

Programkönyvtár véges adatfolyamok programozásához

Tudományos Diákköri Dolgozat

Nyilas Árpád
e-mail: nyilas.arpad@npcit.hu
évfolyam: 2.

Témavezető: Dévai Gergely tanársegéd
Programozási Nyelvek és Fordítóprogramok Tanszék,
ELTE Informatikai kar

2012. április 15.



Tartalomjegyzék

1. Bevezetés	2
1.1. Eredmények	2
1.2. Magas szintű optimalizálás, ciklusösszevonás (fusion)	3
1.3. Beágyazott nyelv	6
1.4. Feldspar	6
1.4.1. Feldspar Haskellbe ágyazása	7
1.4.2. A Feldspar alkalmazási területe	7
2. Jelenlegi megoldások	8
2.1. Vector	8
2.1.1. Hátrányai	12
2.2. Stream	16
2.2.1. Hátrányai	21
3. Seq	25
3.1. Alapötlet	25
3.2. Megvalósítás	25
3.2.1. A Seq típus definíciója	25
3.2.2. A Seq könyvtár függvényeinek implementációja	26
3.3. Az előző fejezetben felvetett problémák megoldása, mérési eredmények	38
3.3.1. A könyvtár teljessége	39
3.3.2. Felhasználóbarát interfész	39
3.3.3. Fusion	40
3.3.4. Mérési eredmények	44
4. Összefoglalás	48
4.1. Eredmények	48
A. Függelék	50
A.1. Generált kódok	50
A.1.1. Iir filter megvalósítása Vectorral	50
A.1.2. Vector és Stream függvényeinek kompozíciója	53
A.1.3. Seq függvényeinek kompozíciója	58

1. fejezet

Bevezetés

A dolgozatom témája ciklus optimalizáció megvalósítása véges streamekkel egy beágyazott funkcionális nyelvben. Ez magában foglalja a meglévő megoldások elemzését, a véges streamek megvalósítását Feldsparban és a megvalósított könyvtár függvényeinek hatékonyság elemzését.

A Feldspar egy digitális jelfeldolgozás specifikus nyelv, melynek jelenlegi fordítója C-kódra képes fordítani. A nyelv két típust is definiál az azonos típusú adatok sorozatának kezelésére, a Vectort és a Streamet. Ezek a felhasználási terület igényeinek megfelelően igen magas absztrakciót biztosítanak. Azért van szükség két ilyen típusra is, mert az egyikkel a véges algoritmusokat lehet hatékonyan kódolni, a másikkal pedig a rekurzívan definiált sorozatot előállító függvényeket. Ezek hiányossága az, hogy a kettő közti konverzió nagyban csökkenti a generált kód hatékonyságát.

A dolgozat célja egy olyan adatszerkezet és hozzá kapcsolódó könyvtár megvalósítása, melynek segítségével hatékonyan lehet komponálni egy rekurzívan definiált sorozatot előállító függvényt és egy, csak véges adatszerkezeten definiálható műveletet. Az alapötlet az, hogy a jelenlegi két módszer tulajdonságait ötvöző könyvtárat hozzunk létre.

Az általam definiált könyvtár véges streamek segítségével képes a fent említett két módszer jó tulajdonságait ötvözni. Ezt elméleti síkon is, és mérési eredményekkel is bizonyítom a dolgozatban. Az általam megvalósított könyvtár további jó tulajdonsága, hogy sokkal felhasználóbarátabb is mint, a végtelen streameket megvalósító könyvtár.

1.1. Eredmények

Megvalósításra került a Feldspar nyelvhez egy olyan könyvtár, mely véges adatfolyamok által olyan absztrakciót ad a programozó kezébe, mely segítségével könnyen kódolhat jelfeldolgozó algoritmusokat, anélkül, hogy törekednie kellene arra, hogy az algoritmus minimális számú konkatenációt vagy minimális számú rekurzív függvénnyel definiálható algoritmust tartalmazzon, ezzel elkerülve a fölösleges konverziókat.

A megvalósított könyvtár és típus a *Seq* nevet viseli. A *Seq* típus jól illeszkedik a Feldspar magnyelvéhez, így használata igen kényelmes.

A *Seq* könyvtár tartalmazza a funkcionális nyelvekben megszokott lista függ-

vények, Seq-re implementált változatát, az ezekből generált C-kódok hatékonysága nem rosszabb az elvártnál.

Elmondható, hogy rekurzív sorozatot definiáló függvény megvalósítható az elvárt $O(n)$ -es hatékonysággal. A jelenlegi megoldásokkal megvalósított, ilyen jellegű algoritmusok konstansszor jobb C kódot eredményeznek, ha önmagában használjuk őket. De a nyelv magas szintű absztrakcióinak pontosan az a lényege, hogy a függvények kompozíciója lehetőség szerint egyetlen ciklusban hajtsódjon végre. Amennyiben az ilyen jellegű algoritmusokat kompozícióban használjuk például egy, csak véges adatszerkezetekre definiálható függvénnyel, akkor a *Seq-kel megvalósított kompozíció konstansszor jobb*. Minél összetettebb a kompozíció, annál nagyobb ez a konstans, mint ahogy ez a mérési eredményekből is látszik.

A Seq típus további előnye, hogy valójában a már meglévő Vector típus általánosítása, így futásidejű költség nélkül konvertálhatunk Vectort Seq-é. Ez azért fontos, mert jelenleg a Feldsparban a Vectort használják leginkább azonos típusú elemek reprezentálására.

1.2. Magas szintű optimalizálás, ciklusösszevonás (fusion)

A magas szintű optimalizálás általában olyan transzformációkat hajtsd végre a programon, melyeket az adott programnyelven a forráskódot író programozó is megtehetne, de ezzel elrontaná a kód átláthatóságát. Tehát elsősorban azt a célt szolgálja, hogy a programozónak ne kelljen azzal foglalkoznia, hogy a kódja optimális legyen. Így nem kell olyan kompromisszumot kötnie, hogy feláldozza a kód átláthatóságát a gyors programért cserébe.

A magas szintű optimalizálás egyik módja, ha az egymás utáni program konstrukciókat megpróbáljuk összevonni. A dolgozat szempontjából a ciklusok összevonása az érdekes, mivel a jelfeldolgozás esetén a ciklusok a leggyakrabban és legkomplexxebben használt vezérlési szerkezetek.

A ciklusösszevonásoknak kézenfekvő módja, ha létrehozunk egy olyan absztrakciót, mely elfedi a tényleges ciklusokat. Általánosságban elmondható, hogy a ciklusokkal valamilyen adatok sorozatával dolgozunk. Vagy konkrét, tárolt adatok sorozatát dolgozzuk fel, vagy az adatok a ciklus futása során jönnek létre. Ezért érdemes egy olyan absztrakciót választani, mely a konkrét értékek tárolása helyett azok előállítását tárolja.

Ennek több ismert módja van ([2]). Ezek közül kettőt emelnék ki, a vektor és stream fusiónt. Ezeket a teljesség igénye nélkül mutatom be ([2], [3]).

