Utolsó óra összefoglaló

Fourier Transzformáció

A bázistranszformációk általában a következő alakúak:

$$f(x) = \sum_{k=0}^{N} a_k \Phi_k(x)$$

Ahol: f a kifejtendő függvény, a az újbázisbeli együtthatók, Φ-k a (legtöbbször ortonormált) bázisfüggvények. Megj.: fordított irányban is hasonló formulát írhatunk fel, tehát f és a szerepe (melyik az eredeti, melyik a transzformált) csupán nézőpont kérdése.

A legegyszerűbb bázis a Fourier-bázis, ahol: $\Phi_k(x) = \mathrm{e}^{-ikx}$

A bázisokhoz tartoznak kitüntetett pontok (kollokációs pontok, ezek a bázishoz tartozó kvadratúra kiértékelési helyei), amik általában a bázisok zérus helyei, vagy szélsőértékei. Fourier esetben általában: $x_l = \frac{2\pi l}{N}$ -t használunk.

Ezekkel a Fourier transzformáció: $f(x_l) = f_l = \sum_{k=0}^{N} a_k \mathrm{e}^{-ik\frac{2\pi l}{N}}$

Gyors Fourier Transzformáció

Az ötlet az, hogy a szumma átzárójelezhető, és közös faktorok emelhetőek ki, pl. N=8 esetén:

$$a_k = f_0 + f_1 e^{-\frac{2i\pi k}{8}\cdot 1} + f_2 e^{-\frac{2i\pi k}{8}\cdot 2} + f_3 e^{-\frac{2i\pi k}{8}\cdot 3} + f_4 e^{-\frac{2i\pi k}{8}\cdot 4} + f_5 e^{-\frac{2i\pi k}{8}\cdot 5} + f_6 e^{-\frac{2i\pi k}{8}\cdot 6} + f_7 e^{-\frac{2i\pi k}{8}\cdot 7}$$
$$= \left(f_0 + f_2 e^{-\frac{2i\pi k}{8}\cdot 2} + f_4 e^{-\frac{2i\pi k}{8}\cdot 4} + f_6 e^{-\frac{2i\pi k}{8}\cdot 6}\right) + e^{-\frac{2i\pi k}{8}\cdot 1}\cdot\left(f_1 + f_3 e^{-\frac{2i\pi k}{8}\cdot 2} + f_5 e^{-\frac{2i\pi k}{8}\cdot 4} + f_7 e^{-\frac{2i\pi k}{8}\cdot 6}\right)$$
$$= \left(\left[f_0 + f_4 e^{-\frac{2i\pi k}{8}\cdot 4}\right] + e^{-\frac{2i\pi k}{8}\cdot 2}\cdot\left[f_2 + f_6 e^{-\frac{2i\pi k}{8}\cdot 4}\right]\right) + e^{-\frac{2i\pi k}{8}\cdot 1}\cdot\left(\left[f_1 + f_5 e^{-\frac{2i\pi k}{8}\cdot 4}\right] + e^{-\frac{2i\pi k}{8}\cdot 2}\cdot\left[f_3 + f_7 e^{-\frac{2i\pi k}{8}\cdot 4}\right]\right)$$

Ekkor:

- A fenti zárójelek faktorai különbözően periodikusak (ahogy k fut 0-tól 7-ig):
  Legbelül 2 a periódus, ezért k=0, 2, 4, 6-ra pontosan ugyanazt a szummát kell kiszámolni, sőt k=1, 3, 5, 7-re is, csak akkor egy mínusz előjellel.
  A []-k között 4 a periódus, ezért k={0, 4}, {1, 5}, {2, 6}, {3, 7} csoportokra ugyan az a szumma, de a szorzó faktorok: 1, -1, -i, i.
  A ()-között 8 a periódus, tehát csak a -1 · szorzás marad meg.

- Ezek kihasználásával lényegesen csökkenthető az elvégzendő szorzások (és exp számolások) száma.

Gyors Fourier Transzformáció

A műveletek átrendezése viszont fontos, de elég fura alakú.
Ha a bemenő tömb sorban van elhelyezve a memóriában, akkor az indexek:
[0, 1, 2, 3, 4, 5, 6, 7]

Az első lépésben a jelölt párokon kell dolgozni:

[[0, 4], [1, 5], [2, 6], [3, 7]]

Majd a másodikban:

[[0, 2], [4, 6], [1, 3], [6, 7]]

Végül:

[[0, 1], [2, 3], [4, 5], [6, 7]]

A párokon végrehajtandó művelet (pillangó): $(x,\ y) \rightarrow (x + f \cdot y,\ x - f \cdot y)$, ahol f faktor az éppen aktuális exponenciális faktor az előző oldalról.

