EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
NUMERIKUS ANALÍZIS TANSZÉK

FParLin könyvtár

Témavezetők:

Dr. Gergó Lajos
egyetemi docens
Numerikus Analízis Tanszék

Berényi Dániel
Tudományos segédmunkatárs
Wigner Fizikai Kutatóközpont

Szerző:

Leitereg András
programtervező informatikus BSc

Budapest, 2016

Tartalomjegyzék

Bevezetés

A legtöbb reáltudományban szükség van lineáris algebrai számítások elvégzésére. Nagy szerepük van az adat analízisben, statisztikában, de a differenciál egyenletek 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. De a párhuzamos végrehajtásnak költsége is van, ami a feladatok szétosztásánál és az eredmények összegyűjtésénél jelentkezik. A gyakorlatban előforduló lineáris algebrai műveletek sokszor hierarchikusan egymásba ágyazva jelentkeznek, más algoritmusokkal kombinálva, akár több szinten keresztül. Egy ilyen összetett kifejezésben nem könnyű feladat megtalálni azokat a részkifejezéseket, amelyeket megéri párhuzamosítani, azaz a párhuzamos végrehajtás költsége lényegesen kisebb, mint a sorosé.

A szakdolgozat azt vizsgálja, hogy a fenti feladat megoldható-e automatikusan, program segítségével. A témát a Wigner Fizikai Kutatóközpont GPU Laborja (1) ajánlotta. Ők javasolták annak a kihasználását, hogy a magas szinten, funkcionálisan felírt vektor műveleti fák sokkal könnyebben analizálhatók és transzformálhatók, mint az imperatív kódok, hiszen sokkal explicitebbek az adat- és függvényfüggőségek. A párhuzamosítás könnyen integrálható magas szintű optimalizációs transzformációkkal és a felhasználó (programozó) részéről kevés szaktudással lényegesebben jobb kódok generálhatók. A fő cél nem a maximális hatékonyság, hanem a fejlesztési- és futásidő megfelelő kompromisszum mellett történő minimalizálása.

A dolgozat keretében elkészített FParLin könyvtár elsődleges célja a fenti megközelítés megvalósíthatóságának vizsgálata, az implementáció során felmerülő nehézségek feltérképezése. A párhuzamos végrehajtáshoz csak processzor szálakat használ, de kiterjeszthető videokártyák, klaszterek vagy más, párhuzamos végrehajtást lehetővé tévő platformok egyidejű használatához.

A program kétféle információt vár a felhasználótól. Az elvégzendő számítást reprezentáló kifejezés fát, amelynek levelei adatok, belső csúcsai pedig műveletek, valamint egy konstans értéket, ami a fent leírt párhuzamosítási költségnek felel meg. A kapott fán költségelemzést végez, vagyis minden belső csúcsra megbecsüli az adott művelet kiértékelésének költségét. Ezután a párhuzamosítási költség figyelembe vételével létrehoz egy olyan programot, amely a megadott kifejezés kiértékelését végzi azokat a részkifejezéseket párhuzamosítva, amelyeknél ez várhatóan lényeges gyorsulást eredményez.

A jelenleg elérhető eszközök a párhuzamosítás támogatására vagy nagyon alacsony szintűek (a felhasználónak kell megírnia a memóriakezelést és a szálak indítását, mert ezeket a primitíveket nyújtja az API), vagy túl magas szintűek (kész algoritmusok párhuzamos futtatását támogatják a klaszteren vagy a GPU-n, de itt is előáll a probléma, hogy egy bonyolult hierarchiában hol kellene valóban párhuzamosítani). Az előbbi kategóriába tartozik a legtöbb GPU API, például a SYCL, az utóbbinak pedig a Hadoop tipikus példája.

Felhasználói dokumentáció

1. Alkalmazási területek

Az FParLin egy C++ könyvtár, amely

- eszközöket nyújt összetett mátrix-vektor kifejezés fák felépítésére
- képes a felépített fán különböző elemzéseket végezni
- létrehoz egy függvényt, melynek segítségével a felhasználó tetszőleges adatokon kiértékelheti a kifejezést

A létrehozott függvény párhuzamosan számítja ki a kifejezésfa erre alkalmas műveleteit, de csak akkor, ha az előzetes költségbecslés ezt indokolttá teszi. Ennek az intelligens párhuzamosítási stratégiának köszönhetően a számítás várhatóan hatékonyabb lesz, mint ha minden műveletet párhuzamosan hajtunk végre, amit lehetséges.

Az olyan alkalmazási területeken, ahol

- a számítás hatékonysága kiemelkedően fontos (High Performance Computing - HPC)
- a kiszámítandó kifejezések túl összetettek ahhoz, manuálisan keressük meg az optimális párhuzamosításhoz tartozó műveleteket
- a fejlesztők kevésbé képzettek a gyakorlati párhuzamos/alacsony szintű programozás terén, de jól képzettek a magasabb szintű, matematikai, algoritmikus területeken

érdemes lehet az elterjedt párhuzamosságot támogató könyvtárak helyett az FParLin-t használni.

2. Felhasznált eszközök és módszerek

A HPC alkalmazások jellemzően a C++ nyelvet használják a hatékony hardver kihasználás mellett elérhető viszonylagosan magas absztrakciós szint miatt. A

könnyű beépíthetőség érdekében így az FParLin is C++-ban készült. Nagyban épít a C++11-es és 14-es szabvány által bevezetett újdonságokra.
A kifejezésfa elemzéseket és transzformációkat F algebrák C++ implementációival írjuk le a minél magasabb szintű absztrakció érdekében. Az F algebrákban a fákat összegtípusokkal (logikai vagy kapcsolattal) írjük fel rekurzívan, amelyek reprezentálására a Boost könyvtár (2) variant osztályát használjuk. Az F algebrák, Funktorok és egyéb kategóriaelméleti fogalmak ismeretére nincs szükség a könyvtár használatához, de az 5.3 fejezet részletesen leírja őket.

3. Szükséges hardver és szoftver környezet

Az FParLin 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 felhasznált eszközöknél említett `boost::variant` osztályt a felhasználó nem használja közvetlenül, de a kifejezés fák felépítéséhez szükség van rá, ezért fordításkor elérhetőnek kell lennie a Boost könyvtárnak. A könyvtár fejlesztése a dolgozat elkészítésekor legfrissebb 1.60.0 verzióval történt, ezért ez a javasolt verzió.
Mivel az FParLin és a Boost is standard C++-ban készült, az egész alkalmazás bármilyen platformon használható, amire létezik C++ fordító. Sikeresen teszteltem Linux operációs rendszereken g++ és clang fordítókkal, valamint Windowson Visual C++-szal és clang-gal, így ezek az ajánlott platform-fordító kombinációk. Az operációs rendszer és fordító verziókat a Tesztelés fejezet írja le.
Az FParLin és a Boost is használja a modern C++ szabványokat, ezért a használathoz minimálisan szükséges fordító verziók: g++ 5, clang 3.4 és Visual Studio 2015.

4. Az FParLin használata

A következő alfejezetek feltételezik a C++ nyelv használatában való jártasságot, mert ez elengedhetetlen az FParLin (mint C++ könyvtár) tényleges használatához. Azok számára, akik csak kipróbálni szeretnék, készítettünk egy bemutató projektet Eval néven. Ez tartalmazza egy alkalmazás azon részeit, amiket a felhasználónak kellene megírnia a saját céljainak megfelelően. Használatát a projekt könyvtárában található `readme.txt` fájl írja le.

4.1 Előkészítés

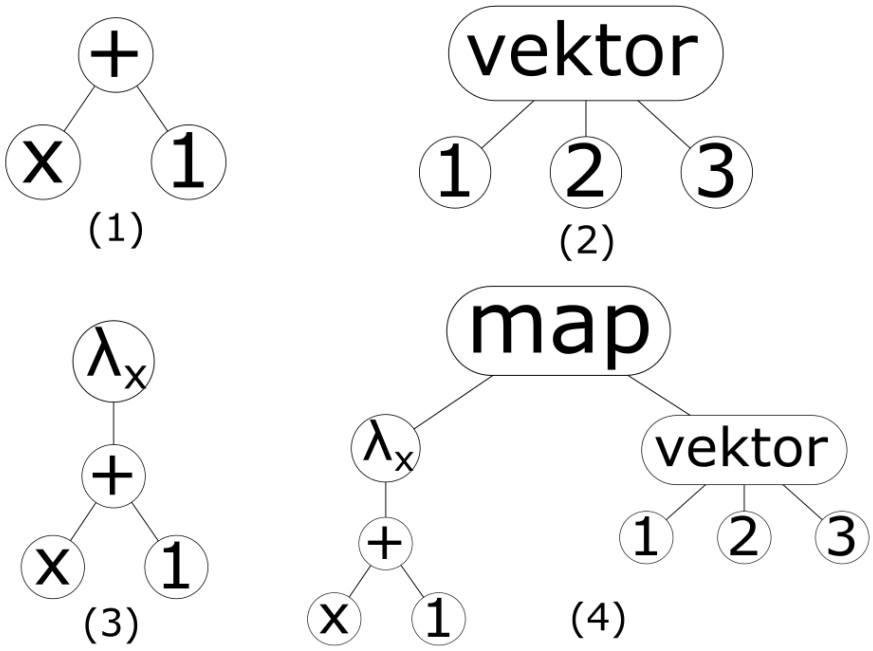
A kifejezésfa felépítéséhez szükséges szimbólumok deklarációját az `expr.h` és a `type.h` header fájlok, a tényleges végrehajtást elindító függvényt pedig az `evaluator.h` tartalmazza. A felhasználónak include-olnia kell ezeket a könyvtár használatához.
A program munka könyvtárában el kell helyezni az `fparlin.h` és a `config.txt` fájlokat. Az előbbi az FParLin által generált programnak nyújt segédfüggvényeket. Az utóbbi pedig egy konfigurációs fájl, amelynek első sorába a program fordítására használt fordítóprogram elérési útját kell írni. A Visual C++ konzolos futtatása előtt be kell állítani a környezeti változókat a vcvarsall.bat futtatásával, ezért ha cl.exe-t írunk az első sorba, akkor a második sorban meg kell adni a batch fájl elérési útját is. Egy tipikus rendszeren a `config.txt` tartalma ilyen lehet:
Windows:
"C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\x86_amd64\cl.exe"
"C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\vcvarsall.bat"
Linux:
"/usr/bin/g++"

4.2 Kifejezés fák felépítése

A kiszámítandó kifejezést egy fa felépítésével kell megadni, melynek levelei konstans értékek vagy változók, belső csúcsai pedig műveletek. Egy ilyen fa

felépítésének menetét mutatja be a következő ábra. Alkalmazzuk az [1,2,3] vektor minden elemére az x→ x+1 függvényt. Az eredmény természetesen a [2,3,4] vektor.

Az x→ x+1 függvény hozzárendelést a következő ábrán látható módon reprezentáljuk kifejezés fával. A későbbiekben függvények definíciójára használjuk még a lambda-kalkulus jelölésrendszerét (λx.x+1), ami az irodalomban elterjedt, és forráskódban az FParLin-beli függvény absztrakció műveletet (`Lam(...,x,x+1)`) is.



1. Az x+1 kifejezés.
2. Az [1,2,3] vektor.
3. Az x→x+1 függvény, azaz egy lambda függvény az x változóval és x+1 helyettesítési értékkel. A lambda-kalkulust külön fejezet mutatja be bővebben a Fejlesztői dokumentációban.
4. A map művelet a kapott egy változós függvényt alkalmazza a kapott vektor minden elemére, az eredmény a függvényértékekből álló vektor.

Látható, hogy a kifejezésfa bármely csúcsa által meghatározott részfa maga is egy helyes kifejezést ír le. Minden ilyen részkifejezésnek, azaz a fa minden csúcsának (explicite megadott vagy kikövetkeztetett) típusa van. Ez lehetővé teszi, hogy az

FParLin típus ellenőrzést végezzen, ami könnyen felfedi a felépített kifejezésben előforduló hibákat.

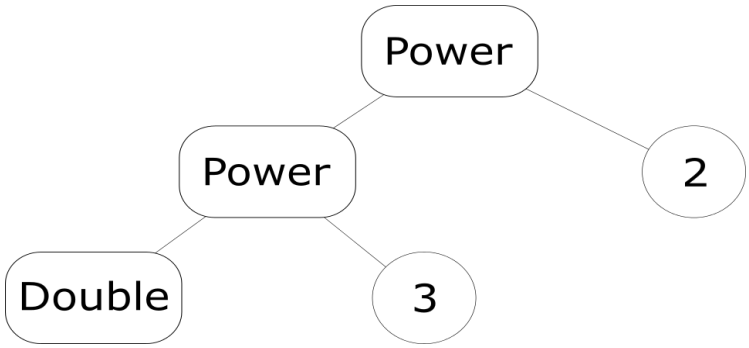
4.2.1. Típusok

Magukat a típusokat is fa szerkezettel lehet leírni. Ennek a fának a levelei primitív típusok vagy konstans egészek, belső csúcsai pedig típus műveletek. Egy típus megadásához, vagyis egy ilyen fa felépítéséhez a `type.h` header ad segédfüggvényeket. Mindegyik segédfüggvény visszatérési értéke egy típus fa, ami önmagában is egy érvényes típus, és összetett típusok felépítésére is használható.

