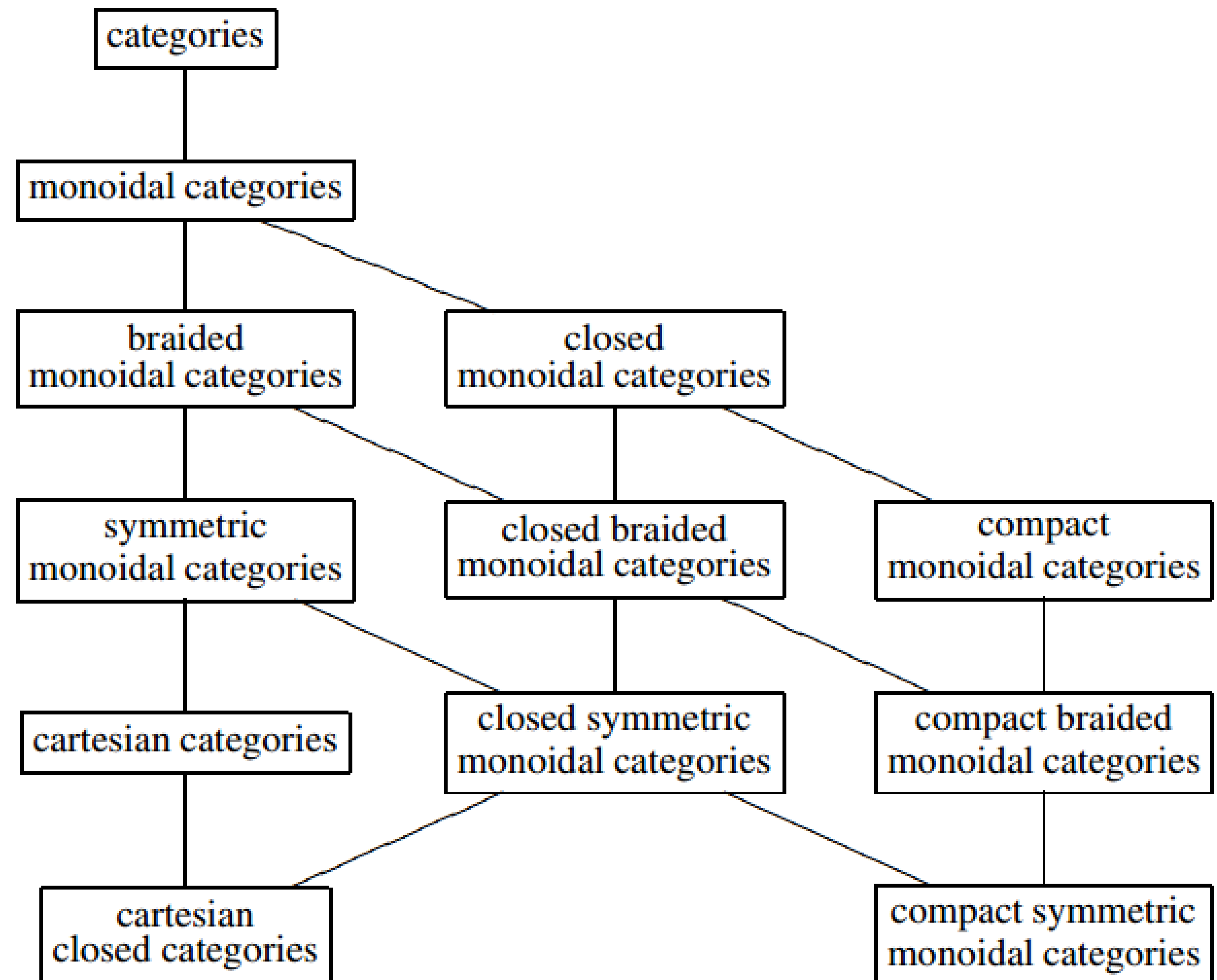
2. Funkcionális Programozás

Visszatekintés

- Logikus összefüggések mentén akarunk programozni és ezeket ellenőrizni is akarjuk, de paradoxonokat sem szeretnénk
- A Curry-Howard korrespondencia azt mondja, hogy a logika megfelel a λ-kalkulusnak, a paradoxonok kikerüléséhez meg típus rendszert is rakunk rá.
- A Curry-Howard korrespondencia hátterében a közös kategória elméleti struktúra áll

Visszatekintés

- A programozás / típuselmélet a Cartesian Closed Category struktúrának felel meg.
- De pl. a fizikai Hilbert terek más fajtának (Compact Symmetric Monoidal Cat)



Visszatekintés

- A Cartesian Closed Category-k ún. belső nyelve a λ-kalkulus.

- A funkcionális nyelvek a λ-kalkulusra épülnek, így a kategóriaelméleti eredmények jórésze átvihető a funkcionális nyelvekbe.