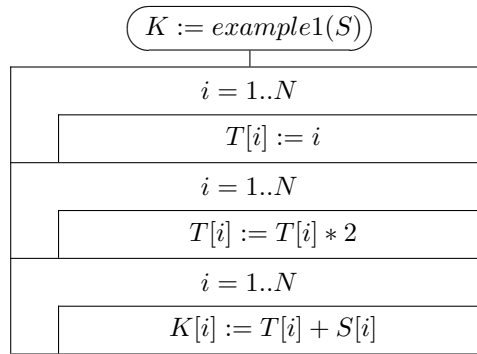
Vektor fusion. A vektor fusion olyan esetekben hatékony, mikor az adatok sorozatát akarjuk feldolgozni, de úgy, hogy a feldolgozás végeredménye vagy egyetlen érték, vagy egy olyan kimenő adatsor, ahol egy érték csak konstans számú bemenő értéktől függhet. Az alapelve az, hogy az elemek indexelő függvényeit tárolja az absztrakt adatszerkezet, ehhez a függvényhez kompozícióval kapcsolhatunk további függvényeket, így adva meg a végleges algoritmust. Fontos, hogy ezzel tényleg csak az algoritmus feladatát adtuk meg. Majd a végén előállítjuk egyetlen ciklussal az adatszerkezet elemeit.

Tehát a programozónak az absztrakciónak köszönhetően, elegendő csak azt

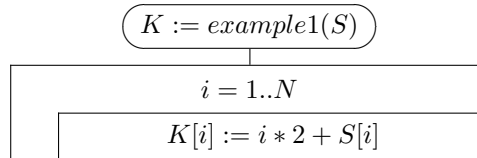
megadnia, hogy az adott sorozat elemeit hogyan kell kiszámítani. Így a programozónak nem kell figyelnie arra, hogy az algoritmus összekomponálása esetleg nem hatékony módon történik, hisz a végén úgylis egyetlen ciklus lesz a programban. Ezt jól illusztrálja az alábbi példa.

1. Példa. Állítsunk elő T tömböt úgy, hogy minden eleme egyenlő legyen az indexével. Majd szorozzuk meg minden elemét 2-vel. Tegyük fel, hogy van egy S tömb, ennek az elemeit páronként adjuk össze T tömb elemeivel K tömbbe. Feltehetjük, hogy minden tömb hossza N .

A feladatot megoldó stuktogram a következő.



Látható, hogy az algoritmus jó, viszont az alábbi megoldás hatékonyabb lenne, de nem olyan áttekinthető. Nyilván ilyen kis méretű példa esetén ez nem okoz gondot, de nagyobb feladat esetén igen.



Ezt az algoritmust kaphatjuk, ha alkalmazzuk a vektor fusiót. Legyen I a K kimeneti vektor index függvénye, I_i az i -ik összeállítási lépésben vett index függvény. És jelölje id az identitás függvényt. Ekkor definiáljuk I -t, a feladatnak megfelelően. Ez lépésenként a következőképpen nézhet ki.

1. $I_1(i) := i$
2. $I_2(i) := I_1(i) * 2$
3. $I(i) := I_2(i) + S[i]$

Tehát az I index függvény a következő lett: $I(i) = i * 2 + S[i]$. Vegyük észre, hogy ez megegyezik a kívánt algoritmus ciklusmagjában lévő értékadás jobb oldalával. Azaz a kívánt algoritmust megkapjuk, ha egy számlálós ciklusban a kimeneti tömb értékül kapja $I(i)$ értékét.

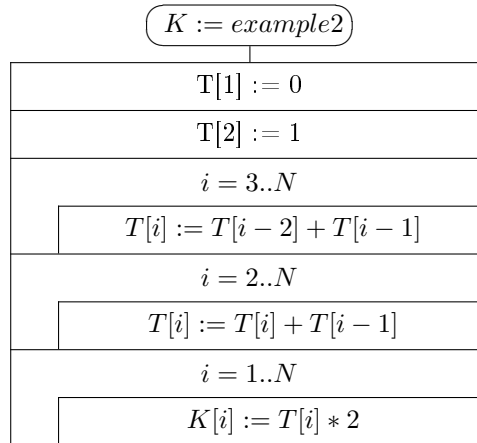
Stream fusion. A stream fusion olyan esetekben használható jól, ahol a kívánt algoritmus rekurzívan definiálható sorozatot ad vissza. Az alapötlete, hogy tárolja a kimeneti sorozat következő elemét – valamilyen paraméterek alapján – előállító függvényt (továbbiakban rákövetkezési függvény). Stream fusion használatánál a rákövetkezési függvényt adjuk meg, illetve ehhez komponálhatunk további függvényeket. Így egy végtelen adatsorozatot kapunk. Itt a ciklus előállításánál azt is meg kell mondanunk, hány elemet akarunk letárolni.

Tehát a programozó egy stream absztrakt adatszerkezetet transzformál, míg összekomponálódik a kívánt adatszerkezet. Itt is egyetlen ciklust generálunk. Ez esetben is igaz az, hogy az algoritmus már fordítási időben ismert, ezért megkomponálása már akkor megtörténhet.

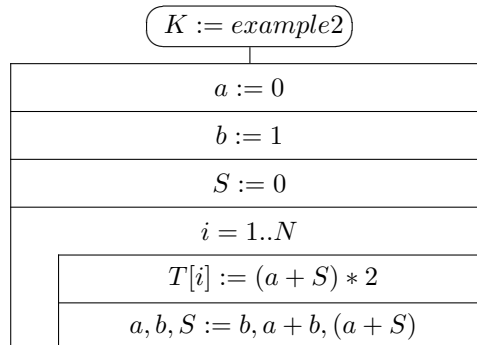
A stream fusion-t a következő példával lehet jól szemléltetni:

2. Példa. *Állítsuk elő a Fibonacci-sorozatot egy tömbben, majd állítsuk elő a részletösszegeit. Végül az így kapott értékeket szorozzuk meg kettővel. Itt is feltehetjük, hogy minden tömb hossza N .*

A feladatot megoldó algoritmus a következő:



Ez esetben is adhatunk hatékonyabb algoritmust az által, hogy a ciklusokat összevonjuk. A ciklusok összevonásával az alábbi algoritmust kaphatjuk.



Látható, hogy ez esetben nem volt olyan egyszerű a helyzetünk, hisz a ciklusaink nem minden esetben ugyanazt az intervallumot járták be. Nézzük, hogy

ezt hogyan adhatjuk meg stream fusion-nel. Legyen $I(x, y, s) = (e, (x', y', s'))$ a rákövetkezési függvény, ahol e az érték (x', y', s') , pedig az új állapot. Továbbá $A = (a, b, S)$ a kezdő állapot. Jelentse $I_i()$ az I leszűkítését az i -ik komponensre, I', I'' pedig az összeállítás közbülső lépéseiben vett rákövetkezési függvényeket.