- **Int()**: Primitív egész szám típus.
- **Double()**: Primitív lebegőpontos szám típus.
- **Size(x)**: Konstans egész, értéke x.
- **x_size**: Konstans egész literál, értéke x. Ekvivalens `Size(x)`-szel, egyszerűbb jelölést tesz lehetővé, ha x értéke ismert.
- **Power(a,b)**: Hatvány típus, egy *a* típusú elemekből álló *b* elemű tömb típusa. *b*-nek **size**-nak kell lennie.
- **Arrow(a,b)**: Függvény típus, a bemenet típusa *a*, a kimeneté *b*.

A kívánt típust a fenti függvények egymásba ágyazásával lehet felépíteni. Például egy 2x3-as lebegőpontos mátrix típusa, vagyis egy 2 elemű tömb, amelynek minden eleme egy 3 lebegőpontos számból álló tömb:

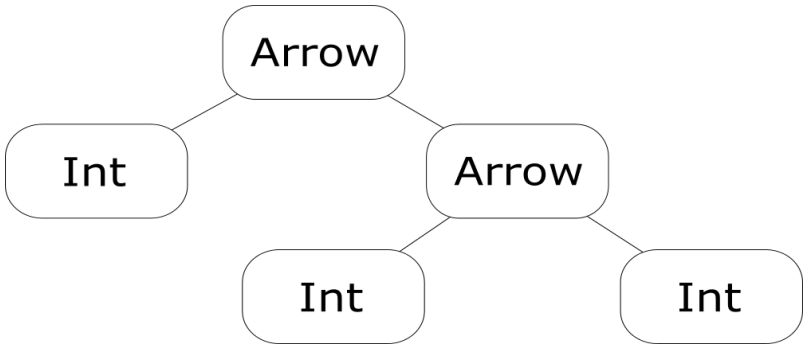
```
Power(Power(Double(), 3_size), 2_size)
```



A függvény típushoz tartozó `Arrow` függvény fix paraméter számú, csak egy változó típusát várja. Több változós függvényeket curryzéssel lehet leírni, amelyet

részletesen bemutat az 5.2.3 fejezet. Vizsgáljuk meg az (x,y) → x*y kétváltozós függvényt. Ez a függvény felfogható az x → (y → x*y) függvényként is, azaz egy olyan függvényként, ami x-hez azt a függvényt rendeli, ami y-hoz x*y-t rendel. A szorzás művelet curryzett alkalmazását így követhetjük végig: (’*’)(2,3) = ((’*’)(2))(3) = (2*)(3) = 6.

A fenti függvény típusának felírásakor is ugyan ezt a gondolatmenetet kell követnünk. A két egész számhoz egészet rendelő függvényt felírhatjuk egy olyan függvényként, ami egy egész számhoz egy egész → egész függvényt rendel: `Arrow(Int(), Arrow(Int(), Int()))`



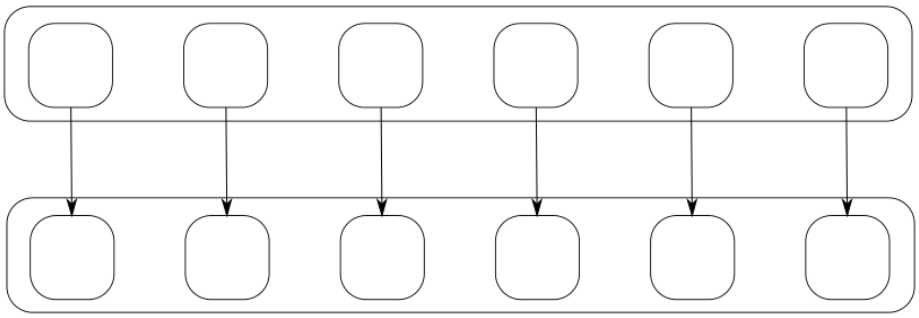
4.2.2. Értékek és műveletek

A kifejezés fát az `expr.h` header segédfüggvényeivel lehet felépíteni. Minden függvény visszatérési értéke egy helyes részkifejezést reprezentáló fa, ezek egymásba ágyazásával lehet felépíteni a kívánt kifejezést.

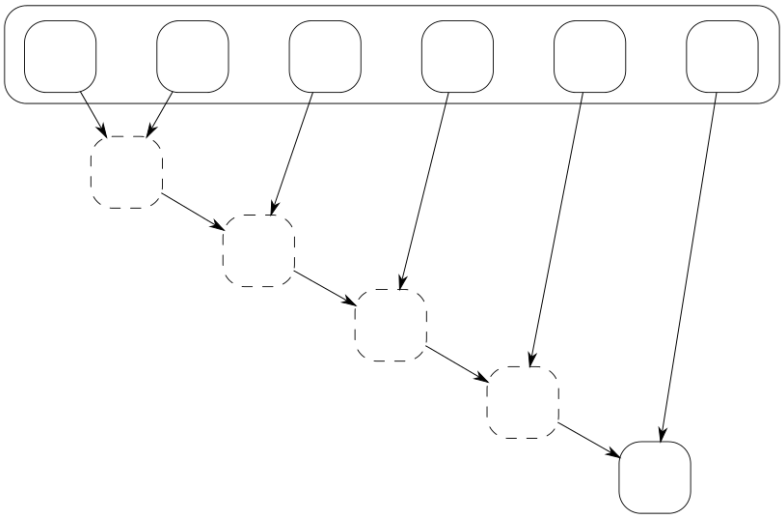
- **x_scl**: Skalár x értékkel, ami lehet egész vagy lebegőpontos. A típusa `Double()`. Az FParLin-ban jelenleg nincs mód `Int()` típusú skalár megadásra, ennek okát a 6.2.1 fejezet írja le.
- **VecView(név, méret)**: Hivatkozás (a név alapján) egy futtatáskor megadott, memóriában lévő adatvektorra. A vektor nézetekről bővebben a 4.4.1 fejezetben lesz szó. A műveletek szempontjából egy adott méretű lebegőpontos tömb, ennek megfelelően a típusa `Power(Double(), Size(méret))`.
- **Vec(elem lista)**: A megadott elemekből álló vektor. Az előzővel ellentétben ennek a vektornak teljes kifejezés fák az elemei. Követelmény, hogy az

elemek típusa megegyezzen. Ekkor a létrehozott fa típusa `Power(T, Size(s))`, ahol `T` az elemek típusa és `s` az elem lista mérete.

- **a + b**: Skalár összeadás, *a* és *b* skalár. Típusa `Int()`, ha *a* és *b* is `Int()`, különben `Double()`.
- **a * b**: Skalár szorzás, *a* és *b* skalár. Típusa `Int()`, ha *a* és *b* is `Int()`, különben `Double()`.
- **Var(típus, név)**: Egy lambda függvény változója. A típusa a megadott típus. A név egy tetszőleges karakter a változók megkülönböztetésére. Független lambdák változóinak lehetnek azonos neve, de egymásba ágyazott lambdák esetében a belső változója ilyenkor elfedi a külsőét.
- **Lam(visszatérési típus, változó, törzs)**: Lambda függvény (bővebben az 5.2 fejezetben). Jellemzően a törzs részfának legalább egy levele a megadott változó. A törzs típusának meg kell egyeznie a megadott visszatérési típussal. Ekkor a változó típusát `A`-val, a visszatérési típust `B`-vel jelölve a lambda részfa típusa `Arrow(A,B)`.
- **App(lambda, bemenet)**: A megadott lambda függvény alkalmazása a bemenetre. A lambda törzsében a lambda változójának összes előfordulása megkapja a bemenet értékét. Az így kapott törzs értéke lesz az App részfa értéke. Természetesen a lambdának függvény típusúnak kell lennie. Ha ez a típus az `Arrow(A,B)`, akkor a bemenet típusa kötelezően `A`, az eredmény típusa pedig `B` lesz.
- **Map(lambda, vektor)**: A megadott egy változós lambda függvény alkalmazása a vektor minden elemére. Az eredmény egy vektor, aminek minden eleme a bemeneti vektor megfelelő elemének a lambda alatti képe. $map(f,[x_1,\ldots,x_n])=[f(x_1),...,f(x_n)]$ A vektor elemeinek olyan típusúnak kell lennie, mint a lambda változójának típusa. Ha a lambda típusa `Arrow(A,B)` és a vektor típusa `Power(A,n)`, akkor a map típusa `Power(B,n)`.

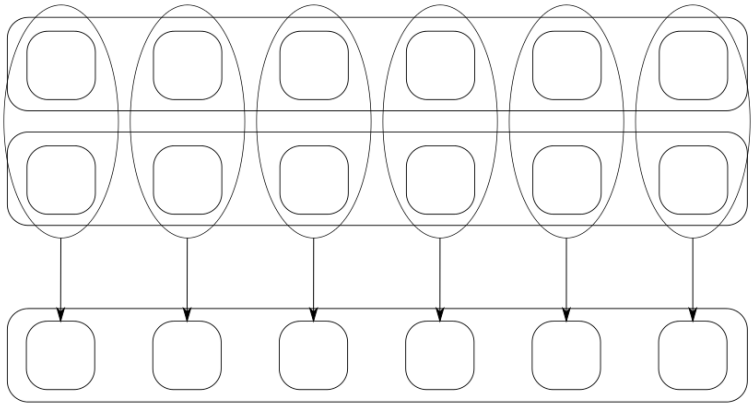


- **Reduce(lambda, vektor)**: Ez a művelet legegyszerűbben rekurzívan definiálható: reduce(f, [x_1]) = x_1 és reduce(f,[x_1,x_2,...]) = reduce(f,[f(x_1,x_2),...]). Tehát minden lépésben a kétváltozós lambda függvényen keresztül kombináljuk a vektor első két elemét, és ez lesz az új első elem. A művelet eredménye az utolsó függvény alkalmazás értéke. A reduce nem értelmezett üres vektorra, és párhuzamosításkor feltételezi, hogy a megadott lambda asszociatív. Látható, hogy a lambdának kétváltozós függvénynek kell lennie. Ez az 5.2.3 fejezetben leírt módon, curryzéssel érhető el. Ha a vektor `Power(A,n)` típusú, akkor a lambdának `Arrow(A, Arrow(A,A))` típusúnak kell lennie és az eredmény típusa `A`.



- **Zip(lambda, vektor_1, vektor_2)**: A map általánosítása kétváltozós függvényre. Az eredmény egy vektor, aminek minden eleme a bemeneti vektorok lambda függvény alatti elemenkénti képe.

zip(f,[x_1,…,x_n],[y_1,…,y_n])=[f(x_1,y_1),...,f(x_n,y_n)] A vektorok elemeinek olyan típusúnak kell lennie, mint a lambda változóinak típusa. Ha a lambda típusa `Arrow(A, Arrow(B,C))`, a vektorok típusa `Power(A,n)` és `Power(B,n)`, akkor a zip típusa `Power(C,n)`.

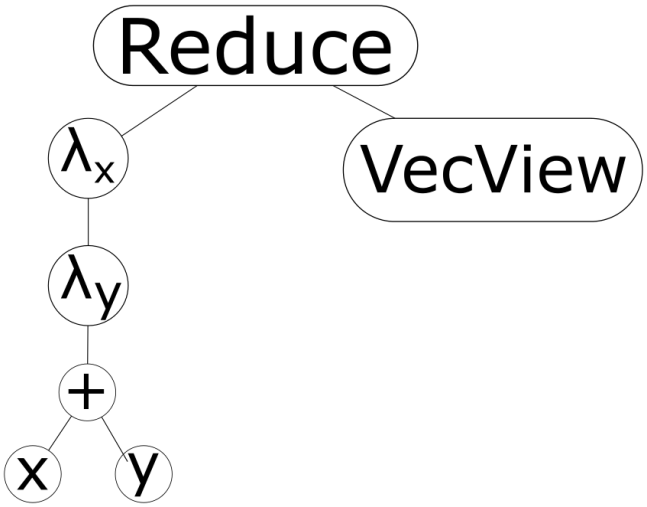


A következő táblázat összefoglalja a magasabb rendű (azaz függvény paraméterű) függvények típus követelményeit, eredmény típusát és értékét. Az egy soron belül azonos betűvel jelölt típusoknak meg kell egyezniük.