Visszatekintés

- A megfeleltetés alapja, hogy a típusok kategóriát alkotnak, ahol
  - Az objektumok típusok,
  - A morfizmusok függvények,
  - A morfizmusok komponálásának a függvények komponálása felel meg, amely asszociatív,
  - Az identitás morfizmus az identitás függvény.

Következő lépések

- Funkcionális nyelvek összehasonlítása az imperatívakkal
- Minimális Haskell ismeretek
- A C++ template szint, min funkcionális nyelv
- Funkcionális Primitívek

Funkcionális vs. imperatív

Funkcionális	Imperatív
Logikai alapja a λ-kalkulus valamelyik továbbfejlesztett változata	Logikai alapja a hardvernek küldött utasítások sorozata
Fő építő kövei a függvények (matematikai értelemben), és a belőlük épített kifejezések	Fő építő kövei a változók és az ismétlődő utasítás sorozatokat összefoglaló függvények (a szubrutinokból fejlődtek ki)
Jellemzőek a magasabb rendű függvények	A magasabb rendű függvények általában idegenek a nyelvtől

Pure Funkcionális vs. imperatív

Pure Funkcionális	Imperatív
Nincsenek változók, minden érték megváltoztathatatlan (immutable)	Változók: érték adható nekik többször, egy memória helyhez kötődnek
Nincsenek globális állapotok, változók A függvények pure-ok: csak az argumentumaiktól függenek!	Globális állapotok, globális változók, amik mindenhonnan elérhetőek, a függvényeken belülről is!

Funkcionális vs. imperatív

Funkcionális	Imperatív
Nincsenek statement blokkok, minden expression, a függvények „hasa” is egyetlen expression	Statement blokkok: utasítások sorozata, de maguk nem kifejezések (pl. nincs típusuk)
Conditional expression (olyasmi, mint C++-ban a ?: operátor), még akkor is, ha if-else-nek (else kötelező) van álcázva. Van meghatározott típusa	If statement Az else ág nem kötelező
Rekurzió (esetleg fix-pont kombinátorral)	Loop statement (for, while, …)

Pure Funkcionális vs. imperatív

Pure Funkcionális	Imperatív
Ha minden immutable, ilyen problémák nincsenek	A változók miatt mindenféle problémák: • Bejárás során módosuló struktúrák, invalidálódó iterátorok • Szálak között data-race • Mutexek, critical sectionok, stb…
A memóriakezelést a nyelv végzi, a polimorfikusság támogatása miatt általában minden pointerek mögött van, ezért kevésbé hatékony DE: az immutabilitás miatt a fordító sokkal jobban tud optimalizálni	Explicit memóriakezelés, és problémái: • Nullpointerek • Érvénytelen pointerek • Aliasing, kettős felszabadítás DE: kézzel optimalizálható hatékony kezelés

Bár ezek imperatívan is megoldhatóak: rust

Funkcionális vs. imperatív

- Az objektum orientált megközelítés akkor tűnik hatékonyabbnak, amikor rögzített számú műveletet akarunk dolgokon végezni, és a későbbi fejlesztések során elsősorban új dolgokat adunk a kódhoz, nem új műveleteket
- A funkcionális megközelítés akkor tűnik hatékonyabbnak, ha inkább a dolgok száma fix, és elsősorban új műveleteket adunk a rendszerhez a meglévő dolgokon.
- Ez összefügg azzal, hogy az OOP-ban az objektumok polimorfikusak, a funkcionális nyelvekben pedig a függvények.

Funkcionális vs. imperatív

- A fentiek talán érzékeltetik, hogy a két programozási stílus gyökeresen más gondolkodási sémákat kíván.
- Az elmúlt években nyilvánvalóvá vált, hogy mégis érdemes funkcionálisan nekiállni a problémáknak, mert:
  - A funkcionális megoldások sokkal jobban komponálhatóak, ezért jobban skáláznak a probléma mérettel
    (durva példa: ugyan azt funkcionálisan O(log N) kódsor, imperatívan O(N), ha N a probléma mérete)
    konkrét példák vannak az OO tarthatatlanságára, de a vita nagyon indulatos☺
  - A sokszálas környezetekben az imperatív programozás hihetetlenül komplikált lesz a részletezett problémák miatt (szinkronizációk, data race, stb.), ugyanakkor ezek sokkal könnyebben kezelhetőek funkcionálisan.

Néznivaló: Bartosz Milewski – Category Theory

Funkcionális vs. imperatív

Az előző általános megfontolásokon kívül azt érdemes kiemelni, hogy

- A funkcionális nyelvek sokszor sokkal erősebb, kifejezőbb típusrendszerrel rendelkeznek
- Erősebb absztrakciók vannak, tehát generikusabban (újrafelhasználhatóbban) lehet bennük programozni
- A fizikában/matematikában előforduló problémákhoz sokkal jobban illik ez a megközelítés

Funkcionális programozási nyelvek

Jelenleg a következő nyelvekkel találkozni általában sokat:

- Lisp és variánsai (Clojure, Scheme)
- ML és variánsai (F#)
- Scala
- Haskell

Mi most a Haskell-t fogjuk részletezni.

