TDK dolgozat

Leitereg András

2017

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

Masszívan párhuzamos architektúrák generatív programozása

Témavezető:

Berényi Dániel

Tudományos segédmunkatárs

Wigner Fizikai Kutatóközpont

GPU Labor

Szerző:

Leitereg András

Programtervező Informatikus

MSc, 1. félév

ELTE

Budapest, 2016

AZ EMBERI ERŐFORRÁSOK MINISZTÉRIUMA ÚJ NEMZETI KIVÁLÓSÁG PROGRAMJÁNAK TÁMOGATÁSÁVAL KÉSZÜLT

Absztrakt

A fizikában, a statisztikában és más tudományterületeken is gyakoriak a jellegükből adódóan könnyen párhuzamosítható, de nagy adatmennyiségen operáló számítások. A hatékony párhuzamos implementáció azonban komoly programozási tapasztalatot, és a cél hardverarchitektúra részletes ismeretét kívánná meg, ami akadályt jelenthet a kutatásban. Léteznek előre megírt párhuzamos primitíveket kínáló könyvtárak, de ezek a műveletek általában nem paraméterezhetők tetszőlegesen, és az adatok kezelése sem kellően általános (pl. legfeljebb 3 dimenziós adatstruktúrákat támogatnak).

A számítások sokkal könnyebben kezelhetők, és az előbbi korlátozások is elkerülhetők, ha az elterjedt procedurális megközelítés (a könyvtár által biztosított alapfüggvények egymás után hívása) helyett a számítások műveleti fáját elemezzük, transzformáljuk.

A dolgozatomban bemutatok egy lineáris algebrai kifejezések felírására alkalmas, könnyen optimalizálható műveletkészletet, vizsgálom a műveleti fák általános transzformációit és annotációját rekurziós sémák segítségével, bemutatok egy tetszőleges dimenziós struktúrák flexibilis és hatékony kezelésére alkalmas memória modellt, a műveletfa kiértékelését pedig CPU-ra generált C++ programmal demonstrálom.

Tartalom

Ábrajegyzék

1. Bevezetés

A legtöbb reáltudományban szükség van lineáris algebrai számítások elvégzésére. Nagy szerepük van az adatelemzésben, statisztikában, de a differenciálegyenletek diszkretizációja, illetve a sajátérték problémák megoldása is ilyen feladatokra vezet. A számítások gyakran mátrix-vektor műveletekként vannak felírva, és számítógépen értékelik ki őket. A sűrű (dense) mátrix-vektor műveletek könnyen párhuzamosíthatók, ezt nagy mennyiségű adattal való számolás esetén érdemes kihasználni.

Napjainkban a különböző párhuzamosságot támogató platformok (sokmagos processzorok, videokártyák, publikus klaszterek, felhők, ködök) elterjedésével és rohamos fejlődésével lehetőség lenne rengeteg nagy számításigényű feladat párhuzamos végrehajtására. Sajnos ezeknek a platformoknak még nem létezik egységes és egyszerűen használható interfésze, valamint a maximális hatékonyságú kihasználáshoz részletes ismeretek kellenek az adott rendszer működéséről. A különböző területekről érkező (fizikus, matematikus) szakembereknek nincs ideje, kapacitása elmélyülni a párhuzamos programozásban és optimalizálásban. Ők lennének a legtöbben, akik szeretnének nagy számításigényű feladatokat futtatni, de mégis sokan használnak olyan programokat, amik közel sem aknázzák ki teljesen a rendelkezésre álló hardver képességeit. A Wigner Fizikai Kutatóközpontban működő GPU Labor több megközelítésen keresztül is keres megoldást erre a problémára. A jelen dolgozat témája több dimenziós lineáris algebrai műveletek általános reprezentációja és automatikus párhuzamosítása.

1.1 Motiváció

Illusztrációként tekintsünk néhány példát gyakran előforduló adatfeldolgozási problémákra, melyek többdimenziós adatokon dolgoznak:

Képelemzés: Egy videó biztosan legalább három (két kép és egy idő) dimenzióból áll, de akár az egyes színcsatornákat is külön dimenzióknak tekinthetjük. Így egy videó elemző, átalakító algoritmusnak tudnia kell sokdimenziós adatnézeteket kezelni.

Fizikai számításokban a világ 3+1 dimenziós voltából adódóan sok folyamat leírásához 3, 4D-s függvények, tenzorok kellenek. Azonban ha fázistérben (koordináták és impulzusok

tere) szeretnénk leírni a folyamatot, akkor egy egyszerű részecskéhez is 7 dimenzióra lesz szükségünk: 3 tér, 3 impulzus és 1 idő. Egy kétrészecske fázistér már 2 x (3+3) + 1 dimenziós. A dimenziók száma igen flexibilisen változhat egyszerűsítő feltevésekkel, például szimmetriák (tükrözés, forgatás, gömbszimmetria, izotrópia) feltételezésével. A dimenziószám szerinti esetszétválasztással megírni egy programot képtelenség, szükség van a dimenziók számától elvonatkoztatott adat és művelet reprezentációra

Adatfeldolgozás, BigData: ahány rögzített tulajdonság, annyi dimenzió. Például a geoinformatikában, meteorológiában 2D tér a gömbfelszín, és ehhez jön még a magasság, az idő és minden megfigyelt mennyiség.

Léteznek eszközök többdimenziós lineáris algebrai számítások támogatására, de ezek élesen elkülönítik a dimenziókat, nem könnyű változtatni az algoritmuson, ha kiderül, hogy elhagyható egy dimenzió, vagy szükség van egy újra. Az ezek által kínált algoritmusok nem ágyazhatók egymásba tetszőlegesen, mert rögzítve vannak az adott dimenziós struktúrához.
Az elterjedt könyvtárak többsége (pl. a BLAS [1] és az Eigen [2]) csak mátrix-vektor műveleteket támogat. Néhány könyvtár még tartalmaz 3D-s struktúrákat [3], de magasabb dimenziós struktúrákhoz nem nyújt támogatást.
Léteznek N-dimenziós adatreprezentációt kínáló implementációk is (pl. a Boost MultiArray [4], és a Blitz++ [5]), de ezekhez sem állnak rendelkezésre egymásba ágyazható és testreszabható műveletek.
Összességében elmondható, hogy a használható alapműveletek általában előre megadottak (pl. mátrixszorzás), sok függvény nevet, operátort kell megjegyezni, és nem is lehet őket testreszabni az alábbi értelemben:
Ha például szeretnék kiszámolni az A és B mátrixokból és a v vektorból a C mátrixot a $C_{ik} = \sum_j A_{ij}\, v_j\, B_{jk}$ képlettel, akkor ehhez nem tudnánk kis módosítással felhasználni a beépített mátrixszorzás műveletet.
Hasonlóan az $u_i = \sum_j A_{ij}\, f(v_j)$ egyenlőség lényegében egy mátrix-vektor szorzást ír le, de ha a művelet előre rögzített, akkor nem tudjuk benne hattatni a szükséges f függvényt.

Kutatásunk tárgyaként olyan szoftvert terveztünk, ami ezekre a problémákra kínál jobb megoldást. A fentiekkel szemben az általunk biztosított műveletek (amelyeket részletesen a

2.1 és a 3.3 fejezetekben mutatunk be) egymásba ágyazhatók és tetszőlegesen sok dimenzióban működhetnek. További előnyeik, hogy könnyen ellenőrizhető és optimalizálható műveleti tulajdonságaik vannak, intuitívak, de mégis flexibilisek, mert függvény argumentumaikon keresztül testreszabható a működésük.
A korábban említett könyvtárakkal és a GPU API-kkal ellentétben a 3 dimenzió feletti adatstruktúrák reprezentációját is támogatjuk adatnézetekkel, melyeket részletesebben a 3.6 fejezetben tárgyalunk.

Kutatásunkról korábban az Általános vektorműveleti fák párhuzamosított kiértékelése [6] című szakdolgozatban számoltunk be. Az akkor készült (a most bemutatotthoz hasonló funkcionalitású) FParLin könyvtár C++-ban íródott, most viszont az implementáció nyelvének a Haskellt választottuk. Ennek előnye, hogy a számunkra szükséges magas szintű fogalmakhoz, funkciókhoz rendelkezésre álló csomagoknak (Control.Comonad.Cofree, recursion-schemes [7], Vinyl [8]) és az erős típusrendszernek köszönhetően gyors benne a fejlesztés, a forráskód pedig tömörebb és jobban kezelhető. A kódgenerálás során a teljesítménynek nincs kritikus szerepe, így nem volt miért alacsonyabb szintű nyelvet választanunk.
Ugyanakkor a Haskell tisztán funkcionális volta több helyen bonyolulttá teszi a kódot (pl. véletlenszám generátor lejuttatása a kifejezésfában), részlegesen módosító fa transzformációk nehezen fejezhetők ki, kevésbé optimálisak. De összességében az előnyök ellensúlyozzák a hátrányokat.