Az alapötlet az, hogy a stream állapotában tároljuk a Fibonacci sorozat következő két elemét, és az eddigi részletösszeget. Tehát $a := 0, b := 1$ a Fibonacci-sorozat első két eleme, $S := 0$ a kezdeti részletösszeg. Most lépésenként rakjuk össze a rákövetkezési függvényt.

1. $I''(x, y) := (x, (y, x + y))$
2. $I'(x, y, s) := (I''_1(x, y) + s, (I''_2(x, y), x + s))$
3. $I(x, y, s) := (I'_1(x, y, s) * 2, (I'_2(x, y, s)))$

Tehát a rákövetkezési függvény $I(x, y, s) := ((x + s) * 2, (y, x + y, x + s))$, ekkor látható, hogy ez leírja a kívánt algoritmus ciklusmagjának értékadásainak jobb oldalát. Az állapot komponenseit segédváltozókban tároljuk és tartjuk karban, míg az előállított értékeket beírjuk a kimeneti tömbbe.

1.3. Beágyazott nyelv

A nyelvbeágyazás lényege, hogy egy alkalmas nyelvet gazdanyelvnek választva, az általa nyújtott eszközökkel definiálunk egy új nyelvet. Ez az új nyelv általában valamilyen alkalmazás terület - specifikus nyelv. Ez szolgálhat arra, hogy a gazdanyelvben könnyebben ki tudjunk fejezni bizonyos alkalmazásterület-specifikus dolgokat, de szolgálhat arra is, hogy egy önmagában használatos nyelvet hozzunk létre.

Számunkra az utóbbi eset lesz érdekes. Itt a gazdanyelvben az új nyelv egy olyan absztrakció, mellyel a programozó valójában egy absztrakt szintaxisfát épít. Ehhez írunk kell egy beágyazott fordítót is, mely ezt a szintaxis fát képes valamilyen alacsony szintű nyelvre lefordítani. Ezekhez a módszerekhez láthatunk példákat a [4], [5] hivatkozott irodalomban.

Az ilyesfajta beágyazás látszólag semmivel sem jobb, mintha egy teljesen önálló nyelvet fejlesztenénk, azonban ez esetben nem kell szintaktikus és szemantikus ellenőrzőt készítenünk a nyelvhez, hisz ezt a gazdanyelv megteszi helyettünk. Ezáltal könnyen készíthetünk prototípust a nyelvhez, könnyen kísérletezgethetünk benne különböző absztrakciókkal, a beágyazott fordítóban pedig a különböző transzformációkkal.

A mai funkcionális nyelvek megfelelő eszközkészletet nyújtanak ahhoz, hogy könnyen beágyazhassunk tetszőleges bonyolultságú nyelvet. Ezért a Feldspar is ily módon került implementálásra.

1.4. Feldspar

A Feldspar egy beágyazott funkcionális programozási nyelv digitális jelfeldolgozáshoz. Alapvetően egy telekommunikációs vállalat által támasztott igények kielégítésére készült. A nyelvről részletesen olvashatunk a [1] hivatkozott irodalomban. A nyelv MR5-ös változata elérhető a projekt weboldaláról [6]. Fontos

megjegyezmem, hogy az általam megvalósított könyvtár a következő verzióhoz készült.

A nyelv jelenlegi beágyazott fordítója C nyelvre fordítja a Feldspar kódot. Így aránylag hardverközeli, de mégis platformfüggetlen nyelvre fordíthat a Feldspar programozó.

1.4.1. Feldspar Haskellbe ágyazása

A Feldspar Haskellbe van ágyazva. A Haskell azért bizonyult jó gazdanyelvnek, mert a típusrendszere elég rugalmas ahhoz, hogy a beágyazott nyelv típusozását könnyen definiálhassunk. Az algebrai adattípus pedig lehetővé teszi saját, összetett típusok létrehozását, továbbá alkalmas az absztrakt reprezentáció olyan definiálására, hogy azon könnyen lehessen transzformációkat megadni.

Haskellbe van ágyazva a Feldspar magnyelve, ami olyan függvényeket tartalmaz, melyek a Feldspar fordító által fordítható típusokra vannak értelmezve. Ezek építik fel a fordítás bemenetét képező adatszerkezetet. A Feldsparban `Type a=>Data a` típusok a fordítható típusok, ahol a `Type` egy típus osztály, melyhez a Feldsparban használható típusok vannak példányosítva. A `Data` típus becsomagolja ezeket a típusokat, hogy ezzel elkülönítse a Feldspar típusokat a gazdanyelv típusaitól.

A Feldspar igazi erőssége, hogy a magnyelv fölé sokféle absztrakciót definiálhatunk. A nyelvhez definiált magas szintű típusoknak a `Syntactic` osztály példányainak kell lenniük ahhoz, hogy közvetlenül fordíthatók legyenek. Ez nem feltétlen szükséges, de ellenkező esetben az adott típus nem fordítható közvetlenül, konverziós függvényekre van szükség, melyek segítségével egy `Syntactic` típussá konvertálhatjuk.

1.4.2. A Feldspar alkalmazási területe

A beágyazott nyelv alkalmazási területe meghatározza a nyelv alapvető tulajdonságait. A Feldspart digitális jelfeldolgozó algoritmusok hatékony kódolására tervezték. Ezen algoritmusok többsége egy-egy bemenő értéksorozatot dolgoz fel, ebből vagy egy módosított értéksorozatot állít elő, vagy egy ezekből számított valamilyen összegző értéket.

A Feldspar alkalmazási területének sajátosságai miatt, a nyelv jól használhatóságának egy sarkalatos pontja a ciklusok hatékonysága. Ezért magas szintű könyvtárként megvalósítja az előző szakaszban taglalt stream és vektor összevonást.

2. fejezet

Jelenlegi megoldások

2.1. Vector

Jelenleg a Feldsparban a vektorok ábrázolásának egyik lehetősége a `Vector` könyvtár, aminek egy magas szintű könyvtára a Feldsparhoz. Magas absztrakciós szintet biztosít algoritmusok leírásához. Ebben az alfejezetben ezt a könyvtárat mutatom be, kitérve a hiányosságaira.

A `Vector` alapötlete az, hogy nem a tárolt értékeken manipulálunk közvetlenül, hanem egy `Vectoron`, ami valójában egy indexelő függvény kiterjesztése (vektor fusion, Lásd 1.2 fejezet). Azaz az algoritmusok definiálása során azt adjuk meg valójában, hogyan kell egy adott indexű értéket kiszámítani, így összetettebb algoritmusok esetén ciklus összevonásokat végzünk, anélkül, hogy tudatosan kellene figyelni a generálódó ciklusok felépítésére. A C-kódban tömbökön dolgozunk, tehát az így definiált értéksorozatot le kell tárolni, emiatt generálódik a kívánt ciklus (optimális esetben csak egy ciklus).