Haskell

„Learn you a Haskell for great good"

Gyorstalpaló:

- `-- comment`
- `f x  --f függvény meghívása x argumentummal`
- `[1, 2, 3] --három elemű lista`
- `4 : [1, 2] –elem hozzáadás listához, eredménye: [4, 1, 2]`
- `(1, "A") --tuple 2 elemmel`
- `4 + 5 --infix operátor meghívása, eredmény: 9`
- ```
  sq :: Int -> Int --1 argumentumú fv deklaráció (nem kötelező)
  sq x = x*x       --a fv törzse
  ```
  ```

Haskell

Gyorstalpaló:

- ```
  sqAdd :: Int -> Int -> Int --2 argumentumú fv deklaráció
  sqAdd x y = x*x + y*y      --a fv törzse
  ```
- `sqAdd 3 4 --értéke: 25`
- ``3 `sqAdd` 4 --infix forma, ugyan az, mint fent: 25``
- `\x y -> x*x + y*y --a fenti függvény lambda formában`
- ```
  sqAdd :: a -> a -> a --2 argumentumú generikus fv deklaráció
  sqAdd x y = x*x + y*y
  ```

C++:

```
template<typename a>
a sqAdd( a x, a y ){ return x*x+y*y; }
```

Haskell

Gyorstalpaló:

C++ analógok:
type ~ typedef,
data ~ enum / struct / union

- `type Name = String --új típus bevezetése korábbiból`
- `type BigFunction = (c -> d) -> a -> (b -> c) -> (a -> b)`

- `data Bool = True | False --sumtype jel: |`
- `data Either a b = Left a | Right b --generikus típus a, b-ben`
- ```
  f1:: Either [a] Int -> Int --részben generikus típusú függvény
  f1 (Left xs) = length xs    --pattern match: definíció az 1. esetre
  f1 (Right y) = y            --pattern match: definíció a  2. esetre
  ```

Konstruktor nevek
  ```

Haskell

Gyorstalpaló:

```
data Vector2D a = Vec2 a a --product type
```

```
sqlength :: Vector2D a -> a
sqlength (Vec2 x y) = x*x + y*y
```

```
data Point = Point {px :: Integer, py :: Integer}
--névvel ellátott mezők a product type-ban (record, struct)
```

Haskell

Gyorstalpaló:

Függvény komponálás, $f \circ g$:

- ```
  (.) :: (b -> c) -> (a -> b) -> (a -> c)
  f . g = \x -> f (g x)
  ```

Függvény alkalmazás alacsony precedenciával / infix formával, $f(x)$:

- ```
  ($) :: (a -> b) -> a -> b
  f $ x = f x
  ```

Haskell

Gyorstalpaló:
Lista műveletek:

- `head [1, 2, 3]`
  `1`
- `tail [1, 2, 3]`
  `[2, 3]`
- `init [1, 2, 3]`
  `[1, 2]`
- `last [1, 2, 3]`
  `3`

Haskell

Gyorstalpaló:
Typeclass-ok, ezek típus megszorításokat írnak le
(mint majd hamarosan C++-ban a Conceptek)

```
class Eq a where --egyenlővé tehető típusok koncepciója
  (==), (/=) :: a -> a -> Bool --két kötelező függvény deklarációja
  x /= y = not (x == y) --definíciók egymással
  x == y = not (x /= y) --így elég az egyiket megadni, a másik
                          következik belőle.
```

```
class Functor f where --funktorként viselkedő típusok koncepciója
  fmap :: (a -> b) -> f a -> f b
```

Haskell

Gyorstalpaló:
Typeclass-ok, ezek típus megszorításokat írnak le
(mint majd hamarosan C++-ban a Conceptek)

- ```
  class Eq a where --egyenlővé tehető típusok koncepciója
    (==), (/=) :: a -> a -> Bool --két kötelező függvény deklarációja
  ```
- ```
  data Point = Point {px :: Integer, py :: Integer}--példa korábbról
  ```
- ```
  instance Eq Point where --az Eq koncepció implementálása
    (Point x1 y1) == (Point x2 y2) = (x1 == x2) && (y1 == y2)
  ```
  ```

C++ template funkcionális nyelv

C++ template funkcionális nyelv

- Számok reprezentálása a template szinten:
- A C++ template nyelve ugyan megengedi a nem típus template paramétereket, de ezek felett nem lehet absztrahálni, ezért csináljuk a következőt:
-
```
template<int x>
struct Int
{
    static const int result = x;
};
```
- Innentől az `Int<n>` már típus, és absztrahálhatunk felette a szokásos `typename` kulcsszóval, ahogy mindjárt látni fogjuk.

C++ template funkcionális nyelv