1.2 Célkitűzés

Kutatásunk célja nem abszolút optimális kód előállítása, mert az olyan hardver, szoftver és feladat specifikus részletektől függ, amiket nagyon nehéz figyelembe venni, kiabsztrahálni (pl. utasítás költségek, pipeline modell). Inkább a lehető legáltalánosabb építőköveket szeretnénk megadni, és olyan programot generálni, ami a naiv kódnál lényegesen hatékonyabb. A dolgozat egy lépéssel közelebb visz ahhoz a hosszabb távú célhoz is, hogy a kiértékelés a többi nem flexibilis könyvtárral összemérhető teljesítményű legyen. A generált kódtól azt is elvárjuk, hogy a leggyakoribb programozási hibákat, és a hardver számára

leginkább szuboptimális megoldásokat elkerülje. Ilyenek a memóriafoglalás, rossz memória elérési mintázatok, rossz cache kihasználás, nem megfelelő szintű párhuzamosítás stb.

Szeretnénk tehát egy jó kompromisszumot teremteni a fejlesztési és a futási idő minimalizálása között olyan kutatók, tudósok számára, akiknek nem a szoftverfejlesztés a fő profiljuk, de a problémáik számításigényesek, és az elérhető módszereknél többre van szükségük.

2. Elméleti háttér

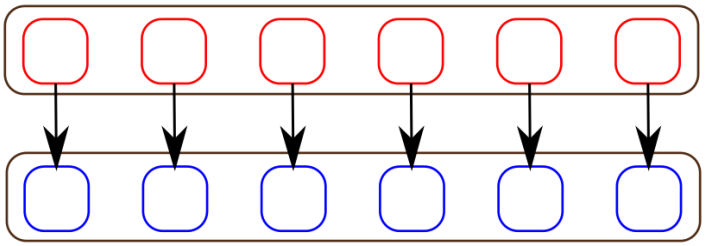
A bevezetőben megfogalmazott általános és flexibilis működéshez magas szintű absztrakciókra volt szükségünk. Testreszabható műveleteknek a funkcionális programozásban széleskörűen használt magasabbrendű függvényeket választottuk, a műveleti fák elemzésére pedig rekurziós sémákat használunk, melyek a kategória elméletből származnak. A következő fejezetek lefektetik ezeknek a fogalmaknak az elméleti alapjait.

2.1 Magasabbrendű függvények

A magasabbrendű függvények általános jellemzője, hogy paramétereik között szerepel egy függvény, amelyen keresztül rendkívül flexibilisen testreszabható a működésük. Az alábbi ábrákon a színek az elemek és tárolók típusát, a nyilak a függvényalkalmazást jelölik.

2.1.1. Map

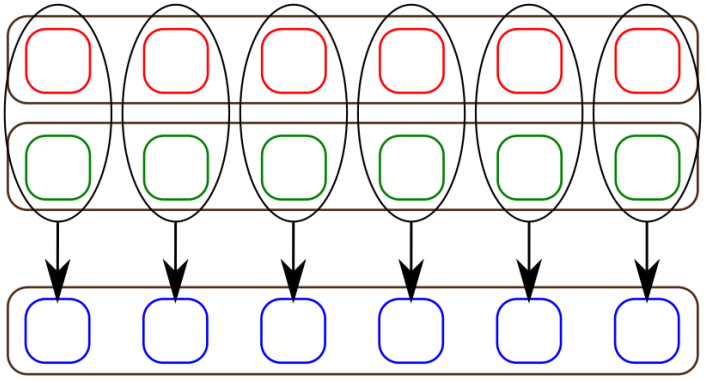
Paraméterei egy unáris függvény és egy tömb. Az eredmény egy tömb, aminek minden eleme a bemeneti tömb megfelelő elemének a függvény alatti képe.



Ábra 1 - A map függvény

2.1.2. Zip

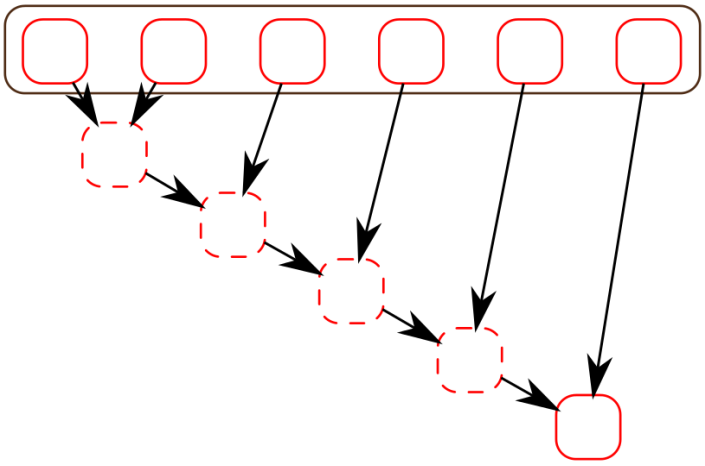
A zip a map általánosítása, bemenetként egy kétváltozós függvényt és két azonos méretű tömböt vár. Az eredmény a tömbök elemenkénti kombinációja a függvényen keresztül.



Ábra 2 - A zip függvény

2.1.3. Reduce

A reduce paraméterei egy kétváltozós függvény és egy tömb. Minden lépésben kombináljuk a tömb első két elemét a függvényen keresztül, és ez lesz az új első elem. A művelet eredménye az utolsó függvény alkalmazás értéke.



Ábra 3 - A reduce függvény

A fenti függvények nagyon fontos tulajdonsága, hogy egyszerűen és hatékonyan párhuzamosíthatók. A map és a zip esetében az eredmény tömb elemeit egymástól függetlenül kiszámíthatjuk. A reduce akkor párhuzamosítható, ha a megadott függvény asszociatív. Ekkor átrendezhetjük a számítást egy bináris fába, amiben minden szint részeredményeit egyszerre számolhatjuk.

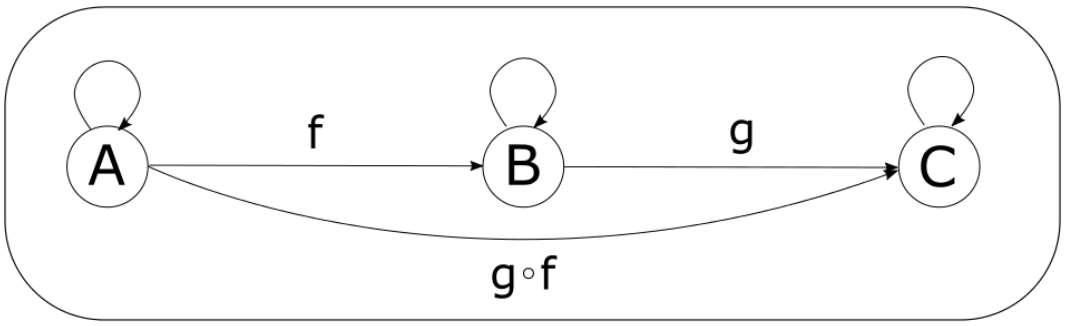
Ismertek a magasabbrendű függvényekre vonatkozó azonosságok is, melyeket a jövőben fel fogunk használni a velük felírt kifejezések egyszerűsítésére, optimalizációjára:

$$\text{map } f_1 \ (\text{map } f_2 \ v) \ = \ \text{map } (f_1 \circ f_2) \ v$$

$$\text{zip } f_1 \ (\text{map } f_2 \ v_1) v_2 \ = \ \text{zip } (\lambda x. \lambda y. f_1 \ (f_2 \ x) y) v_1 \ v_2$$

2.2 Kategóriaelmélet

A fatranszformációra használt rekurziós sémák a kategóriaelméletből születtek, és nagyban építenek ennek bizonyos fogalmaira, ezért a következőkben bemutatjuk ezeket a fogalmakat.