Viszont ekkor a kimenő tömb sorrendje valójában [0, 4, 2, 6, 1, 5, 3, 7] lesz.

Gyors Fourier Transzformáció

A műveletek átrendezése viszont fontos, de elég fura alakú.
Ha a bemenő tömb sorban van elhelyezve a memóriában, akkor az indexek:
[0, 1, 2, 3, 4, 5, 6, 7]

Az első lépésben a jelölt párokon kell dolgozni:

[[0, 4], [1, 5], [2, 6], [3, 7]]

Majd a másodikban:

[[0, 2], [4, 6], [1, 3], [6, 7]]

Végül:

[[0, 1], [2, 3], [4, 5], [6, 7]]

A műveleti párok képzehetőek a következő képpen: félbevágjuk a tömböt, majd a két felet elemenként párokba rendezzük. Ekkor végrehajtható a pillangó. A következő lépésben a párok tömbjét deflatten-elni kell, azaz a párstruktúrát eltávolítjuk. Ezek után újra képezhetjük a műveleti párokat a pillangóhoz.

Az eljárás végén át kell indexelni a tömböt, hogy jó sorrendben legyen a kimenet, amihez az indexek bináris inverzét kell kiszámolni.

Gyors Fourier Transzformáció

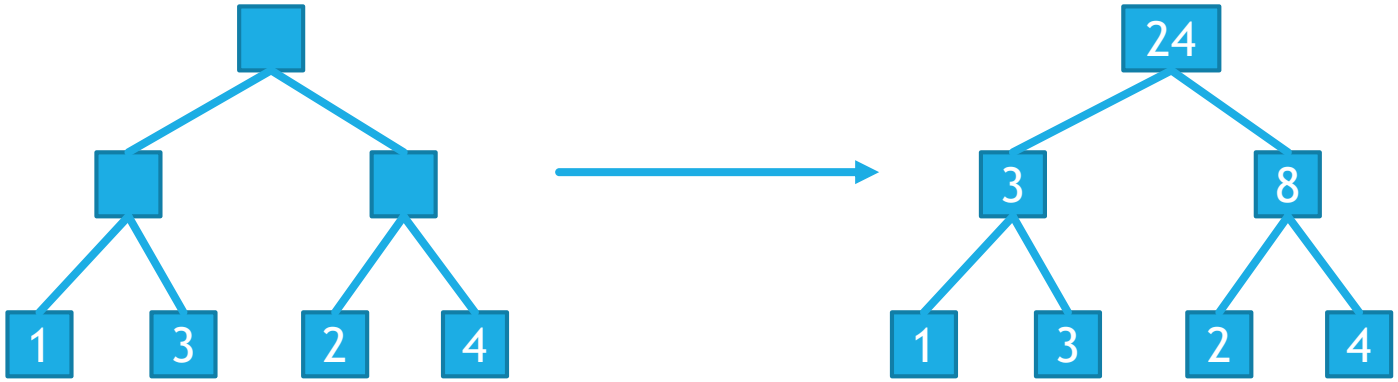
Linkek:
http://www.librow.com/articles/article-10

Bit reversal:
http://www.katjaas.nl/bitreversal/bitreversal.html

Katamorfizmusok

Foldokkal listákon ki tudjuk számolni egy bináris művelet ismételt alkalmazását, és egyetlen elemre redukálni a kollekciót.