```
template<int x>
struct Int
{
        static const int result = x;
};
```

Ugyanakkor az Int-séget ellenőriztetni (mint típusságot), csak SFINAE-vel tudunk:

```
template<typename T> struct isInt;//általában undefined

template<int x> struct isInt<Int<x>>{ using result = void; }; //kivéve Int-ekre


template<typename x,
          typename Req = typename isInt<x>::result>
          struct F;
```

- `F` csak akkor lesz használható, ha az argumentuma `Int<n>` alakú.

C++ template funkcionális nyelv

Egy egy argumentumú függvény:

- ```
  sq :: Int -> Int   //Haskell érték szint
  sq x = x*x
  ```

- ```
  int sq(int x){ return x*x; }   //C++ érték szint
  ```

```
//C++ template szint
```

- ```
  template<typename x, typename Req = typename
  isInt<x>::result> struct sq
  {
      using result = Int<x::result * x::result>;
  };
  ```
  ```

C++ template funkcionális nyelv

Mivel a C++ template nyelv szörnyen nyakatekert és terjengős, ezért a továbbiakban nem írjuk ki a típusellenőrzéseket, mert átláthatatlan lenne minden.

Egyszerűbben az előző példa:

```
//C++ ’untyped’ template szint
```

- ```
  template<typename x> struct sq
  {
      using result = Int<x::result * x::result>;
  };
  ```
  ```

C++ template funkcionális nyelv

Az egy argumentumú függvény meghívása:

```
sq :: Int -> Int   //Haskell érték szint
sq x = x*x
let y = sq 4
```

```
int sq(int x){ return x*x; }   //C++ érték szint
int y = sq(4);
```

```
//C++ template szint
```

```
template<typename x>
struct sq{ using result = Int<x::result * x::result>; };
```

```
using y = typename sq<Int<4>>::result;
```

Ehhez a névhez van kötve az eredmény

Ez a kulcsszó kell, hogy a fordító tudja, hogy milyen nyelvi elemet várjon a :: után

C++ template funkcionális nyelv

Listák:

```
std::list<int> xs = {1, 2, 3}; //C++ értékszint
let xs = [1, 2, 3]             //Haskell
```

```
//C++ ’untyped’ template szint
template<typename... elems> struct List;
using xs = List<Int<1>, Int<2>, Int<3>>;
```

C++ template funkcionális nyelv

Hogyan vegyük ki az első elemet a listából?

```
std::list<int> xs = {1, 2, 3}; //C++ értékszint
int x0 = xs.front();
```

```
let x0 = head [1, 2, 3]        //Haskell
```

```
//C++ ’untyped’ template szint
```

```
template<typename... elems> struct List;
```

```
template<typename L> struct Fst;
```

```
template<typename A0, typename... As>
struct Fst<List<A0, As...>>{ using result = A0; };
```

```
using x0 = typename Fst<List<Int<1>, Int<2>, Int<3>>>::result;
```

C++ template funkcionális nyelv

Hogyan vegyük ki az n-dik elemet a listából?

- ```
  std::list<int> xs = {1, 2, 3};
  int xn  = *std::next(xs.begin(), n); //C++ értékszint
  ```

- ```
  let xn = [1, 2, 3]!!n                //Haskell
  ```

```
//C++ ’untyped’ template szint...
Nem olyan egyszerű, rekurziót kell írni...
```

C++ template funkcionális nyelv

Hogyan vegyük ki az n-dik elemet a listából? Rekurzió!

```
//Helper struct a rekurzióhoz
template<typename Icurrent, typename Igoal, typename L> struct Nth_impl;

template<typename Icurrent, typename Igoal, typename A, typename... As>
struct Nth_impl<Icurrent, Igoal, List<A, As...>> //Általános lépés
{
    using result = typename Nth_impl<Int<Icurrent::result+1>, Igoal, List<As...>>::result;
};

template<typename I, typename A, typename... As> //Rekurzió záró lépése
struct Nth_impl<I, I, List<A, As...>>  { using result = A; };


template<typename L, typename I> //felhasználóbarát wrapper:
using Nth = typename Nth_impl<Int<0>, I, L>::result;
```

C++ template funkcionális nyelv

Hogyan vegyük ki az n-dik elemet a listából?