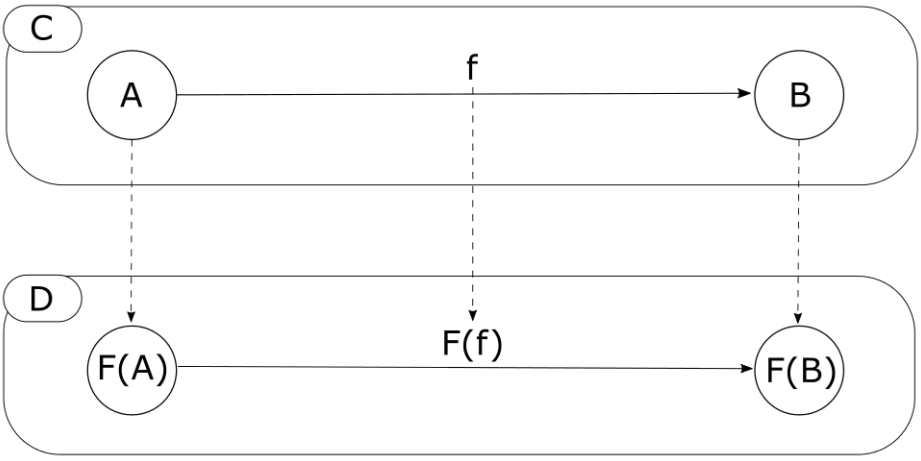
Ábra 4 - Objektumok és morfizmusok egy kategóriában

A kategóriaelmélet az algebrai struktúrák közötti összefüggéseket, leképezéseket tanulmányozza. Maga a kategória egy algebrai struktúra, amelynél a teljességet (zártságot) nem követeljük meg, csak az asszociativitást és az egységelemet.

A kategóriák alkotó elemei objektumok és ezeket összekötő morfizmusok. Minden morfizmusnak létezik egyértelmű kezdő- és végpontja, és a morfizmusok komponálhatók. Tehát ha f: A→B és g: B→C morfizmusok, akkor egyértelműen létezik az g∘f: A→C morfizmus is. A kompozíció művelete asszociatív, azaz bármely h∘g∘f kompozícióra (h∘g)∘f = h∘(g∘f). Emellett minden X objektumhoz biztosan tartozik egy id_X:X→X önmagába képző morfizmus, amely a kompozícióban egységelemként viselkedik. Tehát bármely A, B objektumokra és f: A→B morfizmusra f∘id_A = f = id_B∘f.

2.3 Funktor

Gyakran nem elég önmagukban, izolált algebrai struktúraként tekintenünk a kategóriákra, hanem az ilyen struktúrák közötti leképezéseket kell vizsgálnunk. A kategóriák közötti leképezések közül a legegyszerűbbek a funktorok.



Ábra 5 - Az F funktor által létesített leképezés

Egy F leképezés valamely C és D kategóriák között akkor funktor, ha

- minden C-beli A objektumhoz hozzárendel egy F(A) objektumot D-ből
- minden C-beli f: A→B morfizmushoz hozzárendel egy D-beli F(f): F(A)→F(B) morfizmust úgy, hogy megőrzi az identitás morfizmusokat és a kompozíciókat, azaz
    - $\forall A \in C: F(id_A)=id_{F(A)}$
    - $\forall (f: A \to B, g: B \to C) \in C: F(g \circ f) = F(g) \circ F(f)$

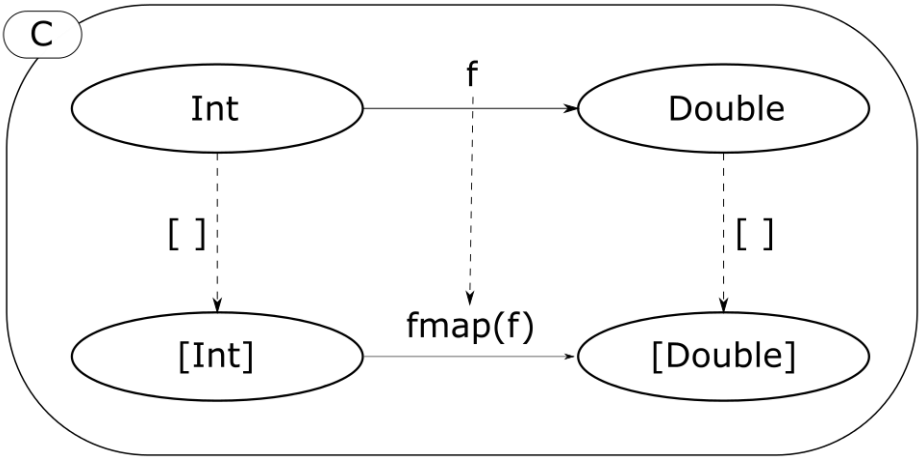
A programozási nyelvek típus rendszere is felfogható egy kategóriaként, ahol az objektumok típusok és a függvények a morfizmusok, a funktorok pedig típus szintű leképezések polimorfikus, vagy generikus típusok (pl. tárolók) felett. Ez az a tulajdonságuk, ami miatt nagyon jól használhatók a kifejezés fák felépítésében és transzformálásában. Mivel csak egyetlen kategória van ekkor jelen, ezért a Funktorok kiinduló és cél kategóriája megegyezik, így endofunktorokról beszélhetünk. A kategóriaelméletből levezethető összefüggéseknek azonnal használható eredményei vannak a programozási nyelvekben, így például az általunk használatos funktorokra mindig létezik és értelmes az fmap magasabbrendű függvény, ami

rögzített F funktor esetén minden f függvényhez hozzárendeli a fent leírt tulajdonságokkal rendelkező F(f) képét.

Az fmap függvény legfontosabb tulajdonságai:

- fmap(id) = id, ahol id az identitás függvény
- fmap(f ∘ g) = fmap(f) ∘ fmap(g), vagyis az fmap disztributív a függvény kompozícióra nézve

A következő példában a funktor (típus szintű leképezés, tároló) a Haskell lista, az fmap pedig képes egy függvényt a listán belül hattatni.



Ábra 6 - A lista funktor és az fmap működése

2.4 Rekurzív adatstruktúrák

Számos esetben van szükség olyan adatstruktúrákra, amik ismétlődők abban a tekintetben, hogy a részeik ugyan olyan szerkezetűek, mint maga a részeket tároló egész. Ezeket a befelé ismétlődő adatszerkezeteket rekurzív adatstruktúráknak hívjuk. Ezeknek az absztrakt, közös tárgyalásához hasznos fogalom a fixpont.

Egy leképezés fixpontja olyan értéket jelent, amit a leképezés önmagába visz. fix f-fel jelölve az f leképezés fixpontját, ezt a

$$fix\ f\ =\ f\ (fix\ f)$$

azonossággal írhatjuk le. Ezt az azonosságot mindkét irányban alkalmazni fogjuk, legyen

```
in: f (fix f) -> fix f
out: fix f -> f (fix f)
```

Az utóbbit többször ismételten alkalmazva a $fix\ f\ =\ f(f(f(f(\ldots))))$ alakot kapjuk, ami jól mutatja, hogy a fixpontra a függvény végtelen sokszor való alkalmazásaként is gondolhatunk. Ez az alak azonban véges is lehet, ha a függvénynek van nem rekurzív ága. Készítsünk egy egyszerű műveleti fát az

```
ExprF a = Const Int | Add a a | Mul a a
```

összegtípus segítségével. Ez a struktúra funktor, hiszen áganként hattathatunk rá egy a→b függvényt, és a felépítése nem változik. Ha az a paraméteren keresztül önmagába ágyazzuk, képes potenciálisan végtelen mélységű fa reprezentálására, amely minden szinten ezekből a fajta elágazásokból, levelekből áll. A kapott konstrukció típusa `ExprF ExprF ExprF …` lesz, amiről már láttuk, hogy `fix ExprF`-fel ekvivalens. Ez nem jelent szükségszerűen végtelen mélységet, mert, ahogy az előbb is láttuk, ha van nem rekurzív ág, esetünkben a `Const`, akkor a rekurzió lezárásra kerül, hiszen a `Const` nem függ a típus paramétertől. Nézzük meg most például az 1+2*3 kifejezéshez tartozó kifejezésfát:

```
in (Add
        (in (Const 1))
        (in (Mul (in (Const 2)) (in (Const 3)))))
```

Az `in` műveletet tekinthetjük “becsomagolásnak”, ami elrejti a fa közvetlen gyerekeit, egyetlen értékké általánosítja a fát. Ez lehetővé teszi, hogy a különböző mélységű részfákat egységesen kezeljük a műveletek operandusaiban. Persze kiértékeléskor meg kell tudnunk különböztetni a műveleteket, erre az `out` függvényt, a “kicsomagolást” használhatjuk.