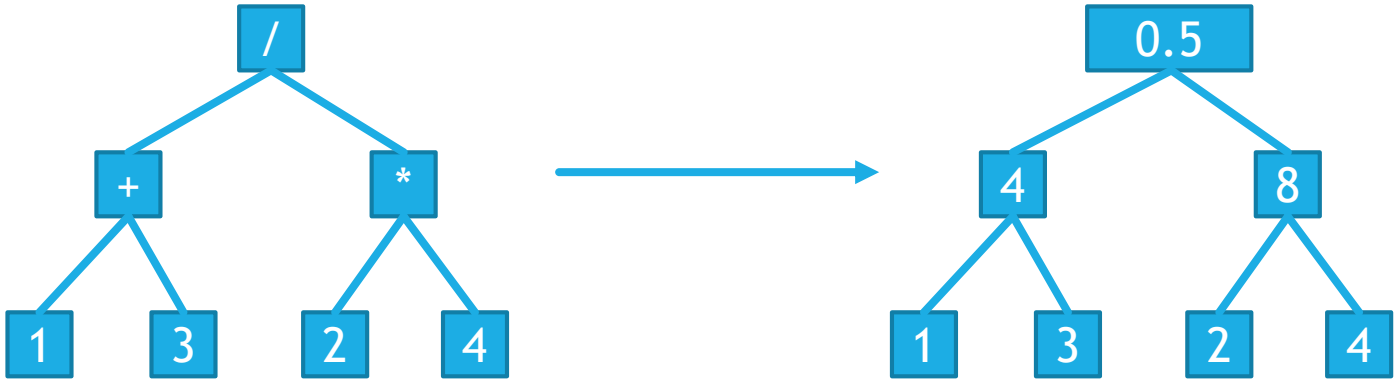
Egyszerűen véggondolható, hogy ez bináris fákra is működik, pl.: a szorzás művelettel:



Ekkor tehát az eredmény 24.

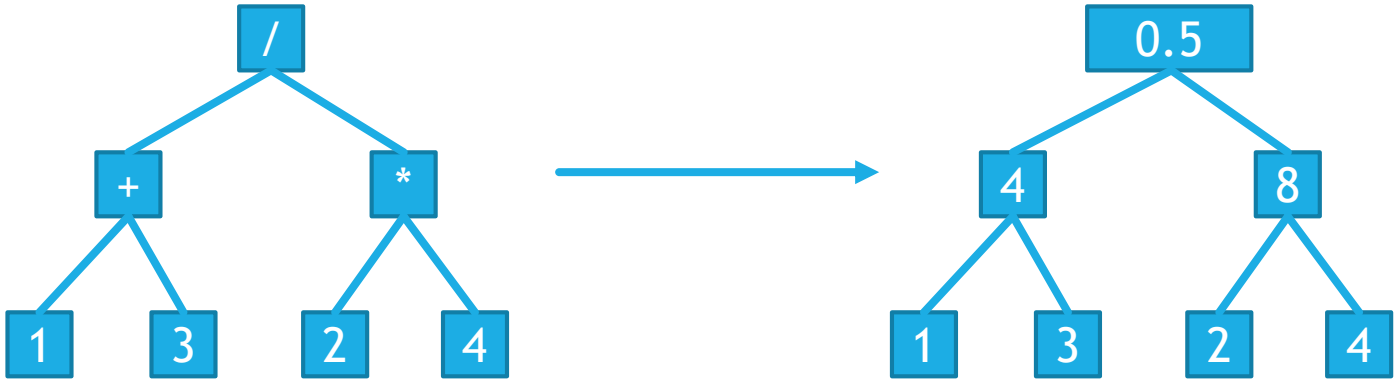
Katamorfizmusok

De, mi van akkor, ha több féle műveletet szeretnénk végrehajtani a fa különböző helyein, pl. mert egy műveleti fát kell kiértékelnünk?



Katamorfizmusok

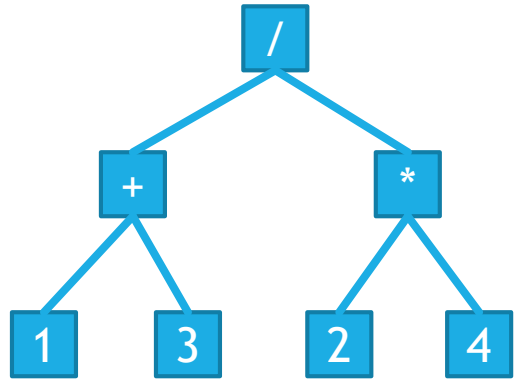
Ha több lehetséges elágazás, és/vagy csúcs fajta van a fában, akkor azok egy sumtype-ot alkotnak. Sumtype-t csak a match-al tudunk eliminálni, ezért a fa alulról felfelé feldolgozása során minden ponton egy match-et kell végrehajtani. Nyilván minden esethez meg kell adni a megfelelő kezelő függvényt.