- ```
  std::list<int> xs = {1, 2, 3};
  int xn  = *std::next(xs.begin(), n); //C++ értékszint
  ```

- ```
  let xn = [1, 2, 3]!!n                //Haskell
  ```

```
//C++ ’untyped’ template szint..., az előző dián látottakkal:
using xn = typename Nth< List<Int<1>, Int<2>, Int<3>>, Int<n> >::result;
```

Kód itt

C++ template funkcionális nyelv

Gyakorló példák:

Implementáljuk a 20. dián látható listaműveleteket template szinten!

Írjuk meg azt a típus szintű függvényt, ami két listát összehasonlít, hogy azonosak-e
(ugyan olyan hosszúak, és minden elemük ugyan az,
típusok egyenlőségéhez használjuk az std::is_same-t).

Funkcionális primitívek

Funkcionális primitívek

A kategóriaelmélet – típuselmélet megfeleltetés alapja, hogy a típusok kategóriát alkotnak, ahol

- Az objektumok típusok,
- A morfizmusok függvények,
- A morfizmusok komponálásának a függvények komponálása felel meg, amely asszociatív,
- Az identitás morfizmus az identitás függvény.

Funktor:

- Ez egy leképezés különböző kategóriák között.
- Mivel egy nyelvben egyszerre egy típusrendszer szokott létezni, ezért lényegében csak endofunktorokról beszélhetünk, amelyek kiindulási és kép kategóriája megegyezik.

Funktor

Funktor tulajdonságok:

A funktor korábban bevezetett legfőbb tulajdonsága, hogy létezik hozzá az fmap:

- $\text{fmap}: (a \rightarrow b) \rightarrow f\ a\ \rightarrow f\ b$

Ennek a következő tulajdonságai vannak:

- Legyen az identitás függvény $\text{id}: x\ \rightarrow x$

Ekkor:

- $\text{fmap id}\ =\ \text{id}$
- $\text{fmap}\ (\text{g} \circ \text{h}) =\ (\text{fmap g}) \circ (\text{fmap h})$

Azaz: az fmap-elés disztributív a komponálásra nézve.

Funkcionális primitívek

Funktor:

- ```haskell
  class Functor f where -- Haskell typeclass
  fmap :: (a -> b) -> f a -> f b
  ```

- Szavakban:
  Amelyik típus Funktor, annak kell, hogy létezzen egy fmap függénye, ami vár egy `a -> b` függvényt és egy `a` típus feletti példányát a funktornak, és vissza ad egy `b` típus feletti példányt.

- Pl.:
  ```

Funkcionális primitívek

- Functor Pl.:
- `data Point = Point {px :: Integer, py :: Integer} --példa korábbról`
- ```
  instance Functor Point where
    fmap f (Point x y) = Point (f x) (f y)
  ```

Teljesen hasonlóan Functor-ok lesznek a Vector és Matrix típusaink is, és az fmap-el fogjuk megvalósítani a skalárral való szorzást.
  ```

Funkcionális primitívek

Funktor tulajdonságok:
legyen az identitás függvény:

```
id :: a -> a
id x = x
```

Ezzel:

```
fmap id = id
fmap (g . h) = (fmap g) . (fmap h)
```

Azaz: az fmap-elés disztributív a komponálásra nézve.

Funkcionális primitívek

Funktor:

- ```
  class Functor f where -- Haskell typeclass
  fmap :: (a -> b) -> f a -> f b
  ```
- C++ ban kb.:

- ```
  template<typename Func,
           typename A,
           template<typename> typename Functor,
           typename B = typename std::result_of<Func(A)>::type>
  Functor<B> fmap( Func&& f, Functor<A>&& fa );
  ```

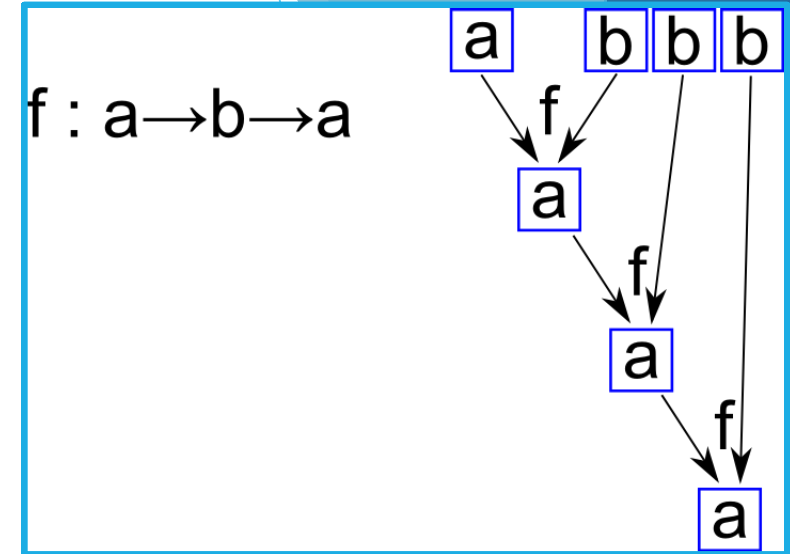
Funkcionális primitívek

Foldable:

- ```haskell
  class Foldable f where -- Haskell typeclass
  foldl :: (a -> b -> a) -> a -> f b -> a
  ```

- Szavakban:
  Amelyik típus Foldable, annak kell, hogy létezzen egy foldl függénye, ami vár egy `a -> b -> a` két argumentumú függvényt egy `a` értéket, egy `b` feletti példányát a funktornak, és vissza ad egy `a` típust.
  ```

Funkcionális primitívek



Foldable:

- ```
  class Foldable f where -- Haskell typeclass
  foldl :: (a -> b -> a) -> a -> f b -> a
  ```

- Mire jó ez? Konkrét példa: Int-ekből lista szummája!
  ```
  foldl :: (Int -> Int -> Int) -> Int -> [Int] -> Int
  foldl (+) 0 [1, 2, 3] -- = 6
  ```

- Tehát az első `a` elem a kezdő (null) érték, majd rá és az első elemre alkalmazódik a bináris fv., majd az eredményre és a 2. elemre ismét, stb.
  ```

Funkcionális primitívek

Foldable:

- A Foldable koncepcióval tehát listák, tömbök sorozatát tudjuk akkumulálni egyetlen elemmé.
- Tipikus példák: szummák, produktumok, min, max kiszámítása
-
```
data Point = Point {x::Int, y::Int, z::Int, w::Int}
instance Foldable Point where
  foldl f zero (Point x y z w) = f (f (f (f zero x) y) z) w

sqsum :: Point -> Int
sqsum p = foldl (\x y -> x+y*y) 0 p
```

Funkcionális primitívek

Foldable megjegyzések :

- Haskell-ben másképp van az alap Foldable typeclass:
  egy monoiddá konvertáló függvényt vár foldMap néven, de ez nem meglepő, hiszen:
  - A monoidban is van egységelem, ez lesz a korábbi zero elem
  - Van kétváltozós művelet, az lesz, amit végig alkalmazunk a listán
- Általában a foldr optimálisabb (pl. listákra), mint a foldl, ezért azt szokás implementálni
- Ezek egymással kifejezhetőek.

Funkcionális primitívek

Foldable:

- C++ ban kb. (elhagytuk a Func szignatúra ellenőrzését):

```
template<typename Func,
         typename A,
         typename B,
         template<typename> typename Foldable>
```

```
A foldl( Func&& f, A&& zero, Foldable<B>&& fa );
```

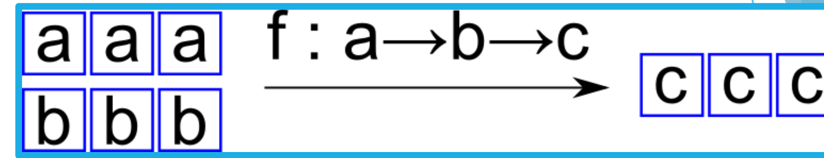
Funkcionális primitívek

„Zippable”:

- ```
  class Zippable f where //Haskell typeclass
  zipWith :: (a -> b -> c) -> f a -> f b -> f c
  ```

- Szavakban:
  Amelyik típus Zippable, annak kell, hogy létezzen egy zipWith függénye,
  ami vár egy `a -> b -> c` két argumentumú függvényt és
  egy `a`, valamint egy `b` típus feletti példányt,
  és visszaad egy `c` típus feletti példányt.
- Pl.:
  ```

Funkcionális primitívek



„Zippable” pl.:

- ```
  data Point = Point {px :: Integer, py :: Integer}
  ```

- ```
  instance Zippable Point where
    zipWith f (Point x1 y1) (Point x2 y2) =
                                  Point (f x1 x2) (f y1 y2)
  ```

- Teljesen hasonlóan Zippable lesznek a Vector és Matrix típusaink is, és a zipWith-el fogjuk megvalósítani pl.: az összeadásukat, kivonásukat.

Funkcionális primitívek

„Zippable” C++ érték szinten kb.:

- ```
  template<typename Func,
           typename A,
           typename B,
           template<typename> typename Zippable,
           typename C = typename std::result_of<Func(A, B)>::type>
  ```
- ```
  Zippable<C> zipWith( Func&& f, Zippable<A>&& fa,
                                 Zippable<B>&& fb );
  ```

Funkcionális primitívek

Konkrét példa, hogy miért jók ezek nekünk:
pl. két vektor skaláris szorzata:

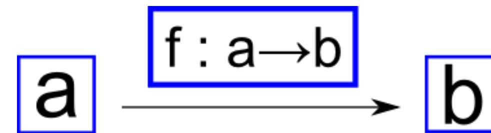
- `foldl (+) 0 (zipWith (*) v1 v2)`

Funkcionális primitívek

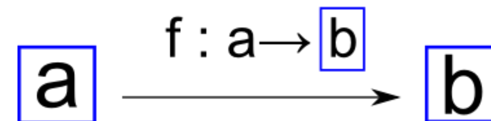
Következzen néhány másik primitív, ami hasznos és sokszor találkozni velük:

- Functor:
  A Functorra máshogyan is lehet tekinteni:
  egy függvényt segít alkalmazni egy kontextusban levő értékre

f : a→b
a
b



- Applicative:
  Ez egy kontextusban levő függvényt tud alkalmazni egy kontextusban levő értékre



- Monad:
  Ez egy függvényt, ami egy kontextust ad vissza, tud alkalmazni egy kontextusban levő értékre

Funkcionális primitívek

Typeclassok:

- ```
  class Functor f where
        fmap :: (a -> b) -> f a -> f b
  ```

- ```
  class Functor f => Applicative f where
        pure  :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b --ap (apply)
  ```

- ```
  class Applicative f => Monad m where
        return :: a -> m a
        (>>=)  :: m a -> (a -> m b) -> m b --bind
  ```
  ```

Funkcionális primitívek

```
class Functor f => Applicative f where
        pure  :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b --ap (apply)
```

A motiváció a következő:
Ha fmap-el alkalmazunk egy függvényt egy funktorra, akkor egy funktort kapunk vissza, azonban ha ez a funktor is függvény (belül), akkor sehogy sem tudjuk alkalmazni argumentumokra, mert nincs ilyesmi funkcionalitás!

Ezt oldja meg az applicative `apply` függvénye!
A `pure` függvény pedig egy sima függvényből csinál egy becsomagolt, `apply`-olható applicative-ot!

Funkcionális primitívek

Applicative szabályok:

```
-- az id becsomagolva és hattatva x-re önmaga:
```

- `(pure id) <*> x = x`

```
-- Homomorfizmus
```

- `(pure f) <*> (pure x) = pure (f x)`

```
-- Felcserélhetőség: mindegy milyen sorrendben
   értékeljük ki f-et és x-et.
```

- `f <*> (pure x) = pure ($ x) <*> f`

```
-- Komponálás (asszociativitás):
```

- `f <*> (g <*> h) = (pure (.)) <*> f <*> g <*> h`

Funkcionális primitívek

Applicative szabályok:

```
-- Felcserélhetőség / kompatibilitás az fmap-el:
```

- `fmap f x = (pure f) <*> x`

Példák: 1), 2), 3)

Funkcionális primitívek

```
class Applicative f => Monad m where
      return :: a -> m a
      (>>=)  :: m a -> (a -> m b) -> m b --bind
```

A motiváció a következő:
Ha az `apply`-al próbálnánk meg olyan függvényt alkalmazni, ami becsomagolt értéket ad vissza, akkor két egymásba ágyazott csomagolást kapnánk (`m (m b)`). A `bind` feloldja ezt, és csak 1x lesz becsomagolva az értékünk!

Funkcionális primitívek

Monad szabályok:

```
-- A return és a bind is a függvény alkalmazást írja le,
return balról egység elem:
```

- `(return x) >>= f   =   f x`

```
-- A return jobbról is egység elemként funkcionál:
```

- `m >>= return   =   m`

```
-- Asszociativitás (mindegy melyik bind történik először):
```

- `(m >>= f) >>= g   =   m >>= (\x -> f x >>= g)`

Funkcionális primitívek

Miért éppen ezek a szabályok?

- Kategória elmélet

Funkcionális primitívek

Egy példán bemutatva:

- Azt akarjuk kifejezni, hogy VAGY van egy A típusú értékünk, vagy egyáltalán nincs értékünk.
- Erre való az `std::optional` (link, olvasnivaló)
- A Haskell-esek ezt maybe-nek hívják:
- `template<typename A> using Maybe = std::optional<A>;`

Funkcionális primitívek

Egy példán bemutatva:

- ```
  template<typename A> using Maybe = std::optional<A>;
  ```

- Ez Functor, mert meg tudjuk írni az fmap-et:

- ```
  template<typename F, typename A,
           typename B = typename std::result_of<F(A)>::type>
  Maybe<B> fmap_impl(F f, Maybe<A> in)
  {
      return in.has_value() ? Maybe<B>{ f(in.value()) } :
                              Maybe<B>{};
  }
  ```

Funkcionális primitívek

Egy példán bemutatva:

- `template<typename A> using Maybe = std::optional<A>;`

- Ebből tudunk csinálni Applicative-ot is:

```
template<typename A>
Maybe<A> pure (Template<Maybe>, A value)
{
   return Maybe<A>{ value };
};
```

Funkcionális primitívek

Ebből tudunk csinálni Applicative-ot is:

```
template<typename F, typename A,
         typename B = typename std::result_of<F(A)>::type>
Maybe<B> apply (Maybe<F> f, Maybe<A> in)
{
   if (f.has_value() && in.has_value())
   {
      return Maybe<B>{ (f.value())(in.value()) };
   }
   else { return Maybe<B>{ }; }
}
```

Funkcionális primitívek

Ebből tudunk csinálni Monad-ot is (most nem ellenőriztük f szignatúráját):