Művelet	**Paraméterek típusa**	**Eredmény típusa**	**Eredmény**
App(f, x)	Arrow(A,B), A	B	f(x)
map(f, [x_1,…,x_n])	Arrow(A,B), Power(A,n)	Power(B,n)	[f(x_1),...,f(x_n)]
reduce(f,[x_1,…,x_n])	Arrow(A,Arrow(A,A)), Power(A,n)	A	x_1 (n = 1) reduce(f,[f(x_1,x_2), …,x_n]) (n > 1)
zip(f, [x_1,…,x_n], [y_1,…,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)]

4.3 Párhuzamosítási határ

A kifejezésfa mellett az FParLin-nek szüksége van a bevezetőben említett párhuzamosítási határ értékére. Ez a szám annak a becslése, hogy hány elemi művelet esetén érdemes a munkát szétosztani a rendelkezésre álló processzor magok között. Ha a felhasználó nem tudja megbecsülni ezt a határt, akkor több különböző értéket kipróbálva tudja megkeresni az optimálisat.
A párhuzamosítási határ működését jól szemlélteti vektorok L^1 normájának (vagyis elemei összegének) kiszámítása. Ehhez a következő kifejezés fát kell felépítenünk:



Ebben a kifejezésben az egyetlen párhuzamosítható művelet a reduce. Ha a VecView vektor nézet egy 100 elemű vektorra hivatkozik, akkor az új processzor szálak elindításának költsége magában is több lenne, mint az egész művelet soros kiszámítása, ezért az utóbbi az optimális megoldás. De ha a hivatkozott vektor 100 000 elemű, akkor a napjainkban átlagosnak mondható processzorok biztosan hamarabb kiszámítják, ha az összes processzor magot kihasználjuk. Tehát például a 10000-et használva párhuzamosítási határnak, az első esetben az FParLin a reduce helyére soros számítást fog generálni, a második esetben pedig párhuzamost, és így mindkét esetben a lehető leghatékonyabb kiértékelést kapjuk.

4.4 Kiértékelés

Az összeállított kifejezésfát és a párhuzamosítási határt átadva az `evaluator.h`-ban deklarált `get_evaluator` függvénynek visszakapjuk azt a kiértékelő függvényt, ami optimálisan párhuzamosítva számítja ki a megadott kifejezést.
A `get_evaluator` a háttérben

1. típus ellenőrzést végez a megadott fán
2. megbecsüli az egyes műveletek kiszámításának költségét
3. a párhuzamosítási határ ismeretében dönt a kiszámítás módjáról
4. generál egy C++ forráskódot (`result_*.cpp`), ami az optimális számítást valósítja meg
5. lefordítja a `config.txt`-ben megadott fordítóval egy dinamikus linkelésű könyvtárba
6. a könyvtárból betölti a számítást végrehajtó függvényt és visszaadja a felhasználónak

Ha a fenti lépések végrehajtása közben hiba történik arról a felhasználó kivétel formájában kap értesítést. Erről bővebben a 4.4.2 fejezet tájékoztat.
A kiértékelő függvény egyetlen paramétere a vektor nézetekhez tartozó adatokat tartalmazó asszociatív tömb, visszatérési értéke pedig egy valós vektor, a számítás eredménye. A vektor nézeteket bővebben a 4.4.1 fejezet írja le. A kiértékelő függvényt a felhasználó tetszőlegesen sokszor meghívhatja (akár különböző vektor nézetekkel), amikor szüksége van az összeállított kifejezés értékének kiszámítására. Természetesen egy programon belül több kifejezést is össze lehet állítani, és az ezekhez generált kiértékelő függvényeket egymástól függetlenül használni.

4.4.1. Vektor nézetek

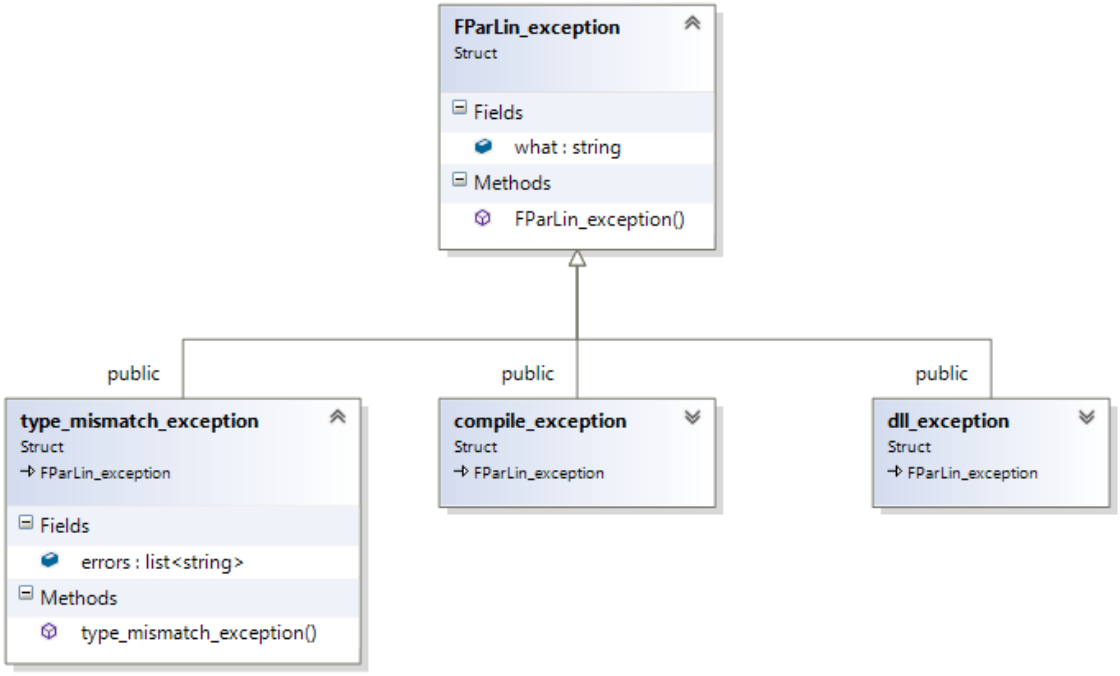
A kifejezés fában a `VecView(név, méret)` művelettel helyőrzőket (placeholder) állíthatunk be, amik később megadott valós vektorokra hivatkoznak. Ez a megoldás azért előnyös, mert

- az adatokat nem kell semmilyen módon átmozgatni az őket tartalmazó (esetleg nagyméretű) vektorokból a kifejezés fába, és
- a generált kiértékelő függvény paraméterezhető marad, így ha ugyan azt a kifejezést többféle adattal szeretnénk kiértékelni, elég egy fát felépítenünk.

Az egyes vektor nézetekhez tartozó adat vektorokat egy asszociatív tömbben (`std::map<std::string, std::vector<double>*>`) kell felsorolni, és átadni ezt a gyűjteményt a kiértékelő függvénynek. Ez futtatáskor a vektor nézetek helyére a név alapján beemeli a megfelelő adat vektorra mutató hivatkozást, és annak aktuális tartalmával számol.

4.4.2. Hibakezelés

Ha a kiértékelő függvény generálása közben hiba történik, azt az `fparlin_exception.h` headerben deklarált `FParLin_exception`, vagy az ebből származó speciális kivételek jelzik.



type_mismatch_exception

A típus ellenőrző felismeri, ha az átadott kifejezés fában nem teljesülnek a műveleteknél leírt típus követelmények, és ezt `type_mismatch_exception` kiváltásával jelzi. A követelményeknek két nagy csoportja van: amikor egy rész

kifejezésnek valamilyen meghatározott típusúnak kell lennie, és amikor két rész kifejezésnek azonos típushoz kell tartoznia.
Például a + művelet mindkét paraméterének skalárnak (`Int()` vagy `Double()` típusúnak) kell lennie, ezért a `0_scl + VecView("vec", 2)` kifejezésre a típus ellenőrző a "(double)^(2) not scalar" üzenettel jelzi, hogy double^2 (vagyis vektor) típusú rész fát talált a várt skalár helyett.
Hasonlóan a `Vec({0_scl, VecView("vec", 2)})` kifejezésre a "double != (double)^(2)" üzenetet kapjuk, mert a vektor minden elemének azonos típusúnak kell lennie. A fában talált összes független típus hiba fel van sorolva a kivétel `errors` listájában.

compile_exception

A generált forráskód fordítása előtt a standard kimeneten megjelenik a kiadandó fordítási utasítás. Ha a rendszer nem tudja végrehajtani ezt, akkor ellenőrizzük a `config.txt`-ben megadott elérési utat. Ha ez tartalmaz szóközt, akkor idézőjelek közé kell tenni az egész sort. A fordítás során keletkező üzeneteket a `build.out` fájl tartalmazza.
Maga a fordító is adhat figyelmeztetéseket vagy hibaüzeneteket. Hiba esetén ellenőrizzük, hogy a fordító megfelel-e a szükséges szoftver környezetben leírt kritériumoknak.
Gyakran előforduló hibák:

- A fordító elérési útja a `config.txt`-ben hibás.
    - üzenet*: The system cannot find the path specified.
    - megoldás: Ellenőrizzük és javítsuk az elérési utat.
- A fő program és a kiértékelő különböző processzor architektúrára lett lefordítva.
    - üzenet*: fatal error LNK1112: module machine type 'x64' conflicts with target machine type 'X86'
    - megoldás: Ellenőrizzük, hogy a megadott fordító tud-e a kívánt architektúrára fordítani. Windows-on győződjünk meg róla, hogy a _WIN64 preprocessor szimbólum pontosan akkor van definiálva, ha x64 a cél architektúra.

*: A hibaüzenetek Windows rendszerből és Visual C++-ból származnak, más konfigurációkon eltérőek lehetnek.
Ha nem sikerül betölteni az elkészített dinamikus könyvtárat, vagy azon belül a kiértékelő függvényt, azt a “could not load library” illetve a “could not get function address” üzenetek jelzik a hiba okára utaló információval együtt a kivétel `what` mezőjében.

4.5 Példa számítások

Az eddig leírtak szemléltetésére bemutatunk két jellemző lineáris algebrai alkalmazást.

4.5.1. Vektorok lineáris kombinációja

Mindennapi feladat, hogy két vektor lineáris kombinációját, azaz például egy 3*x+y alakú kifejezést akarunk kiszámítani, ahol x és y tetszőleges (de azonos) méretű vektorok. Ha ez a méret már milliós nagyságrendű, de a felhasználó nem gyakorlott a párhuzamos programozásban, akkor érdemes lehet az FParLin könyvtárat használnia. Az FParLin-ben nincsenek beépített vektor műveletek, ezért ezeket lambdák és magasabb rendű függvények segítségével kell megalkotnunk. Aki nem gyakorlott a map, reduce és zip használatában, a legtöbb esetben könnyen kiválaszthatja a megfelelőt a paraméterek számának és a várt kimenet jellegének figyelembe vételével. Fontos, hogy a kimenetek elkülönítése csak akkor ilyen egyszerű, ha egy dimenziós (skalár értékekből álló) vektorokkal dolgozunk.

függvény	**vektor paraméterek száma**	**kimenet**
map	1	vektor
reduce	1	skalár
zip	2	vektor

A 3*x-nek egy vektor paramétere van és vektor a kimenete, tehát map-et kell hozzá használni. A művelet, amit a vektor minden elemére hattatni szeretnénk a 3-mal való szorzás. A 3*x tehát map(p→3*p, x) alakban írható fel.
A vektorok összeadása két vektor paraméterű művelet, a zip-et fogjuk használni. Egy olyan lambdát kell még felírnunk, ami a bemenő vektorok egy-egy eleméhez az összegüket rendeli. Ez könnyen menne (p,q) → p+q alakban, de csak egy változós lambdákat használhatunk. Az 5.2.3 fejezetben leírt curryzés módszerét kell alkalmaznunk. Válasszuk le az első változót, és rendeljük hozzá a fenti kifejezés maradék részét: p → (q → p + q). A kapott lambda ekvivalens a kétváltozóssal, de minden hozzárendelés bal oldalán csak egy változó van, így ez már érvényes lambda függvény. A teljes összeadást zip(p → (q → p + q), 3*x, y) alakban kapjuk, amibe behelyettesítve a már összeállított skalárral szorzást a zip(p → (q → p + q), map(p → 3 * p, x), y) kifejezést kapjuk. Feltűnhetett, hogy mindkét lambdában használtuk a p változó nevet, de ez a lambdák lokális hatóköre miatt nem okoz gondot.
Az FParLin-beli átíráshoz először rögzítsük, hogy az x és y vektorokat vektor nézeteken keresztül fogjuk használni. Néhány elemnél nagyobb vektorok esetén ez a racionális megközelítés, és így a generált kiértékelő függvénnyel többféle vektorra is ki tudjuk számolni a lineáris kombinációt.
Ezután már csak a megfelelő típusokat kell beírnunk. Legyenek x és y lebegőpontos vektorok. A könnyebb átláthatóság kedvéért hozzuk létre előre a vektor nézeteknek és a lambda változóknak megfelelő (egyetlen csúcsból álló) kifejezés fákat:

```
auto x = VecView("x", 1000000);
auto y = VecView("y", 1000000);
auto p = Var(Double(), 'p');
auto q = Var(Double(), 'q');
```