A `Vector` típus egy algebrai adattípus, mely lehet üres vektor vagy *Indexed*, utóbbinak három mezője van:

1. *segmentLength*, a szegmens hossza
2. *segmentIndex*, egy indexhez értéket rendelő függvény
3. *continuation*, a következő szegmens

Mint látható ez, az adattípus több szegmensből épül fel, minden szegmens egy *Vector*. Így könnyen konkatenálható lesz, csak az utolsó szegmenst, ami egy üres *Vector*, le kell cserélni a hozzá konkatenálandó *Vector*-ra. Tehát a definíciója:

```
data Vector a
  = Empty
  | Indexed
    { segmentLength :: Data Length
    , segmentIndex  :: Data Index -> a
    , continuation  :: Vector a
    }
```

Illetve a konkatenáció definíciója:

```
(++) :: Vector a -> Vector a -> Vector a
```

```

Empty          ++ v      = v
v              ++ Empty = v
Indexed l ixf cont ++ v   = Indexed l ixf (cont ++ v)

```

Tehát, ha adott egy n hosszú f -fel indexelt és egy m hosszú g -vel indexelt `Vector`, akkor a kettő konkatenációjának reprezentációja így néz ki: `Vector n f (Vector m g Empty)`.

Az `indexed` függvény használható arra, hogy `Vector`-okat definiáljunk, ennek első paramétere a vektor hossza, a második az indexelő függvény.

A `Vector` típus további előnye, hogy a vektor értékeit nem kell letárolni, hisz egy `Vector` nem az elemeket tárolja, hanem csak az indexelő függvényt.

Ahhoz, hogy fordítható típus legyen, meg kell írni hozzá néhány instance-t, hogy tagja legyen a `Syntactic` osztálynak. Ehhez szükséges, hogy konvertálható legyen listává (`freezeVector`), illetve listából készíthessünk `Vector`t (`unfreezeVector`). Előző esetben egyszerűen le kell tárolni az elemeket, míg utóbbi esetben a `Vector` indexelő függvényének a tömb indexelő függvényét kell megadni. Mivel a `Vector` tárolja a vektor hosszát is, így nem lehet túlindexelni.

A `Feldspar` tömböt `Vector`rá alakítani az `unfreezeVector` függvénnyel tudunk. Ez a függvény definiál egy `Indexed`-et, mely hossza a tömb hossza, és az indexelő függvénye a `getIx` magnyelvi függvény, mely a paraméterben kapott tömb paraméterben kapott indexű elemét adja vissza. A `Feldspar` kódja az alábbi:

```

unfreezeVector :: Type a => Data [a] -> Vector (Data a)
unfreezeVector arr = indexed (getLength arr) (getIx arr)

```

`Vector`t `Feldspar` tömbbé alakítani a `freezeVector` függvénnyel tudunk. Ez a függvény rekúzióval végigmegy a `Vector` szegmensein. A szegmenseket `parallel''` magnyelvi függvénnyel ténylegesen előállítja és letárolja a `Vector` elemeit.

```

freezeVector :: Type a => Vector (Data a) -> Data [a]
freezeVector vec = help True vec
  where
    help _ Empty = value []
    help opt (Indexed l ixf cont) =
      parallel'' opt l ixf \$ help False cont

```

A `Vector` típusra implementálhatók a funkcionális nyelvekben megszokott lista függvények, ezzel kényelmes interfészt adva a programozók kezébe. Ennek ellenére a könyvtár nem tartalmazza az összes ilyen függvényt, ugyanis némelyik nem a kívánt hatékonysággal implementálható. Nézzük, mely függvényeket tartalmazza a könyvtár.

A `Vector` véletlen címezhető konstans futásidővel a `'!` operátorral. Ennek implementációja összefuttatja a szegmenseket. Azaz a vektor indexelő függvénye a konkatenált részeket felüldefiniálja úgy, hogy megnézi minden esetben, hogy a lekérdezni kívánt érték indexe, mely szegmensbe esik, így a C-kódba egy elágazás generálódik a szegmensek számával megegyező számú ággal. Tehát a `random` elérés `Feldspar` kódja a következő:

```

instance Syntactic a => RandomAccess (Vector a)
  where
    type Element (Vector a) = a
    (!) = segmentIndex . mergeSegments

```

A felhasznált `mergeSegments` függvény implementációja pedig az alábbi. Ez a függvény összevonja a `Vector` szegmenseit, ezzel egy hosszú, egyszegmenses `Vector`-t hozva létre.

```
mergeSegments :: Syntactic a => Vector a -> Vector a
mergeSegments vec = indexed (length vec) (ixFun (segments
  vec))
where
  ixFun (Indexed l ixf _ : vs) = case vs of
    [] -> ixf
    _ -> \i -> condition (i<l) (ixf i) (ixFun vs (i-1))
```

Tekintsük a következő egyszerű példát, két egymáshoz konkatenált vektort véletlenszerűen elérő függvényt:

```
example a = ((indexed 3 id) ++ (indexed 2 id))!a
```

Ebből a következő C kód fordul, amelybe, mint látható belekerült a fent említett elágazás, tehát a futásideje ugyan konstans idejű, de a `Feldspar` kód megírásakor nem biztos, hogy tudjuk hány szegmenses függvényre fogjuk meghívni, így azt mondhatjuk, hogy a futásidő függ a szegmensek számától. Így a generálódó kód ugyan konstans idejű, de a `Feldspar` kód futásideje valójában $O(s)$, ahol s a szegmensek száma.

```
void test(struct array mem, uint32_t in0, uint32_t * out1)
{
    if((in0 < 3))
    {
        (* out1) = in0;
    }
    else
    {
        (* out1) = (in0 - 3);
    }
}
```

Az alábbiakban a teljesség igénye nélkül bemutatom a könyvtár további függvényeit.

1. take

Visszaadja a paraméterben kapott vektor paraméterben kapott hosszú részét, műveletigénye $O(m)$, ahol m a paraméterben kapott érték.

2. drop

Elhagyja a paraméterben kapott vektor paraméterben kapott hosszú részét, műveletigénye $O(m)$, ahol m a vektor hossza a paraméterben kapott értékkel csökkentve.

3. last

Visszaadja a paraméterben kapott vektor utolsó elemét, ennek műveletigénye a C-kódban csak a szegmensek számától függ.

4. head

Visszaadja a paraméterben kapott `Vector` első elemét.

5. tail

Levágja a paraméterben kapott Vector első elemét.

6. map

A paraméterben kapott vektor minden elemére végrehajtja a szintén paraméterben kapott függvényt.

7. zip

Két vektorból párok vektorát hozza létre, ahol a pár első tagja az első vektor megfelelő indexű eleme, a második a második vektor megfelelő indexű tagja.

8. unzip

A zip függvény inverze.

9. zipWith

Két vektor elemeire páronként végrehajtja a paraméterben kapott két-paraméteres függvényt.

10. fold

Paraméterben kapott kezdőértékkel, a szintén paraméterben megadott kétparaméteres függvénnyel összesíti a vektor elemeit.

11. fold1

A vektor első elemét véve kezdőértéknek, a paraméterben megadott két-paraméteres függvénnyel összesíti a vektor elemeit.

12. sum

Összeadja a vektor elemeit.

13. maximum

Kiválasztja a vektor maximális elemeit.