2.5 Rekurziós sémák

A rekurziós sémák [9] a rekurzív adatszerkezetek természetes, automatikus bejárásai, melyeket Erik Meijer, Maarten Fokkinga és Ross Paterson alkotott meg 1991-ben. A szerzők a kategóriaelméletre építve írták le ezeket az egyszerű, komponálható műveleteket. A rekurzív adatszerkezetek rendkívül elterjedtek, így egy általánosított bejárás nyilvánvaló előnye, hogy kiválthat sok - egy-egy konkrét típus bejárására készített - függvényt. Emellett azáltal, hogy elválasztjuk a bejárás módját attól, amit a függvény valójában csinál az adattal, közvetlenül a rekurzív függvények fő céljára koncentrálhatunk, és könnyebben érthető lesz az algoritmusunk. Optimalizációs lehetőségek is ismertek a különböző rekurziós sémákhoz,

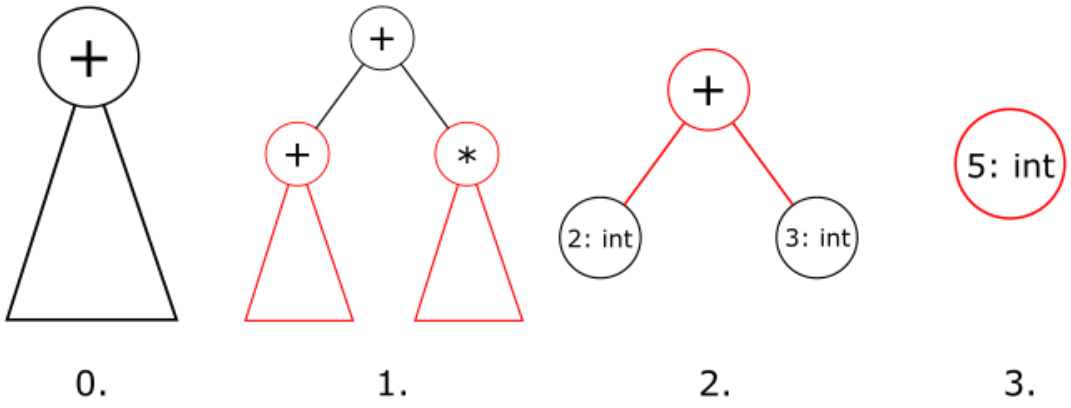
melyeket fel tudunk használni a transzformációk egyszerűsítésére, gyorsítására (cata compose, banana split theorem).

A fák tipikus példái a rekurzív adatszerkezeteknek, így egyértelmű választás volt a rekurziós sémák használata a kifejezés fák transzformációihoz. Vannak alulról felfelé lebontó, és felülről lefelé felépítő sémák, amik egymás duálisai a kategóriaelméletben. Ezek közül néhányat mutatunk be az alábbiakban.

2.5.1. Katamorfizmus

Legyen adott egy becsomagolt kifejezésfa, ahogy azt a 2.4 fejezetben bevezettük. Definiáljunk ezen a fán egy alulról felfelé bejáró rekurzív transzformációt a következőképp:

1. csomagoljuk ki a gyökér csúcsot, hogy hozzáférhessünk a gyerekeihez
2. alkalmazzuk a transzformációt rekurzívan a gyökér csúcs gyerekeire
3. transzformáljuk a gyökér csúcsot az adott művelethez tartozó logikával, felhasználva a már átalakított gyerekeket



Ábra 7 - A katamorfizmus lépései

Az ábra jól mutatja, hogy a katamorfizmus egy lebontó transzformáció, a teljes fát egyetlen értékké redukálja.[1] Ennek az értéknek a típusát *carrier type*-nak, a 3. lépés szerinti művelet specifikus logikát pedig *algebrának* hívjuk. Ez utóbbira gondolhatunk úgy, mint azonos (carrier type) visszatérési típusú függvények halmazára, amelyben minden művelethez tartozik pontosan egy átalakító függvény. Az algebrát a transzformáció során (a 3. lépésben)

[1] Ez az érték lehet egy fa is, így fa→fa transzformációk is leírhatók katamorfizmussal.

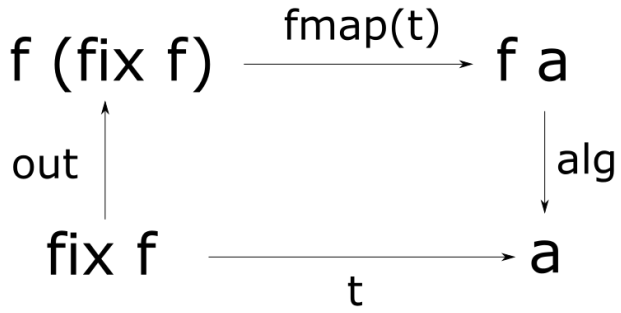
olyan csúcsokra alkalmazzuk, amiknek a gyerekei már transzformálva lettek. Az algebra típusa tehát (a haskell jelölésrendszerét használva, a fát alkotó funktort `f`-fel, a carrier type-ot `a`-val jelölve):

```
type Algebra f a :: f a -> a
```

A teljes transzformáció pedig:

```
cata :: Algebra f a -> fix f -> a
cata alg = alg . fmap (cata alg) . out
```

A katamorfizmus paraméterei az alkalmazandó algebra és a becsomagolt fa, a definíció pedig a fenti felsorolás formalizációja. A végrehajtás egyes lépéseiben létrejövő típusokat szemlélteti a következő ábra (t-vel jelölve a teljes transzformációt):



Ábra 8 - A katamorfizmus típusátmenetei

Tekintsük a 2.4 fejezetben már bemutatott egyszerű műveleti fát, melyet a

```
data ExprF a = Const Int | Add a a | Mul a a
```

funktor ír le.

Készítsünk egy kiértékelő algebrát az ilyen szerkezetű műveleti fákhoz:

```
evalAlg :: Algebra ExprF Int
evalAlg (Const x) = x
evalAlg (Add x y) = x + y
evalAlg (Mul x y) = x * y
```

Ezek után az 1+2*3 kifejezéshez tartozó

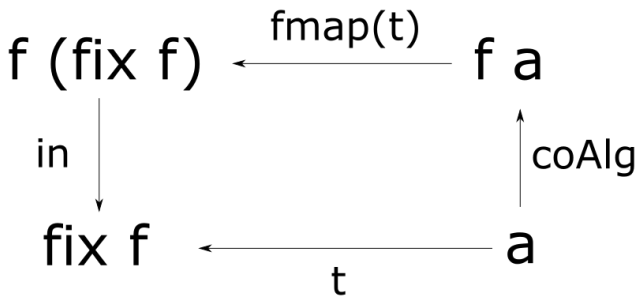
```
expr = in (Add
     (in (Const 1))
     (in (Mul (in (Const 2)) (in (Const 3)))))
```

műveleti fára a következőképpen tudjuk felírni a kiértékelő transzformációt:

```
cata evalAlg expr
```

2.5.2. Anamorfizmus

Ha a katamorfizmus típusátmenet ábráján minden nyilat megfordítunk, az alulról felfelé lebontó helyett egy fentről lefelé felépítő transzformációt kapunk. Az algebra duálisát, ami egy csúcs feldolgozása helyett létrehoz egyet, *koalgebrának* nevezzük.



Ábra 9 - Az anamorfizmus típusátmenetei

Az anamorfizmus működése:

- egy carrier type értékből a koalgebra létrehoz egy csúcsot, aminek a gyerekei carrier type-ok
- rekurzívan alkalmazzuk a transzformációt az új csúcs gyerekeire
- végül a felépített részfát becsomagoljuk

Ugyanez formálisan:

```
type CoAlgebra f a = a -> f a
ana :: CoAlgebra f a -> a -> fix f
ana coAlg = in . fmap (ana coAlg) . coAlg
```

2.5.3. Paramorfizmus

A katamorfizmusok egyszerűek és elegánsak, de a funkcionalitásuk sok esetben nem elegendő. Az algebra az aktuális csúcs eredeti gyerekeihez nem fér hozzá, csak a transzformációjuk eredményéhez, ami viszont nem feltétlenül tartalmaz minden szükséges információt. Az olyan algebrákat, amelyek a csúcs gyerekeinek eredeti értékét is felhasználják, *R-algebráknak* nevezzük:

```
type RAlgebra f a = f (fix f, a) -> a
```

Ahhoz, hogy egy R-algebrát elláthassunk mindkét információval, a katamorfizmust így kell módosítanunk:

- csomagoljuk ki a gyökér csúcsot, hogy hozzáférhessünk a gyerekeihez
- alkalmazzuk a teljes transzformációt a gyökér csúcs gyerekeire, *de emellett őrizzük meg a gyerekek eredeti értékét is*
- transzformáljuk a gyökér csúcsot az adott művelethez tartozó logikával