Katamorfizmusok

Ha pl. a fában a következő dolgok lehetnek:
Konstans, + művelet, * művelet, / művelet

Akkor a fa algebrai alakja vázlatosan:
`Tree = Constant | Add | Mul | Div`



Azonban, mit tárol egy Add? Tárolhat konstanst, alfát (subtree)...

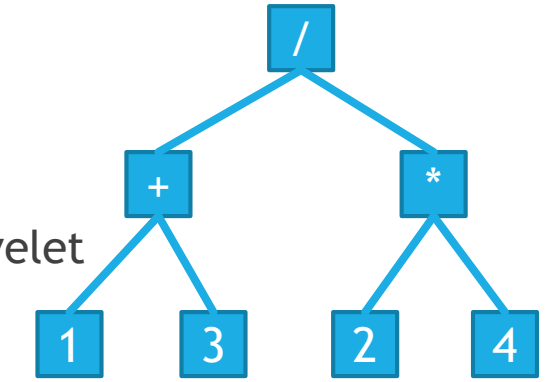
Ahelyett, hogy egy rekurzív definíciót adnánk (amit általában nem támogatnak a típusrendszerek) :

`Tree = Constant Int | Add Tree Tree | Mul Tree Tree | Div Tree Tree`

Kénytelenek vagyunk egy nem-rekurzív, de parametrikus definíciót adni, amit majd később teszünk rekurzívvá…

Katamorfizmusok

Ha pl. a fában a következő dolgok lehetnek:
Konstans (egész szám), + művelet, * művelet, / művelet



Egy nemrekurzív, de parametrikus definíciót adunk, amit majd később teszünk rekurzívvá:

```
Tree a = Constant Int | Add a a | Mul a a | Div a a
```

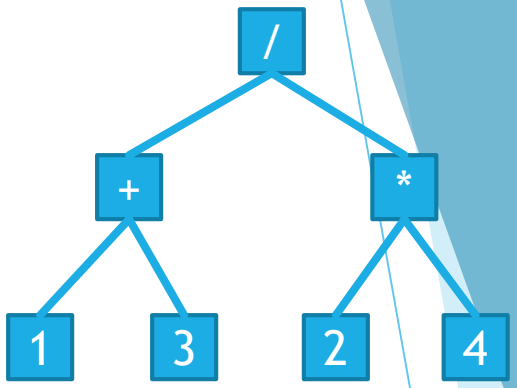
ahol a egy típus paraméter (template arguentum).

A rekurzív fa típust a Fixpont kombinátorral hozzuk létre, mint a korábbi Faktoriális példánál láttuk:

```
ExprTree = Fix Tree
```

Itt `Tree a`-ban az `a` helyére maga a `Tree` helyettesítődik be. A rekurzió nem végtelen: minden értelmes fa előbb utóbb egy `Constant`ban végződik, aminek nincs a paramétere, ezért nem folytatható a rekurzió.

Katamorfizmusok



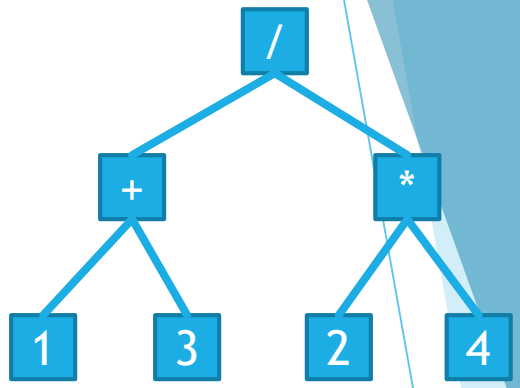
Mivel a Tree egy sumtype, a kiértékelő függvénynek is egy sumtype-nak kell lennie, hogy matchelni lehessen.

Ahhoz, hogy ezek a függvények tetszőlegesen kombinálhatóak legyenek, a visszatérési értéküknek ugyan annak kell lenniük.

Pl.:

$$\text{Evaluator} = \text{Constant} \rightarrow \text{Int} \mid \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \mid \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \mid \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

Katamorfizmusok



Ezt a konstrukciót Kategória Elméletben F-algebrák néven ismerik.