14. tails

Visszaadja a paraméterben kapott vektor posztfixeinek vektorát.

15. inits

Visszaadja a paraméterben kapott vektor prefixeinek vektorát.

A fent felsorolt funkcionális programozási nyelvekben megszokott lista függvényeket implementálja a könyvtár Vectorra, elvárt vagy annál jobb hatékonysággal.

A Vector típus sajátosságai miatt konstans futásidejű permutációs függvények írhatók, hisz csak az indexelő függvényt kell felüldefiniálni, ami a fordított C kódban nyilván nem jelenik meg közvetlenül. Például ilyen a **permute**, mely permutálja a paraméterben kapott vektor elemeit a paraméterben kapott permutációs függvény szerint. Ez valójában egy kompozíció az indexelő függvény és a permutáció közt, többszegmenses *Vector* esetén itt is össze kell fésülni a szegmenseket.

További előnye, hogy a `Vector`-ra implementálva van az intervallum megadás lehetősége. Az definiálva van a `(...)` operátor, mely az intervallum két végpontját paraméterül kapva visszaad egy olyan `Vector`-t, melynek elemei e két szám által meghatározott intervallum elemei.

2.1.1. Hátrányai

Mint említettem, a `Vector` típus nagy hátránya, hogy néhány igen gyakran használt listafüggvény nem implementálható rá a kívánt igényeknek megfelelően, illetve ennél is nagyobb probléma, hogy jó néhány filter algoritmus sem valószínűleg meg vele hatékonyan. Tekintve, hogy a `Feldspar` egy jelfeldolgozás specifikus nyelv, ez jelentősen csökkenti a nyelv könnyen használhatóságát. Az alábbiakban néhány példán keresztül bemutatom ezeket a problémákat.

Scan. A `scan` függvény függvény matematikailag legkönnyebben talán a következőképpen fogalmazható meg:

1. Definíció. $\text{scan}(*, a, X) = Y$, ahol $*$ egy művelet, $Y_n = f(n)$, ahol $f(0) = a$ és $f(n) = f(n-1) * x_{n-1}$ ($n > 0$)

Ezen definíció alapján könnyedén implementálhatjuk a `Vector` típusra a `scan` függvényt, ez még könnyebb, ha felhasználjuk a fent felsorolt függvényeket. Ekkor a definíció a következő képpen néz ki:

```
scanVector :: (Type a)
            => (Data a -> Data a -> Data a)
            -> Data a
            -> V.DVector a
            -> V.DVector a
scanVector f x y = V.map (V.fold f x) (V.inits y)
```

Ez a kód a definíciónak tökéletesen megfelel, a baj az, hogy nem lesz hatékony. Ha jobban megnézzük, láthatjuk, hogy ugyan látszólag $O(n)$ műveletidejűnek tűnhet, hisz alapvetően egy `map` függvény az egész, csak hogy az imént az indexelő függvényt eleminek tekintettük. Könnyen beláthatjuk, hogy jelen esetben ez semmiképp sem áll fenn, hiszen az indexelő függvény egy "összegzés" lesz, ami szintén $O(n)$ művelet igényű. Így azt sejtjük, hogy a generált C kód $O(n^2)$ műveletigényű lesz, amiről a függvény fordítása után meg is bizonyosodhatunk.

Vegyük a következő egyszerű példát:

```
((scanVector (+) 0) :: V.DVector Int32 -> V.DVector Int32)
```

Ha ezt lefordítjuk, a következő eredményt kapjuk. Látható, hogy a kód valóban $O(n^2)$ műveletigényű.

```
void test(struct array mem, struct array in0, struct array *
          out1)
{
    uint32_t v2;

    v2 = length(in0);
    setLength(out1, (v2 + 1));
    {
        uint32_t i3;
```

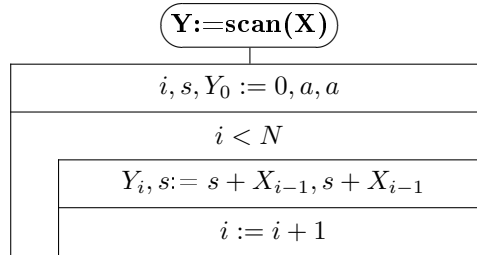
```

for(i3 = 0; i3 < (v2 + 1); i3 += 1)
{
    int32_t temp5;
    uint32_t w4;

    if((i3 < v2))
    {
        w4 = i3;
    }
    else
    {
        w4 = v2;
    }
    at(int32_t, (* out1), i3) = 0;
    {
        uint32_t i6;
        for(i6 = 0; i6 < w4; i6 += 1)
        {
            temp5 = (at(int32_t, (* out1), i3) + at(
                int32_t, in0, i6));
            at(int32_t, (* out1), i3) = temp5;
        }
    }
}
}

```

A scan-t egy rekurzív függvénnyel definiáltuk, aminek egy konstans kezdőértéke van, tehát visszavezethetjük a rekurzív függvény helyettesítési értékének kiszámítására vonatkozó programozási tételre. Ennek alapján az előző példa imperatív megoldásának egyik lehetséges módja – a teljesség igénye nélkül – az alábbi stuktogrammon látható.



Vagyis a függvény definiálható $O(n)$ műveletigénnyel is. Viszont a *Vector* típus sajátosságai miatt nem tudjuk a scan-t *Vector*-okra implementálni ilyen műveletigénnyel. Ezt könnyen be is láthatjuk. Az indexelő függvénynek egy indexhez hozzá kell tudnia rendelni a kívánt értéket, de ennek műveletigénye az index nagyságától függ, hisz más paramétert nem is kap a függvény. Viszont ezt mind az N darab értékre ki kell számítani, tehát a legkisebb műveletigény az $\Theta(\sum_{i=0}^N i) = \Theta(N^2)$.

Iir. Egyik fontos jelfeldolgozó filter az iir (Infinite impulse response), ez már egy kissé összetettebb példa. Több definíciója is ismert, a következőkben az alábbi definíciót fogjuk alkalmazni

2. Definíció. Legyen a, b, X, Y vektorok, ekkor $iir(a, b, X) = Y$, ahol $Y_n = (\sum_{i=0}^P b_i X_{n-i}) - (\sum_{i=1}^Q a_i Y_{n-i})$, ahol P az a Q a b vektor hossza. (Gyakran az a_0 -val osztják még ezt.)