A 2. pont formalizációjához először vezessük be az `&&&` operátort, amely két függvényt hattat egy értékre és a kimeneteikből egy párt készít:

```
(&&&) :: (a -> b) -> (a -> c) -> a -> (b, c)
(&&&) f g a = (f a, g a)
```

Ekkor a paramorfizmus (az identitás függvényt `id`-vel jelölve):

```
para :: RAlgebra f a -> fix f -> a
para rAlg = rAlg . fmap (id &&& para rAlg) . out
```

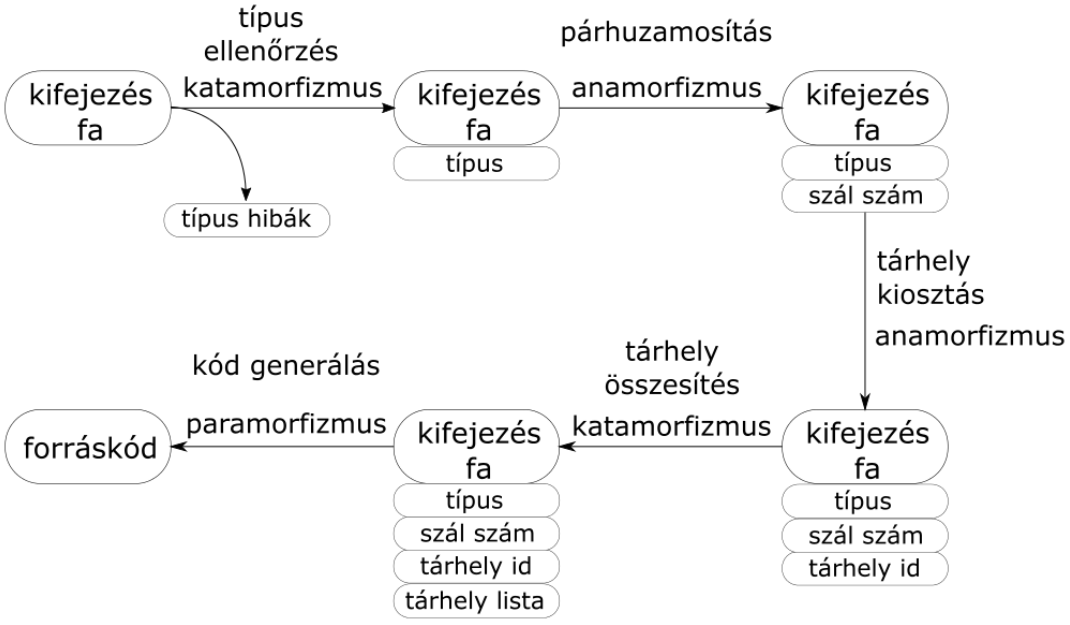
A paramorfizmus kategóriaelméleti duálisa az apomorfizmus, de ezt a sémát nem használjuk a jelenlegi megvalósításban.

3. A LambdaGen rendszer

3.1 Áttekintés

A LambdaGen a bevezetésben bemutatott absztrakció implementációja, flexibilisen használható műveleteket és magasabb dimenziós adatstruktúrát kínál lineáris algebrai számítások felírásához. A neve abból adódik, hogy a bemenete funkcionális primitívekkel írható le, az eredményt pedig generatív programozással állítja elő.

A felhasználó egy Haskellbe ágyazott nyelven írhat le alapműveletekből és funkcionális primitívekből álló kifejezéseket. A felépített kifejezés fán először típusellenőrzést végzünk. Ez után felmérjük, hogy mely műveleteket célszerű párhuzamosan végrehajtani, minden művelet eredményéhez tároló helyet rendelünk, majd összesítjük a számítás memória igényeit. Végül generálunk egy C++ forráskódot, amely az eredetileg felírt számítás hatékony párhuzamos kiértékelését valósítja meg.



Ábra 10 - A LambdaGen transzformációs lépései

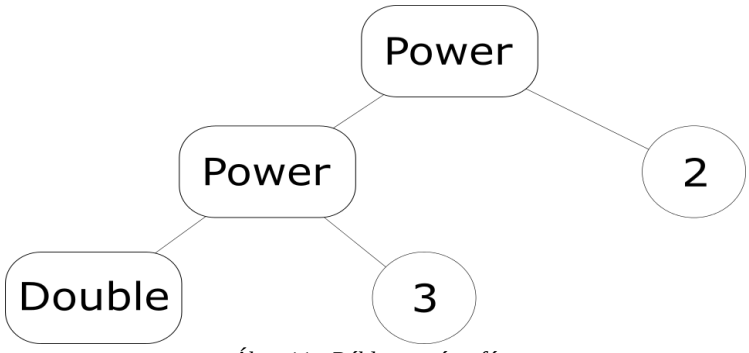
3.2 Típusok

Minden kifejezésnek van típusa, amit vagy a felhasználó ad meg, vagy kikövetkeztetjük az operandusaiból. A méretek is kódolva vannak a típusokban, így a típus ellenőrzés biztosítani tudja a kompatibilitást a műveletek és operandusaik között.
A LambdaGen primitív típusai:

- **double**: a (lebegőpontos) skalárok típusa
- **dim n**: vektor kiterjedése egy dimenzióban (csak power típus második argumentuma lehet)
- **power a b**: *a* típusú elemekből álló *b* méretű vektor (*b* csak dim lehet)
- **arrow a b**: az *a* típust *b*-re képző függvény

Összetett típusokat a fenti primitívek komponálásával állíthatunk elő.
Például egy 2x3-as lebegőpontos mátrix típusa, vagyis egy 2 elemű vektor, amelynek minden eleme egy 3 lebegőpontos számból álló vektor:

```
power (power double (dim 3)) (dim 2)
```



Ábra 11 - Példa egy típusfára

Fontos, hogy a típusok is fát alkotnak, mert így alkalmazhatók rajtuk a 2.5 fejezetben ismertetett transzformációs módszerek.

3.3 Műveletkészlet

A kifejezés fák felírásához biztosított műveletek meghatározásakor arra törekedtünk, hogy azok

- skálázódjanak a dimenzió szám emelkedésével
- könnyen komponálhatók és optimalizálhatók legyenek
- alkalmasak legyenek lineáris algebrai kifejezések felírására, de
- a könnyű megjegyezhetőség érdekében minél kevesebb művelettel

A lineáris algebrából természetesen adódtak a szám konstans (**scl**), skalár összeadás (**add**) és szorzás (**mul**), és a vektorképzés (**vec**) műveletek. A felhasználónak tudnia kell hivatkozni a memóriában tárolt nagyméretű adataira, erre bevezettük a vektor nézetet (**vecView**).
Bevettük az építő elemek közé a lambda kalkulusból a lambda absztrakciót (**lam**, **var**) és alkalmazást (**app**), ezzel lehetővé téve tetszőleges függvények bevezetését.
Magasabb dimenzióban végzett számításokhoz használhattuk volna az ismert lineáris algebrai vektor és mátrix műveleteket, de ezekből nem lehet flexibilisen előállítani újakat, ahogy azt a Bevezetésben bemutattuk.
A funkcionális programozásból ismert **map**, **reduce** és **zipWith** magasabbrendű függvények viszont jól skálázódnak az argumentumok dimenzió számával és könnyen optimalizálhatók, így inkább ezeket választottuk a vektorszámításokhoz.
Az összeállított műveletkészlet tehát:

- **scl x**: Lebegőpontos skalár konstans *x* értékkel. Típusa `double`.
- **add a b**: Skalár összeadás. *a*, *b* és az eredmény is `double` típusú.
- **mul a b**: Skalár szorzás. *a*, *b* és az eredmény is `double` típusú.
- **vec [e_1,...,e_n]**: Az e_i elemekből álló vektor. Ha az e_i-k típusa azonosan `t`, az eredmény típusa `power t n`.
- **vecView id [d_1,...,d_n]**: Az *id* azonosítójú memória címre hivatkozó d_1 x … x d_n-es tenzor. Típusa `power( … power double (dim dn) … ) (dim d1)`.
- **var id t**: Egy lambda függvény *id* azonosítójú, *t* típusú változója.
- **lam v t**: Lambda absztrakció, amely a *v* változót köti a *t* kifejezésben. Ha *v* és *t* típusa rendre `a` és `b`, akkor az eredmény `arrow a b` típusú.