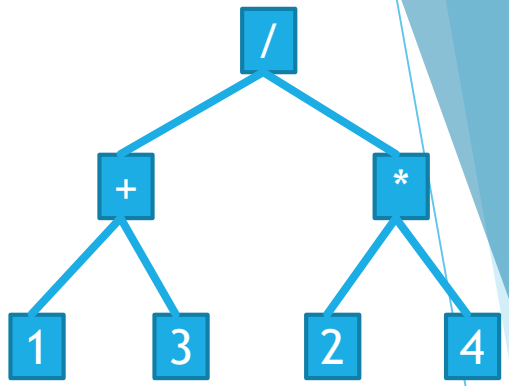
A fa egy Functor: F a, amit egy nemrekurzív parametrikus típus fixálásából kaptnk.

A kiértékelő az Algebra, amely egy függvény kollekció, amelynek az eredendő szignatúrája: F a -> a

A függvény, ami a rekurzív bejárást megcsinálja a katamorfizmus (catamorphism):

cata alg tree = (alg . fmap cata alg . unfix) tree

Katamorfizmusok



cata alg tree = (alg . map cata alg . unfix) tree

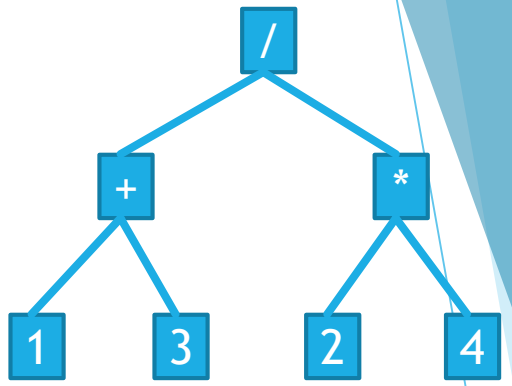
Ezt a következő képpen kell olvasni:

- bejön egy tree
- Erre hattatódik egy unfix, ami felfedi az egy szinttel mélyebben fekvő típust, esetünkben:

  Fix tree -> tree (Fix tree), hiszen a fa elemei is fák.

- Az eredményre hattatódik egy map, aminek a függvénye maga a katamorfizmus és az algebra (azaz már csak egy tree-t vár, amire hatni fog). A tree kötelezően Functor, tehát a map létezik rá, és az implementációja annyi, hogy minden parametrikus (tehát pl. a Constant-ra nem!) elemére hattatja a kapott függvényt, ami persze újra a felfedett altree-kre fogja meghívni a map-et egy unfix után stb… A lefelé lépkedés addig megy, amíg minden ágon el nem érjük a nemrekurzív végelemet (Constant), erre ugyanis nem hat a map!
- Amikor a map visszatér, akkor kerül a vezérlés az algebrához.

Katamorfizmusok



cata alg tree = (alg . map cata alg . unfix) tree

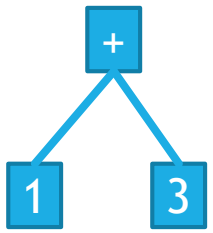
- Alulról, amikor a Constant-hoz ér a bejárás, akkor a map nem csinál semmit, a Constant egyenesen megy az algebrába.
- Matchelődik rá az első függvény, ami mondjuk Int-et csinál belőle (csak kiveszi a tárolt értéket)
- Az eggyel feljebbi szinten ez a visszaadott Int kerül a fában a korábbi alfa helyére, és ezzel tér vissza a map.
- Ezen a szinten az algebra már egy sumtype-ot kap, aminek minden rekurzív alága már ki lett értékelve, és csak az adott szinten levő műveletet kell elvégeznie.

A következőkben ezt lépésenként leírjuk:

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:

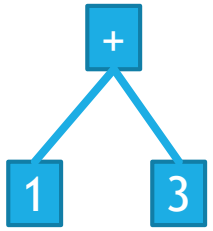


Unfix, map: ekkor itt a típus még: tree (Fix tree)

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:



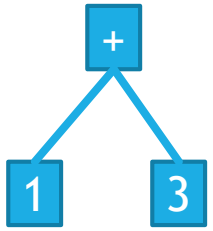
Unfix, map: ekkor itt a típus még: tree (Fix tree)

Unfix, map: ekkor itt a típus is még: tree (Fix tree)

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:



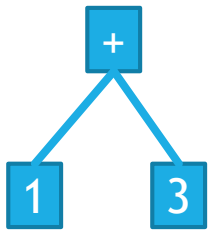
Unfix, map: ekkor itt a típus még: tree (Fix tree)

Itt a map nem hat, mert a Constant nem parametrikus, ezért a típus még mindig tree (Fix tree)

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:



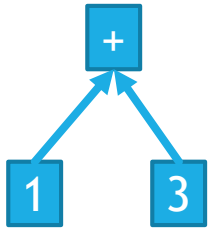
Unfix, map: ekkor itt a típus még: tree (Fix tree)

Hattatódik az algebra, ami a Constant Int-ből visszaadja az Int-et.
Ez mindkét ágon megtörténik egymástól függetlenül.