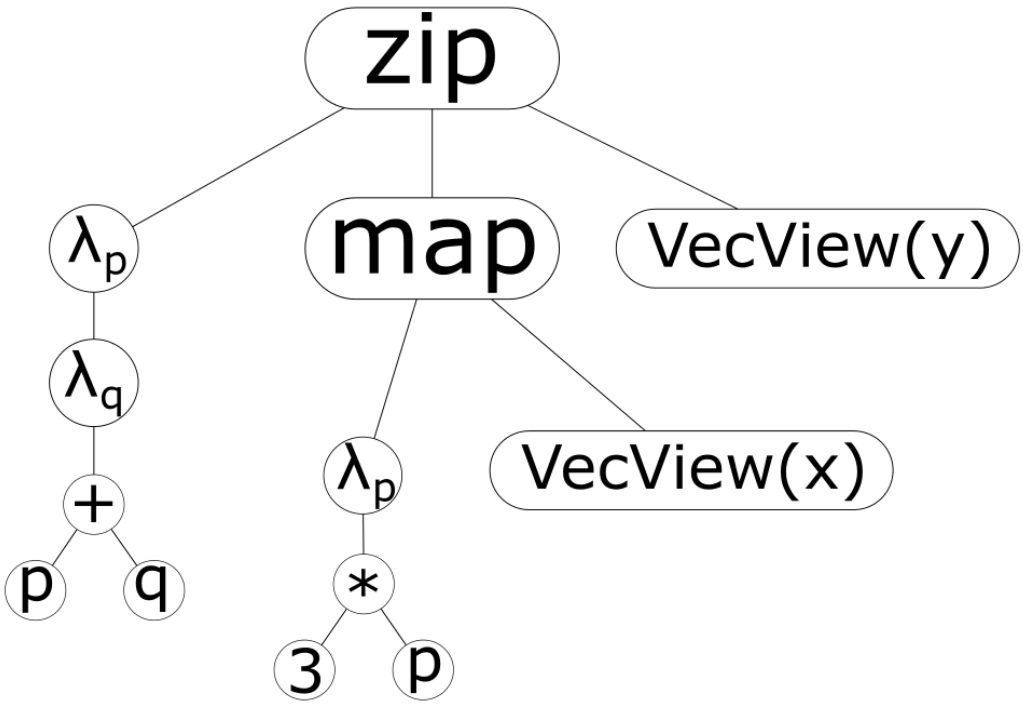
A lambdák felírásához meg kell határoznunk a visszatérési értékük típusát. A zip külső lambdájánál a hozzárendelés jobb oldalán álló (q → p + q) típusa `Arrow(Double(), Double())`, a többi lambda visszatérési értéke `Double()` típusú. A lambdák definíciói ezek alapján:

```
auto zip_lambda =
Lam(Arrow(Double(), Double()), p,
     Lam(Double(), q,
          p+q));
auto map_lambda = Lam(Double(), p, 3_scl * p);
```

A teljes kifejezés pedig:

```
auto lin_komb =
Zip(
     zip_lambda,
     Map(map_lambda, x),
     y);
```

Fontos, hogy lássuk a felépített kifejezés mögött a leírt fa szerkezetét:



Ezt a fát és a párhuzamosítási határt átadva az FParLin `get_evaluator` függvényének az elkészíti a kifejezést kiértékelő függvényt. A párhuzamosítási határ beállításához a 4.3 fejezet ad tanácsokat. A kiértékelő egyetlen paramétere a vektor nézetekhez tartozó adat vektorok asszociatív tömbje.

```
auto evaluator = get_evaluator(lin_komb, 10000);
map<string, vector<double>*> bigVectors{
```

```
    { "x", new vector<double>(1000000) },
    { "y", new vector<double>(1000000) } };
// A bigVectors feltöltése adatokkal.
vector<double> result = evaluator(bigVectors);
```

A result vektor tartalma a megadott x és y vektorok 3*x+y lineáris kombinációja.

4.5.2. Mátrix-vektor szorzás

Szintén gyakran fordul elő különböző mátrix kifejezések kiszámításában egy mátrix vektorral történő szorzása. Az FParLin még nem támogatja mátrix nézetek megadását, ezért a mátrixunk álljon csak 4 sorból, de a másik dimenzió legyen milliós az előző példához hasonlóan. Tehát a kiszámítandó kifejezés legyen M*v, ahol $M \in \mathbb{R}^{4x10^6}$ és $v \in \mathbb{R}^{10^6}$. Mátrix-vektor szorzásnál az eredmény vektor minden eleme a mátrix egy sorának és a vektornak a skaláris szorzata.
Írjuk fel először a skaláris szorzatot magasabb rendű függvények segítségével. Az $m, v \in \mathbb{R}^n$ vektorokra a skalár szorzatot a $\sum_{i=1}^{n} m_i * v_i$ összeg adja, tehát egy olyan vektor elemeinek az összege, aminek i-edik eleme m_i*v_i. Az előző példa táblázata szerint két vektorból egyet előállítani a zip művelet segítségével tudunk, az elem párok szorzatát pedig az (x,y) → x * y ≡ x → (y → x * y) lambda állítja elő. Most össze kell adnunk a szorzatokat tartalmazó vektor elemeit. Ez egy vektorból skalárt előállító művelet, ami a táblázat szerint a reduce. Az elem párok összegét visszaadó lambda a már az előző példában is használt x → (y → x + y). A különbség most az, hogy a reduce egyetlen vektor elemeit vonja össze a lambdán keresztül. Elkészült a skalár szorzat kifejezés: reduce(x → (y → x + y), zip(x → (y → x * y), m, v)). Jegyezzük meg itt is, hogy független lambdákban nyugodtan használhatunk azonos változó neveket, ezek nincsenek egymásra hatással.
Mint már kimondtuk, a mátrix-vektor szorzás eredményének egy eleme a mátrix egy sorának és a vektornak a skaláris szorzata. Fogalmazzuk meg másképp: az M*v szorzás eredményét úgy kapjuk, hogy az M minden m sorát kicseréljük az <m,v> skaláris szorzat értékével. Ezt a kicserélést a map művelet tudja megvalósítani az m → <m,v> lambdával, ami a mátrix egy m sorához annak a v-

vel vett skaláris szorzatát rendeli. Az <m,v> helyére beírva a már összeállított skaláris szorzást megkapjuk a teljes mátrix vektor szorást:
map(m → (reduce(x → (y → x + y), zip(x → (y → x * y), m, v))), M)
Kezdjük az FParLin átírást az elemi alkotó elemektől, a fa leveleitől. Az M négy sora (m1, …, m4) és a v lebegőpontos vektor nézetek lesznek, az M a négy sorból alkotott vektor. Figyeljük meg, hogy így lehet n dimenziós mátrixokat definiálni: n-1 dimenziós mátrixokból összeállítunk egy vektort. Az x és az y változók típusa `Double()`.

```
auto m1 = VecView("m1", 1000000);
auto m2 = VecView("m2", 1000000);
auto m3 = VecView("m3", 1000000);
auto m4 = VecView("m4", 1000000);
auto v = VecView("v", 1000000);
auto M = Vec({m1, m2, m3, m4});
auto x = Var(Double(), 'x');
auto y = Var(Double(), 'y');
```

A kétváltozós lambdák definíciói lényegében megegyeznek az előző feladat zip lambdájával:

```
auto zip_lambda =
Lam(Arrow(Double(), Double()), x,
     Lam(Double(), y, x * y));
auto reduce_lambda =
Lam(Arrow(Double(), Double()), x,
     Lam(Double(), y, x + y));
```

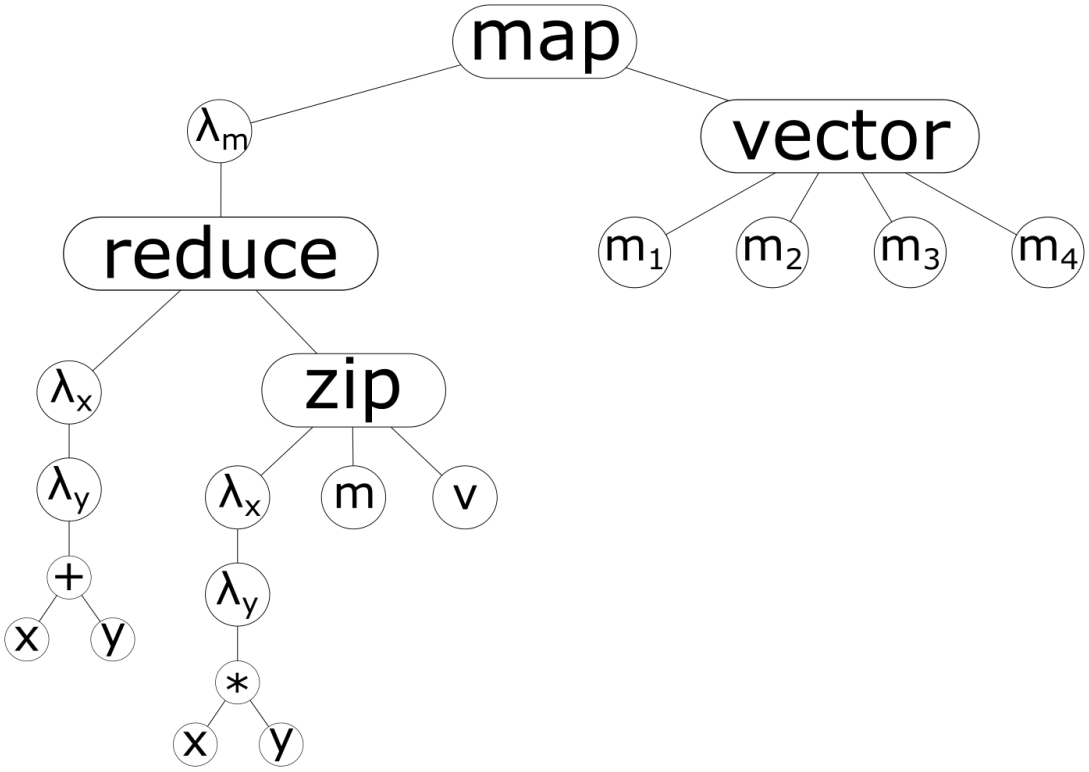
A map lambdájának paramétere a mátrix egy sora, azaz 10^6 lebegőpontosból álló vektor. Az m változó tehát:

```
auto m = Var(Power(Double(), 1000000_size), 'm');
```

Ezekből pedig már össze tudjuk építeni a teljes kifejezést:

```
auto mat_vec_mul =
Map(
     Lam(Double(), m,
          Reduce(
               reduce_lambda,
```

```
                Zip(zip_lambda, m, v))),
    M);
```



Az ábrán az $m_1,...,m_4$ és a v vektor nézeteket (az előző példával ellentétben) csak az azonosítójukkal jelöltük.

A kiértékelő függvény generálása és futtatása innentől az előző példával azonos módon történhet.

Fejlesztői dokumentáció

5. Tervezés

Napjainkban a különböző párhuzamosságot támogató platformok (sok magos processzorok, videó kártyák, publikus klaszterek, felhők) 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, lehetősége 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 számítási kapacitást. A Wigner Fizikai Kutatóközpontban működő GPU Labor több megközelítésen keresztül is keres megoldást erre a problémára. Az én vizsgálatom tárgya egy olyan rendszer, ami a rendelkezésre álló számítási platformok főbb paramétereinek ismeretében képes akár több platformot egyidejűen hatékonyan kihasználva elvégezni a megadott számítást. A probléma sokrétű, és az alkalmazott megközelítés is újszerű, ezért a megvalósítás több lépcsőben történik. A szakdolgozat témája az első lépcső, melynek elsődleges célja a később részletezett megközelítés megvalósíthatóságának vizsgálata és demonstrációja. Erre a fenti funkcióknak a következő lényegesen leszűkített, de a legfontosabb komponenseket megtartó változatát választottuk:

A támogatandó párhuzamos platformok közül csak a több magos processzorokat tartottuk meg. Hatékony kihasználásnak pedig most azt tekintjük, ha csak olyankor indítunk több processzor szálat, amikor ez a soros végrehajtáshoz képest lényegesen gyorsabb számítást eredményez.

5.1 Elméleti háttér

A kitűzött feladatból adódott, hogy a programnak tartalmaznia kell mátrix-vektor kifejezések felépítését és beható elemzését.
A kifejezések ábrázolására a kifejezésfa, mint természetes, flexibilis és könnyen kezelhető reprezentáció egyértelmű választás volt. A kifejezésfa építő elemei közül a skalár konstans, a vektor és az elemi műveletek (összeadás és szorzás) nélkülözhetetlenek az általános mátrix-vektor kifejezések felírásához. Ezekkel viszont már sok más művelet (mátrix képzés, kivonás, osztás) hatása is elérhető, amelyeket emiatt nem implementáltam. A jövőben be fognak kerülni a kényelmesebb használat érdekében. Bevettük az építő elemek közé a lambda-kalkulusból a lambda absztrakciót és alkalmazást, ezzel lehetővé téve tetszőleges függvények leírását. Ezeknek a függvényeknek a flexibilis alkalmazására pedig a funkcionális programozás eszköztárából ismert map, reduce és zip magasabb rendű függvényeket választottuk. Tapasztalataink szerint a felsorolt műveletek eleget tesznek az elvárásainknak, leírható velük a mátrix-vektor kifejezéseknek egy megfelelően széles köre. A funkcionális építőelemek helyett választhattunk volna az imperatív programozásból ismerteket is (érték adás, indexelés, elágazás, ciklus), de egyrészt a reprezentálni kívánt matematikai kifejezéseknek az előbbiek a természetes leírói, másrészt sokkal könnyebb a funkcionális műveletek helyességét ellenőrizni.
A felépített fákon különböző elemzéseket érdemes definiálnunk, amik többféle bejárást és transzformációt is magukban foglalhatnak. A kifejezésfák bejárásainak és átalakításainak leírására absztrakt módot kerestünk, hogy könnyen kezelhetők, illetve a későbbiekben kombinálhatók és optimalizálhatók legyenek, függetlenül a fák reprezentációjától. Ennek a célnak kiválóan megfelelnek az F-algebrák, amelyek a kategória elmélet eszköztárával írnak le tetszőleges fa transzformációkat.
A lambda-kalkulus és az F-algebrák alapszintű ismerete szükséges a később bemutatott eljárások megértéséhez, ezért a következő fejezetek részletesebben bevezetik ezeket a fogalmakat.