- **app l e**: Az *l* lambda alkalmazása az *e* kifejezésre. Ha *l* típusa `arrow a b` és *e* típusa `a`, akkor az eredmény `b` típusú.
- **map f v**, **reduce f v**, **zipWith f v1 v2**: A 2.1 fejezetben bemutatott magasabbrendű függvények alkalmazása.

A következő táblázat összefoglalja a magasabbrendű (azaz függvény paraméterű) függvények típus követelményeit, eredmény típusát és értékét. Az egy soron belül azonos betűvel jelölt típusoknak meg kell egyezniük.

Művelet	**Paraméterek típusa**	**Eredmény típusa**	**Eredmény**
app(f, x)	arrow(a,b), a	b	f(x)
map(f, $[x_1,\ldots,x_n]$)	arrow(a,b), power(a,n)	power(b,n)	$[f(x_1),...,f(x_n)]$
reduce(f,$[x_1,\ldots,x_n]$)	arrow(a,arrow(a,a)), power(a,n)	a	x_1 (n = 1) reduce(f,$[f(x_1,x_2),\ldots,x_n]$) (n > 1)
zipWith(f, $[x_1,\ldots,x_n]$, $[y_1,\ldots,y_n]$)	arrow(a,arrow(b,c)), power(a,n), power(b,n)	power(c,n)	$[f(x_1,y_1),...,f(x_n,y_n)]$

Táblázat 1 - A magasabbrendű függvények szignatúrái

A fenti listában látható, hogy a **lam** műveletnek csak egy változót lehet megadni paraméterként. Több változós függvényeket curryzve lehet felírni, ahogy az alábbi műveletekben is látható. A függvények tárolásának ez a modellje teszi lehetővé, hogy támogassuk a parciális függvényalkalmazást, azaz hogy egy függvény néhány változóját rögzítve egy kisebb aritású függvényhez jussunk.

Nem nyilvánvaló, hogy ezekkel a műveletekkel tényleg kifejezhetők általános lineáris algebrai számítások, ezért ezt a következőkben példákkal támasztjuk alá.

3.3.1. Skalár műveletek

Az `add` és a `mul` műveletek magukban nem lambda kifejezések, de könnyen azzá tehetők, ha becsomagoljuk őket:

```
x = var "x" double
y = var "y" double
sclAdd = lam x (lam y (add x y))
sclMul = lam x (lam y (mul x y))
```

3.3.2. Vektor műveletek

```
v1 = var "v1" (power double (dim d))
v2 = var "v1" (power double (dim d))
```

vektor skalárszorosa:

```
sclVecMul = lam x (lam v (map (app sclMul x) v))
```

két vektor összege:

```
vecAdd = lam v1 (lam v2 (zipWith sclAdd v1 v2))
```

két vektor skaláris szorzata:

```
dotProd =
lam v1 (lam v2 (reduce sclAdd (zipWith sclMul v1 v2)))
```

két vektor diadikus szorzata:

```
outerProd =
lam v1 (lam v2
     (map
       (lam x (app (app sclVecMul x) v2))
       v1))
```

3.3.3. Mátrix műveletek

mátrixösszeadás:

```
m1 = var "m1" (power (power double (dim b)) (dim a))
m2 = var "m2" (power (power double (dim b)) (dim a))
matAdd = lam m1 (lam m2 (zipWith vecAdd m1 m2))
```

mátrixszorzás skaláris szorzattal:

```
m1 = var "m1" (power (power double (dim b)) (dim a))
m2 = var "m2" (power (power double (dim b)) (dim c))
v = var "v" (power double (dim b))
matMulS =
lam m1 (lam m2
     (map (lam v (map (app dotProd v) m2)) m1))
```

mátrixszorzás diadikus szorzattal:

```
m1 = var "m1" (power (power double (dim a)) (dim b))
m2 = var "m2" (power (power double (dim c)) (dim b))
matMulD =
lam m1 (lam m2
     (reduce matAdd (zipWith outerProd m1 m2)))
```

3.4 Transzformációk

3.4.1. Típus ellenőrzés

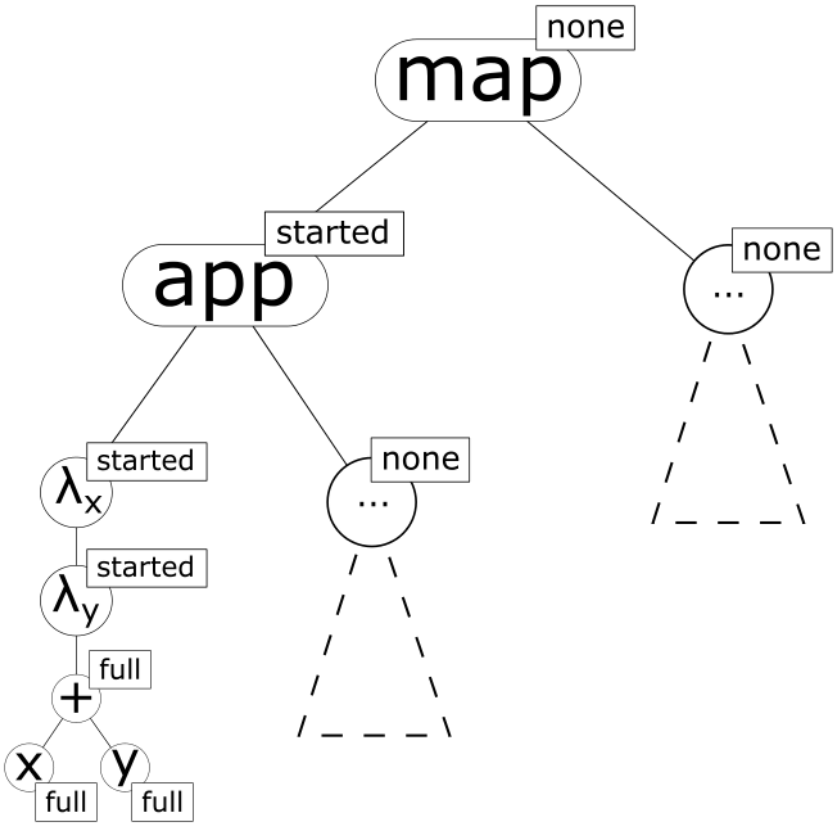
A felhasználó által felírt kifejezésfa szintaktikus helyességét biztosítják a felépítéshez használt segédfüggvények és maga a fa szerkezete, a szemantikus helyességet pedig egy algebrával tudjuk ellenőrizni. A típus ellenőrző algebra minden csúcsban megvizsgálja, hogy az operandusok típusai megfelelnek-e a 3.3 fejezetben leírt követelményeknek. Ha megfelelnek, akkor a transzformáció eredménye a csúcs így már egyértelmű típusa. Ha pedig az algebra típus hibát talál, akkor a hiba okára utaló szöveget ad vissza.
Ha egy csúcs típushibás, akkor az őseiben már nincs értelme a típusellenőrzésnek, ezért azt csak helyes gyerekek esetén tesszük meg.

3.4.2. Párhuzamosítás

A map, reduce és zipWith függvényeket érdemes párhuzamosan kiértékelni, ahogy azt a 2.1 fejezetben kifejtettük. Ilyenkor annyi új szálat indítunk, ahányat maximálisan támogatni tud az architektúra. Így ha egy műveletet párhuzamosítunk, akkor annak a részfájában már nem indíthatunk új szálakat, mert ezzel túllépnénk a támogatott szálak számát.
Úgy döntöttünk, hogy a párhuzamosítandó műveleteket egy mohó algoritmussal választjuk ki: a gyökértől lefelé bejárjuk a művelet fát, és minden magasabbrendű függvényt

párhuzamosítunk, amelyiknek még nincs párhuzamosított őse. Ezzel a lehető legmagasabban osztjuk szét a kiértékelést, és így a lehető legtöbb számítást fogják a párhuzamos szálak elvégezni.

Néhány esetben azonban lehetséges, hogy egy párhuzamosított művelet leszármazottját is lehet párhuzamosítani. Az alábbi példa mutatja, hogy egy párhuzamosított magasabbrendű függvény lambdáját kaphatjuk részleges alkalmazás eredményeként is. Ilyenkor az alkalmazást csak egyszer kell kiértékelnünk, a magasabbrendű függvény kiértékelése előtt, vagyis az alkalmazás argumentum részfájában lévő kifejezést még tetszőlegesen párhuzamosíthatjuk. Hasonló a helyzet a magasabbrendű függvény tömb paraméterével: a függvény előtt kerül kiértékelésre, így nincs korlátozva a párhuzamosítása.