Látható, hogy itt is egy rekurzív függvénnyel állunk szemben, nyilván itt is hasonló akadályokba fogunk ütközni mint a scan esetén. Mivel minden i . indexű érték kiszámítása $\Omega(i)$ műveletigényű, és i 1-től $N - P$ -ig vesz fel értékeket, így sajnos ezt az algoritmust sem tudjuk a kellő hatékonysággal implementálni *Vector*okra. Mint, ahogy azt az alábbi kód is mutatja:

```
vIir :: V.DVector Float
      -> V.DVector Float
      -> V.DVector Float
      -> V.DVector Float
vIir a b x@(V.Indexed l ixf cont) = res l
  where
    res l = V.indexed l ixf'
    ixf' j = snd $ forLoop (j+1) (initState) tr'
    where
      initState = (((V.freezeVector $ V.indexed (V.length a)
        (\i -> 0.0::Data Float)) , (0::Data Index)) ,
        ((V.freezeVector $
          V.indexed (V.length b) (\i -> 0.0::Data Float)) ,
          (0::Data Index)) , 0::Data Float)
      tr' i ((firstVec, secVec), _) = (firstVec', secVec'), newa
        )
    where
      (s', olda) = (s', ixf i)
      firstVec' = insert firstVec newa
      secVec'   = insert secVec olda
      insert (s2, s3) val =
        (setIx s2 s3 val, (s3+1::Data Index) `mod` (getLength
          s2))
      newa =
        ((sumVec b secVec' secVec') - (sumVec a firstVec
          firstVec))
      sumVec v (s2, s3) vec = V.sum $ V.zipWith (*) v
        ( V.reverse $ V.rotateVecR
          ((getLength s2)-s3) $ V.unfreezeVector (fst vec))
      uf = V.unfreezeVector $ V.freezeVector x
```

A belőle generált C kód a függelékben található(A.1.1), alább pedig látható ennek számunkra érdekes része. Megfigyelhető, hogy valóban $O(n^2)$ -es kódot kaptunk.

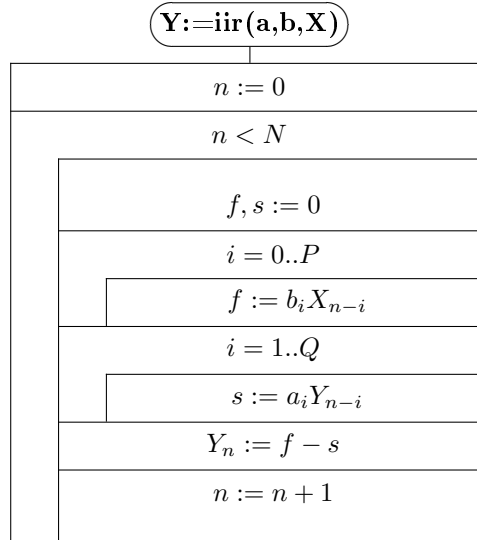
```
...
for(i10 = 0; i10 < length(in2); i10 += 1)
{
    ...
    {
        uint32_t i13;
        for(i13 = 0; i13 < (i10 + 1); i13 += 1)
        {
            ...
            w22 = 0.0f;
```

```

{
    uint32_t i25;
    for(i25 = 0; i25 < w23; i25 += 1)
    {
        temp24 = (w22 + (at(float,in1,i25) *
                        at(float,at(struct array,mem,6)
                        ,(v17 - (((v17 - (v17 - i25)) +
                                v18) % v15))))));
        w22 = temp24;
    }
}
...
{
    uint32_t i29;
    for(i29 = 0; i29 < w27; i29 += 1)
    {
        temp28 = (w26 + (at(float,in0,i29) *
                        at(float,at(struct array,mem,2)
                        ,
                        (v20 - (((v20 - (v20 - i29)) +
                                v21) % v19))))));
        w26 = temp28;
    }
}
...

```

Hatékonyabb imperatív algoritmus létezését pedig a következő stuktogram-ban látható algoritmus bizonyítja. Látható, hogy ennek műveletigénye, ha a -t és b -t konstansnak tekintjük (ezek általában konstansok a gyakorlatban is), akkor $O(n)$, ahol n az X vektor hossza.



2.2. Stream

Feldsparban másik lehetőség azonos típusú adatok sorozatának kezelésére a *Stream* típus. Ennek a típusnak, és a hozzá tartozó könyvtárnak a fő célja az előző alfejezetben taglalt *Vector* típussal kapcsolatos problémák megoldása. Tehát az, hogy egy adott indexű elem értékének kiszámításakor könnyen felhasználhassuk az eddig már kiszámított értékeket. Ebben az alfejezetben a *Stream* könyvtárt fogom bemutatni, mely absztrakciót ad a stream fusion használatához (stream fusion-t lásd: 1.2). Megmutatom, hogy a *Stream* könyvtár nem rendelkezik a *Vector*-nál taglalt hátrányokkal, viszont ezen jó tulajdonságokért a *Vector* sok jó tulajdonságát feláldozza.

A *Stream* típus – a *Vector*-hoz hasonlóan – szintén algebrai adattípus, de csak egyetlen konstruktora van, mely 2 paraméterrel rendelkezik, egy rákövetkezői függvénnyel és egy kezdő státusszal (állapot). A rákövetkezői függvény egy státuszhoz hozzárendel egy értéket és egy státuszt, tehát az értékeket úgy kaphatjuk, hogy a rákövetkezői függvényt meghívjuk a kezdő státuszra, így megkapjuk az első értéket és egy újabb státuszt, melyre újból meghívható a függvény, kapva egy újabb értéket és egy státuszt, és így tovább. Tehát a *Stream* típus végtelen sok értéket tartalmaz. A definíciója a következő:

```
data Stream a where
  Stream :: Syntactic state
          => (state -> (a, state))
          -> state
          -> Stream a
```

Látható, hogy ez a típus is egyparaméteres, csak az érték típusa paraméter, a státusz bármilyen Syntactic típus lehet.

A státuszban eltárolhatók és karban tarthatók a rekurzív függvények hatékony definiálásához szükséges értékek, így azokat nem kell minden érték meghatározásához újraszámolni.

Azaz, ha adott egy rekurzív sorozat, akkor a kezdő státusz legyen a sorozat bázisa. A rákövetkezői függvényt pedig definiáljuk úgy, hogy az megfelelően karbantartsa az állapotot, és értékként a sorozat következő elemét adja vissza.

Vegyünk egy egyszerű rekurzív függvényt, a Fibonacci sorozatot, ennek a definíciója például lehet a következő:

```
Stream (\ (s1,s2) -> (s1,(s2,s1+s2)) ) (0,1)
```

A *Stream* típusra nem lehet megvalósítani az összes funkcionális programozási nyelvekben gyakran használt lista függvényeket, ahogy azt a következőkben bemutatom majd. De előbb a legfontosabb implementált függvényeket mutatom be, a teljesség igénye nélkül.

1. head

Visszaadja a *Stream* első elemét. Azaz a *Stream* kezdőértékére alkalmazza a rákövetkezői függvényt, majd az így kapott rendezett párnak veszi az első tagját. Ez konstans műveletidejű.

A definíciója a következő:

```
head :: Syntactic a => Stream a -> a
head (Stream next init) = fst $ next init
```

2. tail

Visszaadja a Streamet az első eleme nélkül, azaz a Stream első elemére alkalmazza a rákövetkezési függvényt, majd az így kapott rendezett pár második tagját veszi kezdőállapotnak az új Streamben, az indexelő függvényt változatlanul hagyja. Ez szintén konstans műveletigényű lehetne, de mivel a C kódban tömbökkel dolgozunk, a bemenő tömb megmaradó részének tartalmát át kell másolni egy egxel kisebb tömbbe. Nyilván a Streamnek csak egy véges részét tároljuk le.