```
template<typename F, typename A,
         typename B = typename GetTemplateArg<typename std::result_of<F(A)>::type>::type>
Maybe<B> bind_impl(Maybe<A> m, F f)
{
  if (m.has_value()) { return f(m.value()); }
  else { return Maybe<B>{ }; }
}
```

Funkcionális primitívek

Ebből tudunk csinálni Monad-ot is (most nem ellenőriztük f szignatúráját):

```
template<typename F, typename A,
          typename B = typename GetTemplateArg<typename std::result_of<F(A)>::type>::type>
Maybe<B> bind_impl(Maybe<A> m, F f)
{
  if (m.has_value()) { return f(m.value()); }
  else { return Maybe<B>{ }; }
}
```

A monadnál a szignatúra:
`m a -> (a -> m b) -> m b`
Azaz a visszatérési értékhez ki kell venni az F függvény eredményének template paramétere alól az argumentum típusát.
Ehhez használunk egy segéd template-t (`GetTemplateArg`), ami ezt megcsinálja.

Funkcionális primitívek

Mi az alapvető különbség az Applicative és a Monad között?

- Az egymás után komponált applicative-ok mindig végrehajtódnak, a függvényeknek nincs irányításuk a lánc felett
  (de legalább lehet őket láncolni, nem úgy, mint a Funktorokat☺).

- A monad-nál azonban a függvény dönti el, hogy milyen monadikus állapotot ad vissza, így potenciálisan befolyásolhatja a lánc lefutását.

Funkcionális primitívek

Algebrai tulajdonságok:

- Mi történik amikor ezek a typeclassok keverednek az algebrai adattípusokkal?
- Összegtípus, Szorzattípus, Komponálás
- A lényeg:
  1. A Functor, Foldable, mindháromra zárt,
  2. Az Applicative csak az utolsó kettőre zárt,
  3. A Monad csak a szorzatra!

Funkcionális primitívek

Példa: a facebook Haxl könyvtára:

A feladat: adatbázishozzáférések aggregálása

Az applicative-okkal láncolhatóak egymás után műveletek, amik azonban csak késleltetve hajtódnak végre, amikor feltétlenül szükség van az eredményre (pl. egy monadikus pontnál), egyébként csak aggregálják, hogy az adott művelet, mely adatokat akarja majd lekérni a működéséhez.

Olvasnivaló: link, Néznivaló: link

Funkcionális primitívek

State monad:
Egy állapotot viszünk végig a láncon, amely változhat, pl. error state

Leírások: 1), 2)

Segítségével megvalósítható olyan lánc, amely sok egymásba ágyazott if-else ágat valósít meg, sokkal kevesebb szintaktikus zajjal.

Lásd: OpenCL inicializálás példakód.

Funkcionális primitívek

Többszálas async kódok írásánál is hasznos lehet ismerni a monad-ot:

Az `std::future<T>` (illetve a most jövő változtatások C++20-ban [talán]) például éppen a monadikus struktúrát fogják megvalósítani.

Olvasnivaló: link, Néznivaló: link

Természetes Transzformációk

```
//Natural transformation, valid for all X!
template<typename X>

std::vector<X> nt_Maybe_to_vector(Maybe<X> m)
{
    return m.has_value() ? std::vector<X>{ m.value() } : std::vector<X>{};
};
```

Egy Maybe mindig átalakítható egy vektorrá, függetlenül attól, hogy milyen típust tárol. Ezt a tényt a fenti Natural Transformation fejezi ki.

Természetes Transzformációk

- Egy könyvtárat, ha nagyon általánosan szeretnénk felépíteni, akkor a bemeneteket nem konkrét típusokat várunk (pl. Mátrix), hanem olyan párokat, amik egy ismeretlen típusú objektumot, és egy belőle ismert típust létrehozó természetes transzformációt tárolnak (Valamilyen tároló -> Mátrix)

- Másrészt: ha olyan rutint írunk, ahol nem is akarjuk tudni, milyen objektumon operálunk, és milyen argumentumot váró függvénnyel, mert ezek a felhasználótól jönnek (de esetleg eltérő helyekkel), akkor a generikusságot fokozhatjuk azzal, ha egy természetes transzformációt is megengedünk a kettő között.

További olvasnivalók

hemzseg tőlük az internet:

- Learn you a Haskell for great good, specifikusan pl. ez a fejezet
- Haskell Book, kapcsolódó olvasó csoport idehaza
- Typeclassopedia
- Wikibooks: Haskell
- Functor, Applicative, Monad explained

„How to learn about Monads:

1. Get a PhD in computer science.
2. Throw it away because you don't need it for this section!"