5.2 Lambda-kalkulus

A lambda-kalkulus egy formális rendszer a függvények bevezetésének és kiértékelésének kombinálására. Mi most ennek azt a kiterjesztését vizsgáljuk, ahol érvényes lambda kifejezésnek tekintünk egy atominak tekintett réteget is, az egész számokból az alapműveletekkel felírt kifejezéseket.

5.2.1. Absztrakció

A λx.t függvény absztrakció az egy változós x→ t függvényt definiálja, ahol *t* egy érvényes lambda kifejezés, *x* pedig a helyőrző szimbólum. *t* belsejében *x*-et kötöttnek tekintjük. Az f(x)=2*x+3 függvénynek például a λx.(2*x+3) absztrakció felel meg. Ugyanezt azt FParLin beépített nyelvén a 4.2.2 fejezetben leírt `Lam` művelet segítségével

```
Lam(Double(), Var(Double(),'x'),
     2_scl * Var(Double(),'x') + 3_scl)
```

alakban lehet felírni. Az egyetlen lényeges különbség a típusok megjelenése: a típusos lambda-kalkulus lényege, hogy a függvények csak megadott, a változójukkal egyező típusú kifejezésekre (esetünkben `Double()`) alkalmazhatók. Azért választottuk a lambda-kalkulusnak ezt a fajtáját, mert a típusok segítségével jobban ellenőrizhető a felírt kifejezések helyessége. Ha a λx.t absztrakció *t*-jének belsejében egy másik absztrakció is az *x* változót használja, akkor a belső absztrakció törzsében az *x* szimbólum a lambda-kalkulus szabályai szerint a belső absztrakció változójára hivatkozik. Az FParLin a lambda absztrakciót C++ lambda kifejezésekkel reprezentálja, amelyekben a belső lambda változója automatikusan elfedi a külső lambda azonos nevű változóját, ezzel pontosan modellezve a lambda-kalkulust.

5.2.2. Alkalmazás

Az alkalmazás egy lambda absztrakción és egy tetszőleges kifejezésen végzett művelet. Értékét úgy kapjuk meg, hogy a változó minden előfordulását kicseréljük a megadott kifejezésre. Úgy jelöljük, hogy egyszerűen egymás mellé írjuk a

lambdát és a kifejezést. A λx.(2*x+3) 4 kifejezés a λx.(2*x+3) absztrakció alkalmazása a 4 számra. λx.(2*x+3) 4 = (2*4+3), ami már a választott atomi rétegbeli kifejezés, ki tudjuk értékelni. Az alkalmazást az FParLin-ben az `App` művelet valósítja meg. A korábbi példát kiegészítve az

```
App(
     Lam(Double(),Var(Double(),'x'),
          2_scl*Var(Double(),'x')+3_scl),
     4_scl))
```

kifejezést kapjuk, amely hűen tükrözi a lambda-kalkulusbeli alakot. Az alkalmazás bal asszociatív művelet, de az FParLin-ben a kötelező zárójelezés egyértelműsíti a műveleti sorrendet.

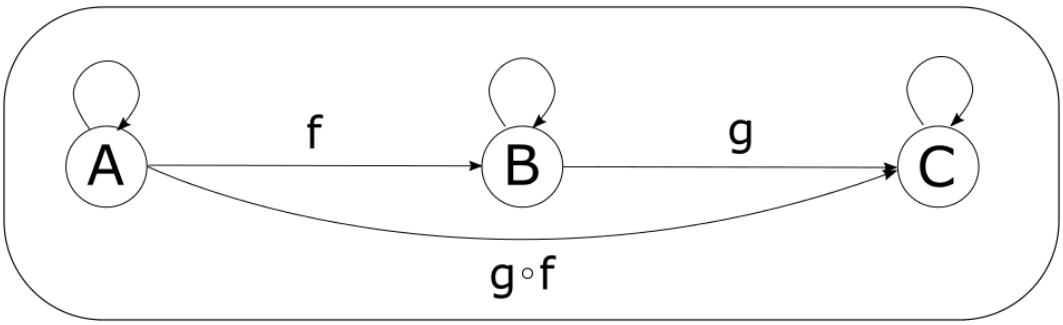
5.2.3. Curryzés

A függvény absztrakció csak egy változós függvények definiálását teszi lehetővé. Több változós függvényekkel egyenértékű hatást absztrakciók egymásba ágyazásával, és az így kapott függvény láncolt alkalmazásával érhetünk el. Ezt a módszert curryzésnek hívják. Figyeljük meg, hogy működik a λx.(λy.(x+y)) absztrakcióval megadott függvény az 1 és 2 kifejezésekre alkalmazva: λx.(λy.(x+y)) 1 2 = λy.(1+y) 2 = 1 + 2. Láthatóan megegyezik az (x,y)→ x+y függvény alkalmazásával az (x=1,y=2) párra.
Ez a módszer megkerülhetetlen a reduce és a zip (lásd: 4.2.2) magasabb rendű függvények használatakor, mert ezek két változós lambdát várnak első paraméternek. Például a reduce és a fenti lambda segítségével így számítható ki az [1,2,3] vektor elemeinek összege:

```
Reduce(
     Lam(Arrow(Double(), Double()), Var(Double(),'x'),
          Lam(Double(), Var(Double(),'y'),
               x + y)),
     Vec({1_scl, 2_scl, 3_scl}))
```

5.3 Kategóriaelmélet

A kategóriaelmélet az algebrai struktúrák közötti összefüggéseket, leképezéseket tanulmányozza, szemben az absztrakt algebrával, ami általában ezeknek a struktúráknak a magukban vett tulajdonságait vizsgálja. Maga a kategória egy algebrai struktúra, amelynél a teljességet (zártságot) nem követeljük meg csak az asszociativitást és az egységelem létezését.
A kategóriák alkotó elemei objektumok és ezeket összekötő morfizmusok (nyilak). Minden nyílnak létezik egyértelmű kezdő- és végpontja, és a nyilak komponálhatók. Tehát ha f: A→B és g: B→C nyilak, akkor egyértelműen létezik a g∘f: A→C nyíl 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: $X \to X$ önmagába mutató nyíl, amely a kompozícióban egységelemként viselkedik. Tehát bármely A, B objektumokra és f: A→B nyílra $f \circ id_A = f = id_B \circ f$.



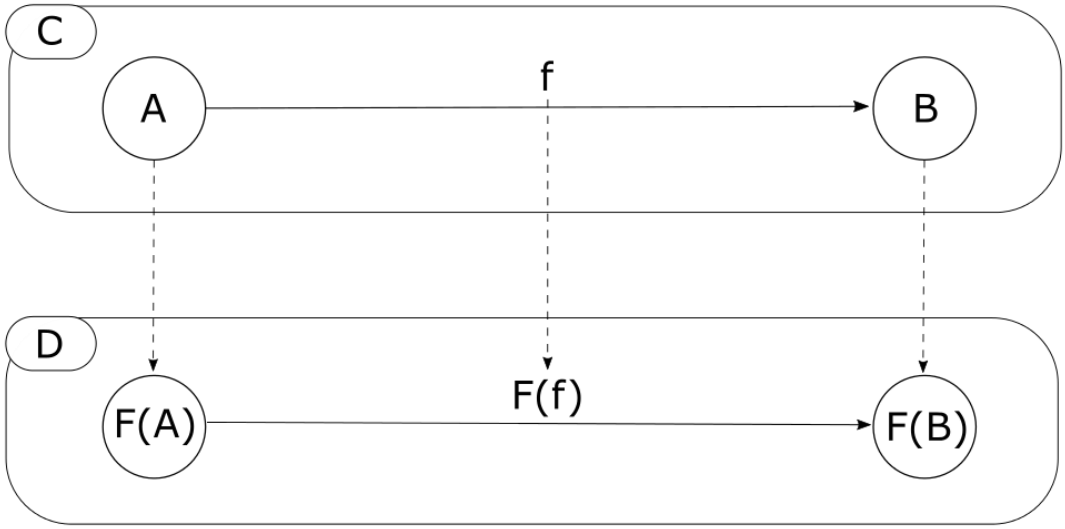
1. Ábra: Objektumok és morfizmusok egy kategóriában.

5.3.1. Funktor

A funktorok kategóriák közötti leképezések. 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
    - $\forall A \in C$: $F(id_A)=id_{F(A)}$

- ∀(f: A→B, g: B→C)∈C: F(g ∘ f) = F(g) ∘ F(f)



2. Ábra: Funktor struktúra két kategória között.

A programozási nyelvek típusrendszere is felfogható egy kategóriaként, ahol az objektumok típusok és a függvények a morfizmusok, a funktorok pedig típus szintű leképezések polimorfikus, vagy generikus típusok felett (pl. tárolók). 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 magasabb rendű 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

5.3.2. F-algebra

Az algebrák két legfontosabb tulajdonsága, hogy lehetővé teszik kifejezések megalkotását, és ezen kifejezések kiértékelését. Az F-algebrák segítségével absztrahálhatunk a kifejezés struktúrák és kiértékelési módok felett.

Kifejezések

Kifejezések felírásának elterjedt módja a generatív nyelvtanok használata. Például egy egyszerű nyelvtan Backus-Naur-formában:

<Expr> ::= <Const Int> | <Add> <Expr> <Expr> | <Mul> <Expr> <Expr>

A generált nyelv az egész számokból prefix összeadással és szorzással felírható kifejezéseket tartalmazza. Az FParLin-ben egy ehhez hasonló nyelvtant szeretnénk használni az érvényes kifejezések meghatározására, de a típusok szintjén, mert akkor a C++ típusrendszere biztosítja a felírt kifejezések helyességét. Ehhez azonban a fenti rekurzív grammatikából előbb ki kell emelnünk a rekurziót, mert a típusrendszerek ezt általában nem támogatják (a C++ sem). Ezt először egy egyszerűbb példán mutatjuk be a C++ nyelvet használva, mert ennek az ismerete egyébként is szükséges az FParLin fejlesztéséhez.

A faktoriális függvényt egyszerűen felírhatjuk rekurzívan:

```
int factorial(int x) {
      if(x == 0) return 1;
      return x * factorial(x - 1);
}
```

A nem rekurzív felíráshoz szükség van a fixpont fogalmára. Egy leképezés fixpontja olyan értéket jelent, amit a leképezés önmagába képez. fix(f)-fel jelölve az f függvény fixpontját, ezt a fix(f) = f(fix(f)) egyenlőség írja le. Fixpont-kombinátornak hívjuk azt a függvényt, ami egy függvényhez a fixpontját rendeli.

```
#include <functional>
#include <iostream>

int fact(std::function<int(int)> f, int x) {
      if (x == 0) return 1;
      return x * f(x - 1);
}

std::function<int(int)>
fix(std::function<int(std::function<int(int)>, int)> f) {
      return [=](auto x){ return f(fix(f), x); };
}

int main() {
      auto factorial = fix(fact);
```

```
	std::cout << factorial(3) << std::endl;
}
```

Megállapíthatjuk, hogy a fenti fact függvény nem hivatkozik önmagára, valóban nem rekurzív, de tartalmazza a faktoriális számítás indukciós lépését. A rekurziót kiemeltük a fix függvénybe, ami egy fixpont-kombinátor, hiszen f-hez f(fix(f))-et rendeli (f második paraméterét szabadon hagyva). Az indukciós lépést kombinálva a rekurzióval (fix(fact)) a fenti rekurzív definícióval ekvivalens függvényt kapunk. Figyeljük meg az előállított factorial függvény kiértékelését:

factorial(3) =

fix(fact)(3) = fact(fix(fact))(3) =

3 * fix(fact)(2) = 3 * fact(fix(fact))(2) =

3 * 2 * fix(fact)(1) = 3 * 2 * fact(fix(fact))(1) =

3 * 2 * 1 * fix(fact)(0) = 3 * 2 * 1 * fact(fix(fact))(0) =

3 * 2 * 1 * 1

A fact a fixponton keresztül újra és újra alkalmazza önmagát, amíg a szám paraméter el nem éri a 0-t. Ekkor a speciális kilépő ága hajtódik végre, ami nem függ az első paramétertől, és így véget ért a rekurzió.