Ábra 12 - A műveletek jelölése párhuzamosításhoz

Ezeket az eseteket háromértékű jelölés alkalmazásával kezeljük. Egy művelet a párhuzamosítás szempontjából lehet nem párhuzamos (`none`), megkezdett (`started`) és teljesen párhuzamos (`full`).

A fentről lefelé bejárást végző párhuzamosító koalgebra a következő szabályok szerint számítja ki az aktuális csúcs gyerekeinek jelölését:

- **map** | **reduce** | **zipWith** `none` jelöléssel: ezt a csúcsot párhuzamosítani fogjuk, a lambda gyereke `started`, a tömb gyereke `none` jelölést kap
- **app** `started` jelöléssel: a lambda gyereke `started`, az argumentum gyereke `none` jelölést kap
- **lambda** `started` jelöléssel: ha a gyereke lambda, akkor `started`, különben `full` jelölést kap
- minden más eset: a gyerekek jelölése megegyezik az aktuális csúcséval

Emellett a koalgebra feljegyzi minden csúcshoz, hogy hány szálon fog kiértékelődni, amire tárhelyek előre foglalásánál és a kódgenerálásnál lesz szükség. Ez úgy történik, hogy a felhasználó a párhuzamosító koalgebra paramétereként megadja, hogy hány szálra szeretné szétosztani a számítást. A szál szám annotáció értéke ez a szám lesz azoknál a csúcsoknál, amik a bejárás során `full` jelölést kaptak, és 1 lesz az összes többi csúcsnál.

3.4.3. Tárhely kiosztás

Egy számítás kiértékelésekor a be- és kimeneten kívül a részeredmények tárolásához is szükség lehet számottevő mennyiségű memóriára. A legtöbb videokártya nem támogatja a dinamikus memóriafoglalást, így a kernelek futtatása előtt allokálnunk kell minden memóriát, amihez előre tudnunk kell az egyes részeredmények méretét. Szerencsére ez kódolva van a típusukban, így könnyen meg tudjuk határozni. Azonban egymásba ágyazott függvényeknél sok esetben nem kell minden részeredményhez külön memóriát foglalnunk. Jó példa erre a már bemutatott mátrix összeadás.

```
matAdd = lam m1 (lam m2 (zipWith vecAdd m1 m2))
```

A naiv megoldás itt külön memóriát foglalna a sor párok összegeinek (amit utána átmásolna az eredmény mátrix megfelelő sorába), és ugyanígy a vektorösszegzésben az egyes skalár összegek eredményeinek is. Összességében hárommátrixnyi memóriát használna a bemeneten kívül, pedig a számítás megvalósítható pusztán az eredmény mátrixszal.
A felesleges memória allokációt és másolást úgy kerülhetjük el, hogy a függvényeknek az argumentumaik mellett az eredmény tárolására szánt memóriát is átadjuk. A fenti példában

ez azt jelenti, hogy a külső zipWith megkapja az eredmény mátrix egy sorát, és oda állítja elő a vektor összeget. Ezt pedig úgy éri el, hogy tovább adja a skalár összeadásnak a kapott eredmény vektor egy-egy elemét.

Egy részeredmény tárhely igénye háromféle lehet: vagy a szülő csúcs memóriáját használja (`inherit`), vagy új memóriát kell allokálni (`prealloc`), vagy pedig egyáltalán nincs szükség hozzá külön memóriára (`implicit`).

Azt, hogy melyik művelet melyik kategóriába tartozik, a következő koalgebra dönti el:

- **scl**, **vecView**, **var**: `implicit`
- **vec, add, mul**: minden operandus `prealloc`
- **app** | **map** | **reduce** | **zipWith**: a lambda operandus `inherit`, az argumentum(ok) `prealloc`.

A tárhely kiosztáshoz tartozik egy katamorfikus (alulról felfelé bejáró) transzformáció is, ami a `prealloc`-kal annotált csúcsokban a típus és a párhuzamos szálak száma alapján meghatározza a szükséges memória méretét. Ezek az információk a gyökérben összegződnek, és a kódgenerálásnál lehetővé teszik az előzetes allokációt. A szálak száma azért lényeges, mert a több szálon futtatott számítások részeredményeinek szálanként különálló tárhely kell.

3.5 Annotációk

A fent bemutatott transzformációk különféle annotációkkal látják el a csúcsokat, és így mindegyik végrehajtása után változik a fa típusa. Egyes transzformációk megkövetelhetik bizonyos annotációk meglétét (például a tárhely kiosztáshoz szükség van a típus ellenőrzés során létrejövő típus annotációra), de torzítaná az absztrakciót (és lényegesen rontaná a kód újra felhasználhatóságát), ha a transzformációk a fa pontos típusától függenének.

Az annotációk feletti helyes absztrakciót a Vinyl [8] kiterjeszthető rekord csomag használatával értük el. A Vinyl segítségével

- paraméterezni tudjuk az algebrák és koalgebrák annotáció halmazát
- megkötéseket tudunk tenni, hogy bizonyos annotációk szerepeljenek a halmazban
- lencsék segítségével ki tudjuk olvasni a megkövetelt annotációk értékét

Mint azt a 2.5 fejezetben kifejtettük, a kata- és a paramorfizmus egyetlen értékké bontja le a fát. Ha például a típus ellenőrző algebrát az eredeti formájában alkalmaznánk, ahol a carrier type maga a kikövetkeztetett típus, akkor a transzformáció eredményeként egyetlen típust, a gyökér csúcs típusát kapnánk. Mi azonban a legtöbb esetben minden csúcsra szeretnénk megőrizni a transzformáció aktuális értékét, vagyis a csúcsot ezzel az értékkel szeretnénk annotálni.

Erre a célra alkottunk egy algebra átalakító függvényt, ami egy hagyományos algebrából egy annotáló algebrát készít. Ezt úgy éri el, hogy az aktuális csúcs már annotált gyerek csúcsaiból csak az átalakítandó algebra eredményeként született annotációt tartja meg, az így átalakított csúcson végrehajtja az eredeti algebrát, majd az eredményt hozzáfűzi a csúcs már meglévő annotációihoz.

Ezzel még tovább egyszerűsítettük a transzformációs algebrák használatát, mert elég a kiszámítandó értékkel foglalkoznunk, az automatikusan bekerül a csúcsok annotációi közé.

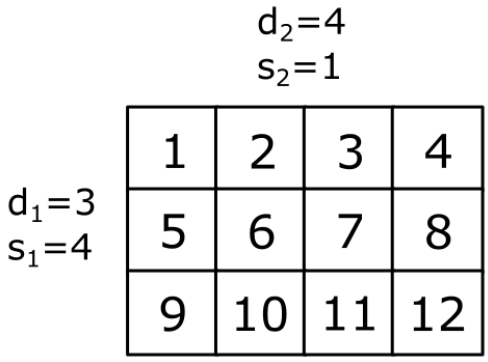
3.6 Adat modell

A kiértékelés során a több dimenziós tenzorokkal való hatékony számoláshoz elengedhetetlen, hogy tudjunk elemmozgatás nélkül, konstans időben dimenzió átalakításokat végezni a generált C++ kódban. Ilyen átalakítások a dimenziók sorrendjének megváltoztatása (a mátrix transzponálás általánosítása), vagy egyes dimenziók szerinti nézetek, szeletek készítése. A mátrixszorzásnál például az egyik mátrixra sorok vektoraként, a másikra oszlopok vektoraként kell tekintenünk, és csak akkor lehet hatékony a számításunk, ha ez nem jár plusz költséggel.[2]

Ennek a problémának a megoldására adat nézeteket vezettünk be. Egy adat nézet tartalmaz egy mutatót az adat memóriabeli helyére, a reprezentált tenzor méreteit az egyes dimenziókban, és a dimenziókhoz tartozó lépésközöket. Egy dimenzió lépésköze azt mutatja meg, hogy egy elemre mutató pointert mennyivel kell léptetni, hogy az adott dimenzió szerint következő elemet kapjuk. Például egy sorfolytonosan ábrázolt 3x4-es mátrix második dimenziójához tartozó lépésköz az ábrázolásból következően 1, az első dimenziójához

[2] A memória nem folytonos olvasásából adódó költségtől most eltekintünk.

tartozó pedig 4, mert pont egy sort kell átlépnünk a memóriában, hogy egy elem “alatti” elemet kapjuk.



Ábra 13 - Egy adatnézet lépésközei

Az adatnézeteket a dimenziók sorrendjében indexelhetjük, egy b mutatónál kezdődő n dimenziós nézet $[idx_1, \ldots, idx_n]$ indexéhez a $b + \sum s_i \; idx_i$ elem tartozik.