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:



Unfix, map: ekkor itt a típus még: tree (Fix tree)

A belső cata visszatér az egyik ágon 1-el, a másik ágon 3-al.

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:

Itt a map megkapja a visszatérési értékeket, amiket visszacsomagol a Functor struktúrába, ekkor ezen a szinten a típus `tree Int` lesz!

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Belépés:

4 Végül ezen a szinten is meghívódik az algebra a `tree Int` -el, de mivel ez most a „+” ág, ezért az ennek megfelelő függvény fog a match-be kerülni, és elvégződik az összeadás, aminek az eredménye Int.

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree Int -> Int
```

Visszatérés:

A legkülső cata hívás visszatér Int 4-el.

Katamorfizmusok

```
cata alg tree = (alg . map cata alg . unfix) tree
alg :: tree a -> a
```

Katamorfizmusokkal tetszőleges fákon tetszőleges alulról felfelé bejárást / kiértékelést leírhatunk, segítségükkel kiabsztrahálható a mélységi bejárás.

A catamorfizmusok a fold általánosításai, amikoris a lista struktúra helyett tetszőleges fixált rekurzív functor sumtype van, és egyetlen függvény helyett pedig ennek megfelelő számú.

Olvasnivalók, amik részletesebben elmagyarázzák az elméletet és az implementációt:
http://bartoszmilewski.com/2013/06/10/understanding-f-algebras/

http://ericniebler.com/2013/07/16/f-algebras-and-c/

Anamorfizmusok

Az unfoldok megfelelő általánosításai az anamorfizmusok:

ana coalg seed = (fix . map ana coalg . coalg) seed

coalg :: a -> tree a

Ezek felülről lefelé bejárást valósítanak meg, minden lépésben a tree struktúrába csomagolva a kapott eredményeket, és az új értékeket használva seednek a következő lépésben.

Megjegyzések a példakódhoz

A sumtype-okat általában úgy szokás implementálni, hogy fordítás időben ismert, hogy milyen lehetőségeket tárolhat, és az aktuális választás a futásidejű érték.

Ez lefordítva a fák nyelvére azt jelenti, hogy a fa lehetséges ágtípusai ismertek, de a konkrét aktuális struktúrája (hogy hol milyen művelet van kijelölve) nem. Pl. beolvasunk egy műveleti fát egy fileból.

A numerikus gyakorlatban azonban az ember tudja, hogy milyen műveleti fát akar lekódolni, és mindent fordításidőben akar leírni, hogy a fordító mindent ki tudjon optimalizálni.

Ekkor, ha template nyelven implementáljuk a katamorfizmusokat, akkora fix és unfix nem csinál semmit, hiszen nincs típusa a típus szintű értékeknek. Azonban, ha továbbra is egyetlen implementációt akarunk használni érték és típus szinten, akkor egyetlen alakot kell adnunk a cata-nak, ezért van kiírva a példában a fix, unfix.

Megjegyzések a példakódhoz

Az órai második példa egy $\lambda - kalkulus$t implementál.

A példában az algebra visszatérési értéke nem egy jóldefiniált típus, csupán az a közös bennük, hogy minegyik egy olyan függvény (lambda), amely egy meghatározott struktúrájú argumentumlistát vár.

Az argumentumlista a változók szimbólumait és a függvényhívásokban rögzített értékeit tárolja (Ctx), last in, first out sorrendben. A variadikus listából a get függvény keresi ki az adott változó aktuális értékét.

A példa tömbökkel és map-el kiegészített $\lambda - kalkulus$t implementál, és kiértékeléskor minden elemre a ap-ben új szálat indít.

A példa lényege, hogy pontosan ugyan ez a kód lesz majd futtatható a C++14 template részét támogató GPU-s API-kban (amikor végre elkészülnek), és ekkor várhatóan CSAK az algebra map részének implementációját kell átírni (l. arr_map2 fv.).