Tehát a Feldspar kódja:

```
tail :: Syntactic a => Stream a -> Stream a
tail (Stream next init) = Stream next (snd $ next init)
```

3. map

A paraméterben kapott függvényt alkalmazza a paraméterben kapott Stream minden elemére. Ezt úgy teszi, hogy a rákövetkezési függvény első elemre való megszorítását összekomponálja a paraméterben kapott függvénnyel. Ez minden elem esetén konstans műveletigényű, a letárolandó értékek arányában a függvény lineáris műveletigényű.

```
map :: (Syntactic a, Syntactic b) =>
      (a -> b) -> Stream a -> Stream b
map f (Stream next init) = Stream newNext init
  where newNext st = let (a,st') = next st in (f a, st')
```

A *Stream* könyvtár implementálja a map függvény különböző változatait is, de mivel ezek a maptól implementációban, és hatékonyságot tekintve nem térnek el, nem részletezem őket.

4. zip

Paraméterben kap két *Stream*et, az eredmény pedig egy olyan *Stream*, melynek *i*. elemei olyan rendezett párok, hogy az első tagja az első paraméter, a második tagja a második paraméter *i*. eleme. Ennek legegyszerűbb módja, ha előállítunk egy olyan Streamet, aminek az állapota is egy pár, az első tagját az eredmény első tagjának, a második tagját az eredmény második tagjának előállítására használjuk. Ennek műveletigénye $O(n)$, ahol n a Streamből készítendő vektor hossza. A Feldspar kódja:

```
zip :: Stream a -> Stream b -> Stream (a,b)
zip (Stream next1 init1) (Stream next2 init2)
  = Stream next (init1,init2)
  where next (st1,st2) = ( (a,b), (st1',st2') )
        where (a,st1') = next1 st1
              (b,st2') = next2 st2
```

5. zipWith

A zip függvényhez nagyon hasonlóan működik, azzal a különbséggel, hogy nem egyszerűen párokat készít, hanem a paraméterben kapott binér műveletet alkalmazza az értékpár tagjaira. Nyilván itt is az állapot a két paraméter Stream állapotainak párja lesz, az értéket pedig úgy kapjuk,

hogy előállítjuk mindkét paraméterben kapott Stream értékét, majd alkalmazzuk a paraméterben kapott műveletet. Ennek műveletigénye is $O(n)$, ahol n a Streamből készítendő vektor hossza. A Feldspar kódja pedig az alábbi:

```
zipWith :: Syntactic c
        => (a -> b -> c)
        -> Stream a
        -> Stream b
        -> Stream c
zipWith f (Stream next1 init1) (Stream next2 init2)
  = Stream next (init1,init2)
  where next (st1,st2) = ( f a b, (st1',st2'))
        where (a,st1') = next1 st1
              (b,st2') = next2 st2s
```

6. unzip

Fekete dobozként nézve az unzip a zip függvény inverze. A párok Stream-jéből állít elő Stream-ek párját, de ezt nem tudja úgy megtenni, hogy az állapotot is szétbontja tagjaira, hisz párok Stream-jét nem csak zip függvénnyel lehet definiálni, nem biztos, hogy az állapot is pár. Így a zip.unzip nem identitás függvényként viselkedik. A műveletigénye $\Theta(n)$, ahol n a Streamből készítendő vektor hossza. A következőkben látható, hogy az implementációja valójában két map függvény.

```
unzip :: (Syntactic a, Syntactic b)
       => Stream (a,b)
       -> (Stream a, Stream b)
unzip stream = (map fst stream, map snd stream)
```

7. drop

Elhagyja a Stream első n elemét. Ehhez ki kell számítani az $n-1$. állapotot, ez lesz a Stream új kezdő állapota. Ehhez jól használható a forLoop magnyelvi függvény. Ennek műveletigénye is $O(n)$, ahol n az elhagyandó elemek száma. A felhasználandó elemek letárolása további $O(m)$ műveletet igényel, ahol m a felhasználni kívánt elemek száma.

```
drop :: Data Length -> Stream a -> Stream a
drop i (Stream next init) = Stream next newState
  where newState = forLoop i init body
        body _   = snd . next
```

8. repeat

A repeat függvény implementációjából jól látható, hogy hogyan kell egyszerű Streameket definiálni. Ez a függvény alapvetően azt teszi, hogy létrehoz egy olyan Streamet, mely minden eleme a paraméterben kapott érték.

```
repeat :: Syntactic a => a -> Stream a
repeat a = Stream next (value ())
  where next _ = (a,value ())
```

9. iterate

Az `iterate`-tel összetettebb Streameket definiálhatunk, melyek egy tetszőleges elsőrendű rekurzív függvény értékeit tartalmazzák. A függvény főleg kényelmi célokat szolgál, de jól mutatja, hogy a Streamekkel milyen könnyű rekurzív sorozatokkal dolgozni.

```
iterate :: Syntactic a => (a -> a) -> a -> Stream a
iterate f init = Stream next init
  where next a = (a, f a)
```

10. unfold

Technikai jellegű függvény, a `Stream` konstruktor kidelegálása, mivel a konstruktor a könyvtáron kívül nem használható.

```
unfold :: (Syntactic a, Syntactic c) => (c -> (a,c)) ->
  c -> Stream a
unfold next init = Stream next init
```

11. take

A `take` függvény is ismerős lehet a lista függvények közül. De ez esetben nem az elvárásainknak megfelelően működik, azt gondolhatnánk, hogy kap egy Streamet és egy számot, hogy hány értéket szeretnénk megtartani belőle, de mivel a Stream végtelen, ez nem megvalósítható. Ezért `Feldspar` tömböt ad vissza, azaz letárolja a megtartani kívánt elemeket. Ez annyi kényelmetlenséget jelent, hogy ezt a függvényt kell utolsónak alkalmazni, ha ki akarjuk használni azt, hogy Streammel a `Feldspar` kódban nem látható ciklus összevonásokat végezhetünk el.

A műveletigénye $O(n)$, hisz le kell tárolnia n darab elemet. A `Feldspar` kódja a következő:

```
take :: (Type a) => Data Length -> Stream (Data a) ->
  Data [a]
take n (Stream next init)
  = sequential n init step (const $ value [])
  where step i st = next st
```

12. Véletlen elérés

A véletlen eléréshez végig kell számolni az összes előző állapotot, majd erre az állapotra még egyszer alkalmazni a rákövetkezési függvényt, azzal, hogy itt az első komponens (a rendezett pár érték tagját) használjuk fel (ez a keresett érték). Ehhez szintén jól használható a `forLoop` magnyelvi függvény. Ennek műveletigénye $O(n)$, ahol n az elérni kívánt elem indexe. Az alábbi `Feldspar` kódjából látható az is, hogy a `Stream` könyvtár tagja a `RandomAccess` osztálynak.

```
instance Syntactic a => RandomAccess (Stream a) where
  type Element (Stream a) = a
  (Stream next init) ! n = fst $ forLoop n (next init)
    body
  where body _ (_,st) = next st
```

13. recurrenceO

Segítségével könnyen hozhatunk létre konstans Streameket. Ez valójában csak egy kényelmi célokot szolgáló kombinátor. Paraméterben kap egy Vectort, mely a létrehozandó Stream állapota lesz, és egy függvényt, mely a rákövetkezési függvény.