A lépésközök jelentősége abban van, hogy ha most a dimenziók sorrendjét a $[p_1, \ldots, p_n]$ permutációval szeretnénk módosítani, akkor elegendő egy új nézetet létrehoznunk változatlan kezdő mutatóval, és a (d_i, s_i) párok p permutációjával. A permutált reprezentációban az $[idx_{p_1}, \ldots, idx_{p_n}]$ indexű elem $(b + \sum s_{p_i} \, idx_{p_i})$ az összeg kommutativitásából adódóan megegyezik az eredeti reprezentáció $[idx_1, \ldots, idx_n]$ elemével, tehát a transzponálás helyes.

Fontos, hogy egy nézetet részlegesen is indexelhetünk: Ha a dimenziók számánál kevesebb indexet adunk meg, akkor természetes módon hozzájutunk a teljes tenzor egy alacsonyabb dimenziójú nézetéhez. Ezt a tulajdonságot ki is használjuk a részeredmények tároló helyének származtatásakor, ahogy azt a 3.4.3 fejezetben leírtuk.

3.7 Kódgenerálás

A felhasználó által megadott számítás kiértékeléséhez egy modern C++ programot készítünk. A nyelvválasztást az motiválja, hogy itt hatékony hardver kihasználás mellet viszonylag magas absztrakciós szint érhető el a C++ 11-es és 14-es szabványaiban bevezetett újdonságoknak köszönhetően. A generált forráskód egyetlen függvényt tartalmaz, ami paraméterként a memóriában lévő adatokra mutató pointereket vár, visszatérési értéke pedig

egy nézet az eredményre. A kódhoz tartoznak még a nézet osztály és a magasabbrendű függvények implementációit tartalmazó header fájlok.
A függvény elején vannak deklarálva és lefoglalva a részeredmények számára a tárhelyek.
A kódgenerálást végző R-algebra minden műveletből egy programkód darabot készít, ami az adott műveletet a hozzá rendelt tárolóhelyre értékeli ki. Azért van szükség R-algebrára (melyeket a 2.5.3 fejezetben mutattunk be), mert egy csúcs transzformációjakor a gyerekekből generált kód mellett az eredeti gyerek csúcsok tárhelyazonosító annotációjára is szüksége van, hogy fel tudja használni a kiértékelt részeredményeket. A paramorfizmus és az algebrák kifejező erejét jól demonstrálja, ahogy a kódgenerálás elsőre összetettnek tűnő feladata természetesen bomlik le az egyes műveletekhez szükséges rövid szövegekre és ezek összeállítására.
Tervezési szempontból érdekes a lambda absztrakcióból generált kód. Ez a lambda kalkulusbeli fogalom messze áll a régebbi C++ imperatív világától, de a C++14-ben bevezetett generikus lambda kifejezés kiválóan modellezi. Nem kell kiírni a bemenet és a kimenet típusát, a változó láthatósága pont az, amit várunk, és bármi külön módosítás nélkül hattatható vagy paraméterül adható egy magasabbrendű függvénynek.

A könyvtár használatához szükséges hardver és szoftver környezetről a függelék tájékoztat.

4. Eredmények

A korábban megvalósított FParLin könyvtár [6] hasonló funkcionalitást nyújt C++ nyelven. A most bemutatott Haskell implementáció kódja

- áttekinthetőbb, a magas szintű absztrakciók is tömören kifejezhetők, kevesebb a szintaktikus zaj
- rövidebb (a típus ellenőrzés például 250 helyett csak 120 kódsor)
- megbízhatóbb a nyelv kifejezőbb típusrendszeréből és tisztán funkcionális voltából adódóan

A különböző annotációkat tartalmazó kifejezés fák általánosítására jó absztrakciót nyújtanak a Vinyl könyvtár kiterjeszthető rekordjai. Az általunk bevezetett annotációs algoritmus pontosan modellezi az annotációk előállításának követelményeit, hiszen meg tudjuk követelni bizonyos annotációk meglétét, és el tudunk vonatkoztatni a többitől.

A bevezetett műveletekkel tudtuk például 3 és 2 dimenziós tenzorok szorzását implementálni, vagyis tényleg jól használhatók magasabb dimenziós kifejezések felírására.

A mátrix-vektor szorzás a szakdolgozat [6] 1.668 másodperces idejéhez képest 0.469[3] másodperce gyorsult annak köszönhetően, hogy a részeredményeket nem másoljuk, hanem egyből a végleges helyükre kerülnek.

A rekurziós sémák használata keretet ad a transzformációk kigondolásához és egyszerű, kézenfekvő implementálásához. Az így készült algoritmusok nem csak átláthatóbbak, de az explicit típuskövetelményeknek köszönhetően sokkal könnyebb helyesen megvalósítani őket. Ennek köszönhető, hogy a fejlesztés során minimális időt kellett csak hibakereséssel tölteni.

[3] Intel Core i5 processzoron, 4 szálon, 8 GB memóriával futtatva

5. Összefoglalás

A dolgozatban bemutatott eredmények röviden a következők:

- sikeresen alkalmaztuk a rekurziós sémákat a kifejezésfák elemzésének, annotációjának és átalakításának magas szintű absztrakciójaként
- bemutattuk, hogy a map, reduce és zip függvények alkalmasak tetszőleges dimenziójú lineáris algebrai számítások kifejezésére
- demonstráltuk az automatikus párhuzamosítás és a generált programmal történő kiértékelés megvalósíthatóságát
- kidolgoztuk és megvalósítottuk a másolások minimalizálását és a dinamikus allokáció teljes mellőzését lehetővé tevő memóriakezelést

A kutatás soron következő fázisában a bemutatott Haskell könyvtárat szeretnénk kiterjeszteni a következő funkciókkal:

- a magasabbrendű függvények blokkosított kiértékelése, ami elengedhetetlen a hatékony GPU programozáshoz
- GPU programkód generálása
- az átmeneti tárhelyek minimalizálása a map-reduce-zip függvények összeolvasztásával (amely több tekintetben hasonlít a funkcionális programozásban ismert stream fusion-re [10])

A könyvtár forrása elérhető a https://github.com/leanil/LambdaGen címen.

6. Hivatkozások

1. *Basic Linear Algebra Subprograms.* [Online] [Hivatkozva: 2016. 11. 30.] http://www.netlib.org/blas/.

2. *Eigen.* [Online] [Hivatkozva: 2016. 11. 30.] http://eigen.tuxfamily.org/.

3. *Armadillo.* [Online] [Hivatkozva: 2016. 11. 30.] http://arma.sourceforge.net/docs.html#Cube.

4. MultiArray. *Boost.* [Online] [Hivatkozva: 2016. 11. 30.] http://www.boost.org/doc/libs/1_62_0/libs/multi_array/doc/.

5. *Blitz++.* [Online] [Hivatkozva: 2016. 11. 30.] http://blitz.sourceforge.net/resources/blitz-0.9.pdf.

6. Leitereg, András. *Általános vektorműveleti fák párhuzamosított kiértékelése.* 2016.

7. Kmett, Edward A. *recursion-schemes.* [Online] [Hivatkozva: 2016. 11. 30.] https://hackage.haskell.org/package/recursion-schemes-5.

8. Sterling, Jonathan. *vinyl.* [Online] [Hivatkozva: 2016. 11. 30.] https://hackage.haskell.org/package/vinyl.

9. *Functional Programming with Bananas, Lenses, Envelopes.* Meijer, E., Fokkinga, M. és Paterson, R. New York : Springer-Verlag, 1991.

10. *Stream Fusion. From Lists to Streams to Nothing at All.* Duncan Coutts, Roman Leshchinskiy, Don Stewart. ICFP’07 2007.

7. Függelék

7.1 Szükséges hardver és szoftver környezet

Az LambdaGen most bemutatott verziója processzor szálakat használ a párhuzamos végrehajtásra, ezért a használatával elért gyorsulás arányos a rendelkezésre álló processzor magok számával. Célszerű olyan processzorral használni, ami képes legalább 4 szál párhuzamos végrehajtására.

A könyvtár fordításához szükség van a Vinyl csomagra, és minimálisan a GHC 8-as verziójára.

A generált C++ kód fordításához C++14-et támogató fordítóra van szükség, azaz minimálisan g++ 5-re, clang 3.4-re vagy Visual Studio 2015-re.

A könyvtárat a következő környezetekben teszteltük:

- Intel Core i5 3210M
    - 2 magos processzor, Hyper-Threading miatt 4 szál
- 8GB memória
- Windows 10 és Ubuntu 16 operációs rendszer
- GHC 8.0.1
- clang 3.9 és Visual Studio 2015