Emeljük ki a rekurziót a fejezet elején leírt kifejezés nyelvtanból is, a faktoriálissal analóg módon. Az értékek szintjén működő fact megfelelője a következő, típusok felett értelmezett hozzárendelés lesz:

$$\mathrm{ExprF\ a} = \begin{cases} \mathrm{Const\ Int} \\ \mathrm{Add\ a\ a} \\ \mathrm{Mul\ a\ a} \end{cases}$$

Az ExprF egy generikus *a* típushoz rendeli a jobb oldali kifejezések valamelyikét. A típus elmélet terminológiája szerint ez egy összeg típus (sum type), ami több típus közötti *vagy* kapcsolatot jelent. Az *a* paraméter a fact f paraméterének felel meg, a kilépő ág pedig a Const Int, mert ez nem függ *a*-tól. Az ExprF és a típusokon értelmezett fixpont kombinátor egy lehetséges C++ implementációja, az összeg típust a `boost::variant`-tal kifejezve a következő (3):

```
struct Const {
	int value;
};
```

```
template<typename A>
struct Add {
      A left;
      A right;
};

template<typename A>
struct Mul {
      A left;
      A right;
};

template<typename A>
using ExprF = boost::variant<
      Const,
      boost::recursive_wrapper<Add<A>>
      boost::recursive_wrapper<Mul<A>>>;

template<template<typename> class F>
struct Fix : F<Fix<F>> {
      Fix(F<Fix<F>> f): F<Fix<F>>(f){}
};

using Expr = Fix<ExprF>;
```

Az eredeti rekurzív Expr-t a faktoriális példával megegyező módon, `Fix<ExprF>`-ként kapjuk, ez a kifejezés fánk csúcsainak típusa. Így már tetszőleges kifejezést le tudunk írni, a 2 + 3 alakja például

```
Fix<ExprF>(Add<Expr>{
      Fix<ExprF>(Const{2}), Fix<ExprF>(Const{3}) });
```

Ez a forma meglehetősen terjengős, ezért az FParLin segédfüggvényeket biztosít a kényelmes használathoz (lásd: 4.2.2). Ezek, illetve a kifejezés összeg típus és a fixpont-kombinátor az `expr.h` és a `fix.h` headerekben vannak definiálva.

Kiértékelés

A felépített kifejezésfa egy polimorfikus tároló objektum. Tárolhatunk benne skalár értékeket (aminek a kiértékelésnél van fontos szerepe), kezdetben viszont fix(ExprF) típust tárol, ettől lesz rekurzív. Az egyes műveleteket a fa csúcsaiban egy összeg típus különböző eseteivel reprezentáljuk. A fa egyben kijelöl egy

műveleti sorrendet is: egy csúcsot csak akkor lehet kiértékelni (transzformálni), ha a gyermekeit már transzformáltuk.
Egy adott transzformáció a fában tárolt *a* típust valamely *b* típusba képzi. Viszont a különböző műveletekhez (a sum type különböző ágaihoz) különböző transzformációkat szeretnénk társítani, már csak az esetleg eltérő számú gyerekeik miatt is. Ezért minden művelethez kell egy külön kiértékelő függvény. Ezeknek a gyűjteményét nevezzük algebrának. Ez ekvivalens egyetlen kiértékelő függvénnyel, amely a fa összeg típusát várja bemenetül. Az algebra egy fa csúcsra való hattatásán a továbbiakban ennek a függvénynek az alkalmazását értjük. Szeretnénk, ha ezek a kiértékelők lokálisak lennének, nem függenének az alattuk levő fától. Ezért megköveteljük, hogy a transzformáció eredménye a konkrét művelettől függetlenül ugyan az a *b* típus legyen, amely típust carrier type-nak hívunk. Emiatt az algebra összes kiértékelő függvényének is ezt a típust kell visszaadnia. A lokalitás következményeként a csúcsok kiértékelésénél, csak az alattuk már kiértékelt gyermekek eredményeivel kell dolgozni.
A fentiekből a teljes fa kiértékelését a következő indukcióval kapjuk:

- Ha konstans csúcsot akarunk kiértékelni, aminek nincs gyereke, akkor arra hattatjuk az algebrát, aminek az eredménye *b* típus.
- Ha nem konstans csúcsot akarunk kiértékelni, akkor annak vannak gyerekei, amiket először kiértékelünk, majd a kapott eredményekre hattatjuk az algebrát.

A különböző típusú csúcsok eltérő számú gyerekeire való hattatást általánosan is tudjuk definiálni, ha észrevesszük, hogy a polimorfikus típus, amiből a kifejezéseket építjük, az előző fejezetben definiált ExprF egy funktor. Az F funktorhoz a definíciója szerint tartozik egy fmap: (A→B) → (F(A)→F(B)) függvény, ami bármely f: A→B morfizmushoz (esetünkben C++ függvényhez) hozzárendel egy F(f): F(A)→F(B) morfizmust. Az előzővel ekvivalens az fmap: (A→B, F(A)) → F(B) alak (lásd curryzés). Az F=ExprF esetben az F(A) csak Const, Add(left: A, right: A) vagy Mul(left: A, right: A) lehet, az fmap definíciója így:

fmap(f, Const i) = Const i

fmap(f, Add(left, right)) = Add(f(left), f(right))

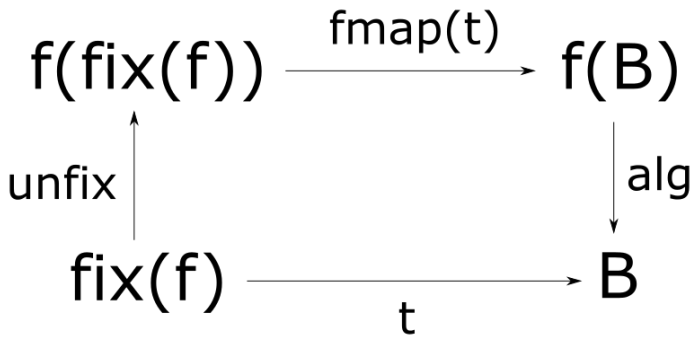
fmap(f, Mul(left, right)) = Mul(f(left), f(right))

A definícióból jól látható, hogy valóban az f általános hattatását írja le a polimorfikus ExprF-re. Ha tehát van egy fix(ExprF)-eket tároló fánk (ExprF(fix(ExprF)), amit a fentiekből F=ExprF és A=fix(ExprF) helyettesítéssel kapunk) és egy f: fix(ExprF) → B függvényünk, akkor az fmap alkalmazásával ExprF(B)-t, azaz B-ket tartalmazó fát kaphatunk.

Ne felejtsük el, hogy a kifejezésfánkban az elemek típusa fix(ExprF). Ahhoz, hogy a fentiek szerint használni tudjuk rajta az fmap-et, először vissza kell alakítanunk a fix inverz műveletével: unfix: fix(f) → f(fix(f)).

Mindent előkészítettünk ahhoz, hogy a fenti indukció alapján definiáljuk az általános fa transzformáló függvényt. t-vel jelölve a függvényt és alg-gal az alkalmazott algebrát: t = alg ∘ fmap(t) ∘ unfix, azaz először kibontjuk a gyökér csúcsot az unfix-szel, ezután alkalmazzuk a gyerekeire magát a transzformációt az fmap segítségével, végül az algebra hattatásával megkapjuk az eredményt. Az fmap-ből az algebrába egy olyan fát adunk át, aminek a levelei carrier type típusúak, és a korábbi részfák eredményeit tárolják.

A rekurziót itt is kiemeljük egy rendkívül általános, a kategóriaelméletben katamorfizmusnak (4) hívott függvénybe: cata(alg, fa) = (alg ∘ fmap (cata alg) ∘ unfix)(fa).

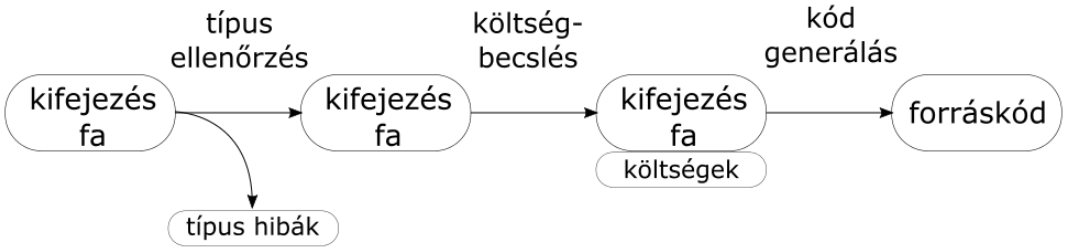


3. Ábra: A katamorfizmust felépítő függvények szignatúráinak szemléltetése.

Az egyes funktorokhoz tartozó fmap függvények az `expr_fmap.h` és a `type_fmap.h` headerekben, a cata függvény az `falgebra.h`-ban van definiálva. A fentiekben az F-algebráknak csak a számunkra legfontosabb tulajdonságait tárgyaltuk a későbbiek megértéséhez szükséges szinten. A témába általános bevezetést ad Bartosz Milewski írása (5).

5.4 Fa elemzések és transzformációk

Az előző fejezetek lefektették az elméleti alapot annak a bemutatásához, ahogyan az FParLin egy kifejezés fából a kifejezést kiszámító forráskódot készít. Jól demonstrálja az F-algebrák kifejező erejét, hogy ezt szó szerint csak három katamorfizmus hívással, három különböző algebrával éri el. Az egyes algebrák deklarációi az `<algebra_neve>.h` header fájlban, definíciói az `<algebra_neve>.cpp` forrás fájlban találhatók.



5.4.1. Típus ellenőrzés

Az FParLin az összeállított kifejezések helyességét két szinten is ellenőrzi. A fa szerkezet helyességét már a C++ típusrendszere biztosítja, ezért döntöttünk a kifejezésfa típus szintű reprezentációja mellett. Az egyes műveletek szemantikus helyességét pedig ez a transzformációs menet ellenőrzi. A felhasznált algebra a `typecheck_alg`.

A transzformáció bemenete egy kifejezésfa, eredménye pedig `pair<Fix<F>, list<string>>` típusú, az eredeti fából és a talált hibák listájából álló pár. Az algebrában az egyes műveletekhez tartozó függvények

1. összegyűjtik a gyerek csúcsokban talált hibákat,

2. ha ilyenek nem voltak, azaz a gyerekek helyesek, akkor ellenőrzik az adott műveletre vonatkozó, a 4.2.2 fejezetben leírt típus követelmények teljesülését,
3. ha a teljes rész kifejezés helyesnek bizonyul, akkor meghatározzák a típusát a gyerekek típusának ismeretében, a szintén a 4.2.2 fejezetben leírt módon.

Ezzel azt érjük el, hogy a végeredményként kapott hiba listában szerepel az összes egymástól független hiba, de nem tartalmaz olyan felesleges hibajelzéseket, amik egy már hibás gyerek következményei.

Típusok összehasonlítása

Mint azt már a 4.2.1 fejezet is említette, magukat a típusokat is fában ábrázoljuk, a kifejezésekhez rendkívül hasonló módon. Ez azért volt célszerű, mert a vektor és a függvény hierarchikusan kombinálható típusok, természetes reprezentációjuk a fa, és így a fák transzformációjára elkészített absztrakt algoritmusainkat is módosítás nélkül fel tudtuk használni típusok manipulációjára. Ahogy már a 4.4.2 fejezet is megjegyezte, a típus követelmények két nagy kategóriába sorolhatók.
Az egyik eset, amikor egy részkifejezés típusával szemben valami rögzített elvárásunk van. Ilyen például az, hogy a map művelet első paramétere lambda, a második pedig vektor legyen. Ekkor elegendő a megfelelő gyerek csúcs típus fájának a legfelső szintjeit megvizsgálni. A map esetében ha az első gyerek típus fájának gyökere lambda csúcs, azzal már teljesíti az előbbi elvárást a fa többi részétől függetlenül. Csak a gyökér csúcs vizsgálata nem minden esetben elég, például a reduce műveletnél, ami kétváltozós lambdát vár. Ennek ellenőrzéséhez azt kell megnézni, hogy a gyökér csúcs is, és annak jobb gyereke is lambda legyen. De a fa többi része most is lényegtelen. Az ilyen ellenőrzések megkönnyítését szolgálja az `expr.h`-ban definiált `get_type` segédfüggvény, ami eldönti, hogy a megadott kifejezés típusa (típus fájának gyökere) megegyezik-e a template paraméterként megadott típussal.
A követelmények másik kategóriája az, amikor két részkifejezés típusa tetszőleges lehet, de meg kell egyezniük. Ilyenre példa az `App` művelet, ahol a lambda

változójának és a bemenetnek azonos típusúnak kell lennie. Ebben az esetben a két kifejezés teljes típus fájának egyeznie kell, nem csak a gyökér csúcsoknak. Két fa egyezőségét nem (vagy csak nagyon körülményesen) lehet közvetlenül vizsgálni az egyébként használt katamorfizmus alapú transzformációk segítségével. Ezért a fákat először szöveggé alakítjuk az 5.4.3 fejezetben leírttal lényegében azonos módon, a `typeprinter_alg` algebrát felhasználva. Ez az átalakítás egy injektív leképezés a fák és a szöveges reprezentációk között, így a könnyen összehasonlítható szöveges eredmények egyenlősége ekvivalens az eredeti típusokéval. Ez a megoldás csak azért helyes, mert a típus műveletek között nincs kommutatív, ahol a gyerekek különböző sorrendje is ugyan azt a típust jelentené.

5.4.2. Költség becslés

Ebben a lépésben készül el az egyes részkifejezések kiszámítási költségének becslése a `costest_alg` segítségével. A kiszámítás költségét egy egész számmal, az elvégzett matematikai műveletek számával becsüljük. A transzformáció bemenete egy kifejezésfa, kimenete ugyan az a fa, aminek a csúcsaiban ki van töltve az adott részkifejezés becsült kumulatív költsége. A kitöltést az algebra függvényei a következők szerint végzik:

- A skalár, vektor nézet és változó csúcsok költsége 1.
- A vektor költsége az elemeinek költségének összege.
- Az összeadás, szorzás és lambda alkalmazás költsége az argumentumok költségének összege +1, ami az adott művelet elvégzésének költsége.
- A lambda absztrakciónak nincs külön költsége, megegyezik a törzs kifejezés költségével.
- A map, reduce és zip műveletek költsége a következők összege
    - a vektor argumentum költsége (zip esetén a két vektor költségének összege)
    - az alkalmazott lambda költsége szorozva a vektor méretével, mert ennyiszer kell alkalmazni (és így kiszámítani) a lambdát
    - 1, ami ennek a műveletnek a költsége

5.4.3. Kód generálás

Az utolsó transzformáció szöveggé (C++ forráskóddá) alakítja a költségbecslésekkel ellátott kifejezés fát a `codegen_alg`-ot felhasználva. Az algebra bemenete a fa mellett a felhasználó által beállított párhuzamosítási határ, és a párhuzamosítás korlátozását megadó logikai érték (szerepe lejjebb kerül kifejtésre), ami alapértelmezetten igaz.

Ebben a transzformációban különösen fontos szerepe van az algebra map, reduce és zip csúcsokhoz tartozó függvényeinek. Ezek azok a műveletek, amiket sorosan és párhuzamosan is ki lehet értékelni. Pontosan akkor szeretnénk a párhuzamos kiértékelést választani, ha

- az adott csúcs becsült költsége túllépi a párhuzamosítási határt, és
- egyetlen leszármazottja sem volt még párhuzamosítva.

Az első pont nem igényel sok alátámasztást. A párhuzamosítási határ azt a számítási költséget adja meg, ami fölött már megéri a párhuzamos kiértékelést választani. A második pont célja azt elkerülni, hogy többszálú végrehajtás közben az egyes szálak maguk is újabb szálakat indítsanak. Ez várhatóan nem lesz hatékony, mert többszálú kiértékelésnél eleve annyi szálat indítunk, ahányat csak a rendszer egyidejűleg végre tud hajtani. Egy alacsonyabb szinten még több szál elindítása nem fogja növelni a teljesítményt, viszont költsége van. De ha a felhasználó fel akarja oldani ezt a második korlátozást például kísérleti célból, azt megteheti az algebra harmadik, párhuzamosítást korlátozó logikai paraméterének hamisra állításával. Az algebra visszatérési értéke `pair<string,bool>`, ahol a generált forráskód melletti logikai érték azt tárolja, hogy történt-e párhuzamosítás a fában. Ha mindkét fenti feltétel teljesül, akkor az `fparlin.h`-ban implementált segédfüggvények közül az adott művelet párhuzamos implementációjának hívása kerül a generált kódba.

Tervezési szempontból érdekes még a lambda absztrakcióból generált kód. Ez a lambda-kalkulusbeli fogalom elég messze van 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 magasabb rendű függvénynek.
A katamorfizmus és az algebrák kifejező erejét jól demonstrálja ez a transzformáció. A kód generálás elsőre összetettnek tűnő feladata természetesen bomlik le az egyes műveletekhez szükséges néhány karakteres szövegekre és ezek összeállítására. A generált kóddal kapcsolatos implementációs részleteket és az `fparlin.h` segédfüggvényeinek diszkusszióját a 6.2 fejezet tartalmazza.

5.5 Nyelv választás

Mint az az előző fejezetekből világosan látszik, az FParLin erősen épít a funkcionális programozás eszköztárára. Ennek ellenére C++-ban készült, ahol a leírt konstrukciókat csak sokkal nehézkesebben lehet kifejezni, mint egy funkcionális nyelven. Ezt a látszólagos ellentmondást hivatott feloldani ez a fejezet.
Az FParLin egy könyvtár, így nagyon fontos szempont volt, hogy könnyen illeszkedjen azokhoz a programokhoz, amelyekkel használni szeretnék. Azokat a nagy teljesítmény igényű számításokat pedig, amiknek elemzésére és esetleges gyorsítására a könyvtár használható jellemzően C++-ban implementálják. Minden bizonnyal megvalósítható lenne egy funkcionális nyelven írt könyvtár és egy C++ program együttes lefordítása, de ez egy sor egyébként felesleges függőséget kényszerítene a felhasználóra, ezzel lényegesen rontva az FParLin gyakorlati alkalmazhatóságát.
A C++ mellett szólt az is, hogy a rendelkezésre álló alacsony szintű fordítók széles körűen és hatékonyan optimalizálják a kódot. A magasabb szintű nyelvekkel ellentétben explicit a memóriakezelés, aminek előnye a kiszámíthatóbb teljesítmény és erőforrás használat. Ugyanakkor a viszonylag erős típusrendszere képes típushelyesen kifejezni az F-algebra típusrendszerét.

6. Megvalósítás

A fejlesztés során verziókövetésre a GitHub rendszert használtuk (6).

6.1 Fa transzformáció

Az F-algebrák C++ implementációjához Eric Niebler példa kódját (3) használtuk kiinduló pontnak. A transzformációs menetek implementációja szorosan követi az 5.4 fejezetben leírt logikát, így az implementáció közben kevés új kérdés merült fel. Ezeket írják le a következő fejezetek.

6.1.1. Boost

Úgy döntöttünk, hogy a példához hasonlóan mi is a Boost könyvtár `variant` osztályát használjuk a kifejezésfa (és a típus fa) összeg típusához. A Boost egy széles körben használt platform független C++ könyvtár, így ez a függőség vállalhatónak tűnt. A variant lényegében a C++ unió típusának egy biztonságos, típus ellenőrzött generikus megfelelője. Ez után már egyértelmű döntés volt, hogy az algebrákat `boost::static_visitor`-ral fogjuk megvalósítani. Ez az osztály pont azt nyújtja, amit az F-algebra kiértékelés fejezetben algebraként megfogalmaztunk. Kiértékelő függvények gyűjteménye, az összeg típus (`boost::variant`) minden ágához külön-külön függvény azonos visszatérési típusokkal.
Sajnos a Boost hordozhatósága nem tökéletes, a Visual Studio-ba épülő clang fordítóval nem fordul. Szerettük volna elérni, hogy csak az FParLin lefordításához legyen szükség a Boost-ra, és a felhasználóig már ne érjen el ez a függőség, de ez nem volt lehetséges. Ezek miatt a jövőben valószínűleg saját összeg típus implementációra fogjuk cserélni a `boost::variant`-ot.

6.1.2. Műveletek

Az összeg típus ágainak, azaz az egyes műveleteknek egyértelműen tartalmaznia kell a gyerek csúcsokra való hivatkozást. Eric Niebler kódjával ellentétben mi

pointeres hivatkozást használunk, ennek okát a 6.1.3 fejezetben fejtjük ki. Úgy döntöttünk, hogy az egyes csúcsok típusát és költségét is az egyes ágakban tároljuk, nem magában az összeg típusban. Így az algebrák transzformáló függvényei könnyen hozzáférnek ezekhez az információkhoz az adott csúcsban, a gyerekekből viszont nehéz kinyerni, mert azokat a szülő csak összeg típusként látja. Ezt a problémát oldja meg az `expr.h get_type` és `get_cost` függvénye, amik egy-egy külön algebra segítségével kielemelik a megfelelő értéket az összeg típusból.

6.1.3. Anamorfizmus

Az eredeti elképzelésünk az volt, hogy az FParLin a kapott kifejezésfából nem kódot generál, hanem az elemzés után egyből ki is értékeli. A kiszámítás legtöbb lépése pontosan úgy történt volna, mint az F-algebra bevezetésénél leírt összeadást és szorzást használó példánál. Az egyetlen művelet, ami teljesen más kiértékelési logikát igényelt, az a lambda alkalmazás. Ha lentről felfelé értékeljük ki a kifejezés fát a katamorfizmus logikája szerint, akkor egy változó csúcshoz érve meg kellene határoznunk annak értékét, méghozzá csak a gyerekeiből származó információt felhasználva. De a változó csúcsnak nincs gyereke, az értékét az alkalmazás csúcsból hivatkozott teljesen független rész kifejezés adja. Ezt a problémát egy anamorfizmus beiktatásával oldottuk meg. Az anamorfizmus a katamorfizmus duális művelete, legfontosabb tulajdonsága számunkra az, hogy felülről lefelé dolgozza fel a fákat. A megoldás az lett, hogy

- az alkalmazás csúcsokban a bemenetet jelentő részfát elkezdjük lefelé propagálni a fában egy verem tetejére téve
- a lambda absztrakció csúcsok kiveszik a verem legfelső elemét, és egy asszociatív tömbben hozzárendelik a változójukhoz
- ez a tömb szintén propagálódik lejjebb a fában, elér a változóhoz, ami megtudja belőle a saját értékét

Így a következő transzformációs menet a már jól ismert katamorfizmus logika szerint ki tudta értékelni a fát. Az csúcsokban a gyerekekre vonatkozó hivatkozást

pointerekre cseréltük, hogy a rész fák fent leírt terjesztése a fában ne járjon azok lemásolásával.
Az egész projekt szempontjából ez egy értékes eredmény, mert láttuk, hogy lehetséges a fák helyben történő kiértékelése. De a további tervek szempontjából (például videokártyák használata) előremutatóbbnak tűnt a kódgenerálás, így végül ebben az irányban haladt a fejlesztés és most nem használunk anamorfizmust.

6.2 A generált forráskód

6.2.1. Kiértékelő függvény

Az FParLin által generált kód egyetlen `evaluator` nevű kiértékelő függvény deklarációját és definícióját tartalmazza. A deklaráció `extern "C"`, mert később futás időben akarjuk név alapján betölteni a függvényt. A kiértékelő függvény egyetlen paramétere a vektor nézetekhez tartozó adatokat tartalmazó `std::map<std::string, std::vector<double>*>`, visszatérési típusa `std::vector<double>`. Az utóbbi egy erős korlátozás, hiszen egy kifejezés értéke nem csak vektor lehet, hanem akárhány dimenziós mátrix, lambda, vagy akár lambdákból álló mátrix. Ezt is meg lehetett volna valósítani, elég lett volna, ha az `evaluator` egy megfelelő összeg típussal tér vissza. De ebben az esetben a felhasználónak tudnia kellene, hogy milyen típusú a számításának az eredménye. Nem akartuk megkockáztatni, hogy rossz típusként próbálja értelmezni a kapott eredményt, ezért vezettük be ezt a korlátozást. Az pedig, hogy az eredmény csak lebegőpontos lehet logikussá tette, hogy csak lebegőpontos konstansokat használjunk. A kiértékelő törzse lényegében csak lambda kifejezésekből és segédfüggvény hívásokból áll. A lambdák referencia szerint veszik át a szükséges környezetet és a paramétert, így alkalmazásuk nem hordoz extra költséget a végrehajtott számításon kívül.

6.2.2. Segédfüggvények

A kiértékelő által használt segédfüggvények implementációit az `fparlin.h` fájl tartalmazza.

A `vectorize` túlterhelt függvény szerepe, hogy a kiértékelő függvényben a számítás eredményét akkor is vektorrá alakítsa, ha az skalár volt. Ha vektor, akkor semmilyen hatása nincs.

A `make_vector` függvény a kifejezésfa vektorképző műveletének hatását implementálja. A kapott generikus érték listát (`std::initializer_list<T>`) a megfelelő típusú vektorrá alakítja.

A magasabb rendű függvényeknek kész implementációi vannak az STL (Standard Template Library) algoritmusai között, így a (soros) `Map`, `Reduce` és `Zip` csak meghívják ezeket. A `Map` és a `Zip` az `std::transform` megfelelő változatára, a `Reduce` az `std::accumulate`-ra épül.

A párhuzamos változatok (`ParMap`, `ParReduce` és `ParZip`) először elosztják a kapott vektor(ok) méretét a hardver által támogatott párhuzamos szálak számával (`std::thread::hardware_concurrency()`), hogy megkapják az egy processzor szál által feldolgozandó elemek számát. Ezután elindítanak a támogatott számú `std::thread`-et amelyek mindegyike feldolgozza a paraméter vektor(ok) egy-egy megfelelő méretű részét a soros implementációval megegyező módon. A `Reduce` esetén az egyes szálak eredményeiből álló átmeneti vektort még egyszer redukálni kell a végeredmény kiszámításához.

6.3 Fordítás

Az FParLin elsődleges feladata véget ér a kiértékelő forráskódjának előállításával. Ezt már bárki hozzáteheti a saját programjához és használhatja a korábban felépített kifejezés kiértékelésére. De ez egy kényelmetlen felhasználási mód lett volna. Két külön programot kellett volna írni, egyet a faépítéshez és kódgeneráláshoz, egy másikat pedig az elkészült kód felhasználásához. A kiértékelendő kifejezés változtatásakor újra le kellett volna fordítani és futtatni

mindkét programot, és a generált eredmény fájlok verzióit is a felhasználónak kellett volna követnie.
Ezeket a nehézségeket kerültük el azzal, hogy az FParLin a generált kódot lefordítja, betölti, és visszaadja a kiértékelő függvény hivatkozását a felhasználó programjának, ami így rögtön használhatja azt.
A fordításhoz a `config.txt`-ben megadott fordító programot használja. A fordító fajtáját (Visual C++, clang vagy g++) kikövetkezteti a futtatható állomány nevéből. Visual C++ esetén a fejlesztői környezet által biztosított definíciókból megtudja, hogy a felhasználó milyen módban (Debug vagy Release) és milyen platformra (x86 vagy x64) akar fordítani, és ennek megfelelően állítja be a fordítási opciókat. Linuxon a Visual Studio hiányában ezek a definíciók nem állnak rendelkezésre, ezért egy előre meghatározott opció készlettel fordít.
Az elkészült dinamikus könyvtárból az operációs rendszernek megfelelő módon betölti a kiértékelő függvényt, és visszaadja a felhasználónak.

7. Tesztelés

Az FParLin teszteléséhez használt program verziók:
Windows 10 64-bit, Visual Studio 2015 Update 2, clang 3.9.0
Ubuntu 16.04 64-bit, g++ 5.3.1, clang 3.8.0
Ubuntu Server 14.04 64-bit, clang 3.6

7.1 Funkcionális tesztelés

Az FParLin működésének fő lépései a kifejezésfa építés, a transzformációs lépések (típus ellenőrzés, költségbecslés és kódgenerálás), és a generált kód fordítása és betöltése.
A kifejezésfa építés helyességét a típusok szintjén való implementációnak köszönhetően a C++ típus ellenőrzője biztosítja.
Az FParLin típus ellenőrzőjét az összes lehetséges típus hibát tartalmazó kifejezések kiértékelésével ellenőriztük. Ezek a kifejezések az `error_test.cpp`-ben vannak definiálva.

A generált kód szintaktikus és szemantikus helyességét a lefordításakor ellenőrzi a fordító program. Az egyes műveletek helyes működését a `functional_test.cpp`-ben külön erre a célra definiált kifejezések kiértékelésén keresztül ellenőriztük. Előnye a lambda-kalkulus használatának, hogy ha az absztrakció és alkalmazás művelet helyesen működik, akkor ez skálázódik tetszőlegesen összetett függvényekre is, így nem kellett nagyon sok különböző tesztet csinálni.
A fordítás és linkelés megbízható működését azzal igyekeztünk biztosítani, hogy a fejezet elején felsorolt többféle környezetben is teszteltünk.

7.2 Teljesítmény tesztelés

Teszt környezetek:
Laptop: Intel Core i5-3210M (fizikailag 2, logikailag 4 processzor mag), 8GB memória
HPC klaszter: 2 * Intel Xeon E5-2650 (fizikailag 16, logikailag 32 processzor mag), 32GB memória

7.2.1. Mérések

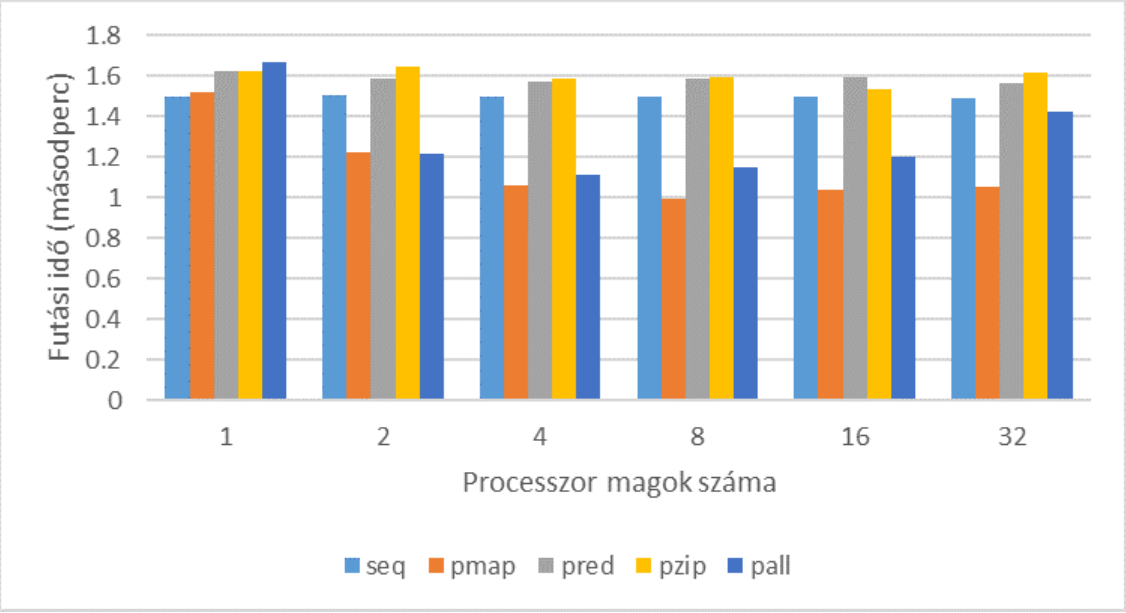
Az FParLin teljesítményét egy $16x10^7$-es mátrix-vektor szorzással, és egy 10^8 elemű vektor összegzésével mértük. A mátrix-vektor szorzásban három párhuzamosítható művelet van (lásd a 4.5.2 fejezet példáját). Lemértük a kiszámítási időt az egyes műveletek egyenkénti (pmap, pred, pzip), és mindhárom művelet párhuzamosítása (pall) esetén, valamint soros végrehajtással (seq). A vektor összegzésnél csak egy művelet párhuzamosítható, így csak soros (vseq) és párhuzamos (vpar) időket mértünk.
Minden időmérés 4 futtatásból állt, és ezeknek az időknek a minimumát vettük, mert az operációs rendszer működése szigorúan pozitív zajt ad a mérési eredményekhez.

Ezeket a méréseket elvégeztük a laptopon, és a klaszteren különböző számú processzort használva.

7.2.2. Eredmények:

	1	2	4	8	16	32
seq	1.494	1.501	1.495	1.494	1.498	1.491
pmap	1.519	1.219	1.063	0.991	1.038	1.052
pred	1.619	1.587	1.573	1.582	1.593	1.562
pzip	1.619	1.641	1.583	1.594	1.533	1.618
pall	1.669	1.216	1.114	1.149	1.197	1.42
vseq	0.055	0.074	0.055	0.074	0.057	0.055
vpar	0.077	0.037	0.033	0.031	0.037	0.034

1. Táblázat: Az egyes teszteken, a klaszteren különböző szál számok mellett mért futási idők másodpercben

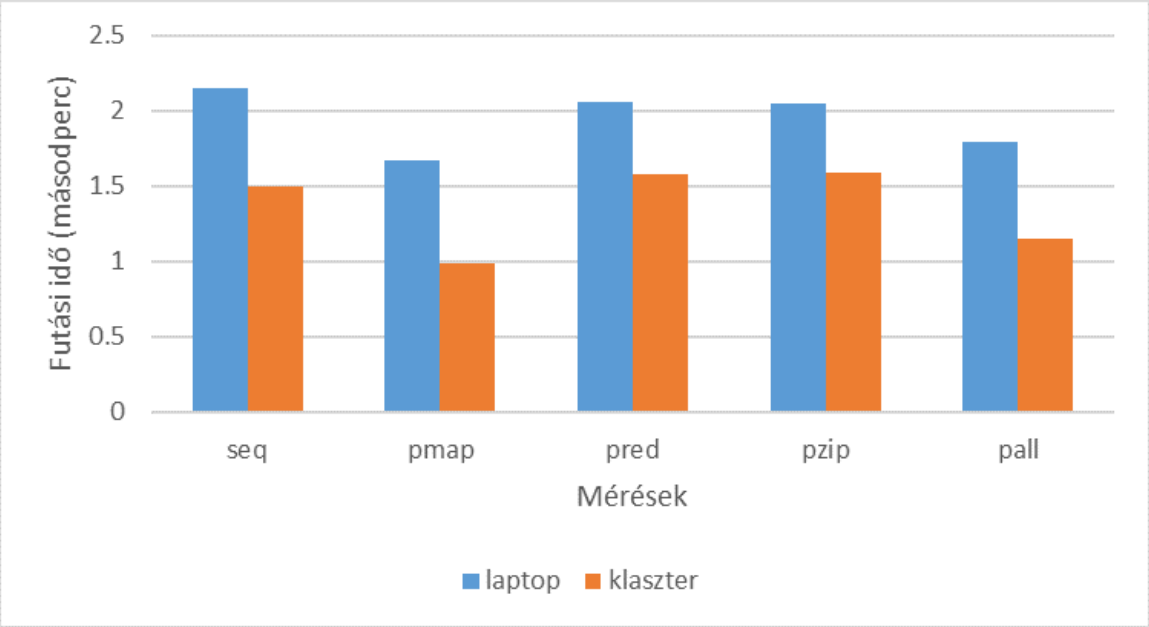


4. Ábra: Az 1. Táblázat adatainak szemléltetése.

	laptop (4 szál)	klaszter (8 szál)
seq	2.15	1.494

pmap	1.668	0.991
pred	2.055	1.582
pzip	2.05	1.594
pall	1.798	1.149
vseq	0.104	0.074
vpar	0.037	0.031

2. Táblázat: A két teszt környezetben mért futási idők másodpercben.



5. Ábra: A 2. Táblázat adatainak szemléltetése.

7.3 Konklúzió

A kategóriaelméleti módszereket alkalmazva egy könnyen kiterjeszthető és tesztelhető alkalmazást implementáltunk, amely képes az elemzéssel megalapozott processzor szintű párhuzamosításra. Ezzel az elsődleges célt, a megközelítés megvalósíthatóságának vizsgálatát és demonstrációját sikeresen teljesíti.

A teljesítmény elemzésből látható, hogy mindkét platformon lehet találni olyan párhuzamosítási határt (a map művelet párhuzamosításához tartozót), amellyel a

kiértékelés nem csak a sorosnál gyorsabb, de annál is, ha elemzés nélkül mindent párhuzamosítunk.
A két lényegesen különböző hardveren mért idők rendkívül hasonlóan viselkednek, tehát a mérésünk, és maguk a számítások is stabilak. Sajnos valószínűleg a mért idők jelentős részét a memória allokációk teszik ki, amik lényegében platform függetlenek. Ezt a problémát orvosolni fogja a szükséges memória előre lefoglalásra, amelyről a 8.2 fejezet ír bővebben.

8. Továbbfejlesztés

8.1 Felhasználói eszközök

Szeretnénk még többféle kifejezést egyszerűbben leírhatóvá tenni. Ehhez elengedhetetlen a mátrix nézetek bevezetése, de tetszőleges dimenziójú adatok támogatásán is gondolkodunk. Több elemi műveletet is fogunk implementálni, például hatványozást és gyökvonást.

8.2 Működés

Mint azt a Fejlesztő dokumentáció bevezetőjében kifejtettük, ez a program egy lényegesen leszűkített formája egy nagyobb elképzelésnek. Ennek megfelelően szeretnénk video kártyákra és klaszterekre is kiterjeszteni a párhuzamosítást. Ezeken a platformokon nem csak a végrehajtandó műveletek száma alapján kell majd párhuzamosítanunk, hanem a rendelkezésre álló memória szerint is, de az elemzések absztrakt szintje miatt ez nem igényel komoly változtatást.
A jelenlegi processzor szálak szintjén történő párhuzamosításon is lehetne még lényegesen gyorsítani a részeredmények tárolásához szükséges memória előre lefoglalásával. A fa elemzésekor össze tudjuk gyűjteni a rész kifejezések méreteit, és a generált kód betöltése után azonnal le tudnánk foglalni a kiértékeléshez szükséges memória blokkokat.

9. Irodalomjegyzék

1. ***MTA Wigner GPU Labor.*** **[Online] [Hivatkozva: 2016. 05. 17.] http://gpu.wigner.mta.hu/.**

2. *Boost C++ Libraries.* [Online] [Hivatkozva: 2016. 05. 17.] http://www.boost.org/.

3. Niebler, Eric. F-Algebras and C++. [Online] 2013. 07. 16. [Hivatkozva: 2016. 05. 17.] http://ericniebler.com/2013/07/16/f-algebras-and-c/.

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

5. Milewski, Bartosz. Understanding F-Algebras. *Bartosz Milewski's Programming Cafe.* [Online] 2013. 06 10. [Hivatkozva: 2016. 05. 17.] https://bartoszmilewski.com/2013/06/10/understanding-f-algebras/.

6. FParLin. *GitHub.* [Online] [Hivatkozva: 2016. 05. 17.] https://github.com/leanil/FParLin.