14. recurrenceI

Ez a függvény is csak egy kombinátor, hasonlóan a `recurrenceO`-hoz. Ezzel olyan Streameket hozhatunk létre, ahol az állapotban a Stream előző m darab elemét tárolja a bemenő Streamnek. Ennek kezdőértékét paraméterben kell megadni, ennek a vektornak a hossza határozza meg az m -et. A rákövetkezési függvényt szintén paraméterben adjuk meg.

15. recurrenceIO

Kombinátor az előbbi kettő kombinációja. Azaz olyan Streameket definiálhatunk velük, melyek állapota két Vector. Az egyik az előállított Stream előző m elemét, a másik pedig a bemenő Stream előző n elemét tárolja. A Feldspar kódja a következő:

```
recurrenceIO ii (Stream st s) io mkExpr
  = Stream step (ibuf,obuf,s,0)
  where ibuf = freezeVector ii
        obuf = freezeVector io
        p    = getLength ibuf
        q    = getLength obuf
        step (ibuf,obuf,s,ix) =
          let (a,s') = st s
              ibuf'  = p /= 0 ? (setIx ibuf (ix 'rem' p) a, ibuf)
              )
          b = mkExpr
              (indexed p (\i->getIx ibuf' ((i+ix) 'rem' p)))
              (indexed q (\i->getIx obuf ((i+ix-1) 'rem' q))
              )
          in (q /= 0 ? (getIx obuf (ix 'rem' q),b),
              (ibuf'
               ,q /= 0 ? (setIx obuf (ix 'rem' q) b,obuf)
               )
              ,s'
              ,ix + 1))
```

16. scan

A matematikai definícióját lásd a Vector tárgyalásánál (2.1, 1. definíció). A Vector könyvtárnál láthattuk, hogy a `scan` függvény nem valósítható meg Vectorokkal hatékonyan. Láttuk, hogy a Streamekkel viszont jól lehet rekurzív sorozatokat számítani. A várakozásainknak megfelelően a Streammel $O(n)$ műveletigénnyel lehet megvalósítani a `scan` függvényt. A részletösszegeket az állapotban tároljuk, ez alapján konstans idő alatt kiszámíthatjuk a sorozat következő értékét. Csak alkalmaznunk kell a paraméterben kapott műveletet az állapotban tárolt értékre és a paraméterben kapott Stream értékére. Ez utóbbi számításához meg kell őriznünk a bemeneti Stream állapotait is. Tehát az új Stream állapota egy pár lesz. A Feldspar kódja pedig a következő:

```

scan :: Syntactic a
      => (a -> b -> a)
      -> a
      -> Stream b
      -> Stream a
scan f a (Stream next init)
      = Stream newNext (a,init)
      where newNext (acc,st) = let (a,st') = next st
                                in (acc, (f acc a,st'))

```

A könyvtár definiálja a `scan1` függvényt is, mely csak annyiban tér el, hogy a bemeneti Stream első elemét veszi kezdőértéknek.

17. iir

Az `iir` egy filter algoritmus. A matematikai definícióját lásd a Vector tárgyalásánál (2.1, 2. definíció). Ez a filter is könnyen megvalósítható a fenti kombinátorok segítségével, ahogy azt az alábbi implementáció is mutatja:

```

iir :: Data Float -> DVector Float -> DVector Float ->
      Stream (Data Float) -> Stream (Data Float)
iir a0 a b input =
  recurrenceIO (replicate (length b) 0) input
              (replicate (length a) 0)
              (\input output -> 1 / a0 *
                ( scalarProd b input
                  - scalarProd a output)
              )

```

Ezt a függvényt sem lehetett Vectorral hatékonyan megvalósítani, mivel ez is rekurzív sorozatként van definiálva. A Stream esetén $O(n)$ műveletigénnyel valósítható meg, feltéve, hogy a és b vektorokat konstansnak tekintjük, de ezek a gyakorlatban is konstansok.

2.2.1. Hátrányai

Láthattuk, hogy a Vector által felvetett problémákat megoldotta a Stream típus, most nézzük, milyen áron. A Stream könyvtár nem tartalmaz `freeze` és `unfreeze` függvényeket, így viszont nem lehet elkészíteni az `instance-t` a Syntactic osztályhoz. Ennek elvi korlátja van, ugyanis a Stream végtelen, így nyilván nem lehet letárolni az elemeit véges tárterületen, és véges tárterületről sem lehet végtelen számú elemet beolvasni. Tehát közvetlenül nem fordíthatók a függvények. Ezért definiál a könyvtár 2 függvényt, mely segítségével Vector függvényeké konvertálhatjuk a Stream függvényeket.

A `unsafeVectorToStream` függvény a Vectort Streammé konvertálja, de mivel a vektorok végesek, így könnyen túlfuthatunk a feldolgozásnál a vektor hosszánál, aminek nem definiált hatása. Ezért ez a függvény a Stream könyvtáron kívül nem látható. Viszont a könyvtár a segítségével definiál két függvényt, melyek a Streamről Streamre képező függvényt Vectorról Vectorra képező függvénné konvertálják.

A `streamAsVector` függvény egy Streamről Streamre képező függvény bemeneti Streamjét egy Vectorból állítja elő, majd a kimeneti Streammet Vector-rá

konvertálja. De ehhez le kell tárolnia a Stream elemeit egy Feldspar tömbben, amit már lehet Vectorrá alakítani, mivel a Stream közvetlenül nem alakítható Vectorrá.

Így viszont az algoritmusok memóriaigénye növekedhet meg, hisz ha egy algoritmusba vegyesen építünk Vector és Stream függvényeket is, a konverzióknál az eredmény mindig letárolódik, míg a tisztán Vector függvényekkel definiált algoritmusoknál ez nem állt fenn. Ezen függvények definíciója:

```
unsafeVectorToStream :: Syntactic a => Vector a -> Stream a
unsafeVectorToStream vec = Stream next 0
  where next i = (vec ! i, i + 1)

streamAsVector :: (Type a, Type b) =>
  (Stream (Data a) -> Stream (Data b))
  -> (Vector (Data a) -> Vector (Data b))
streamAsVector f v
  = unfreezeVector
    $ take (length v) $ f $ unsafeVectorToStream v
```

Láthatjuk, hogy itt használja a take függvényt, ami – mint, ahogy azt már kifejtettem – nem Streamet ad vissza, hanem listát, melynek hossza a kapott vektor hossza. Így biztos, hogy a Stream nem definiált működésű része nem értékelődik ki. De mi van akkor, ha az eredmény vektor hosszának nem kell egyeznie a bemeneti vektor hosszával? Ez esetben szükséges egy olyan függvényt definiálni, ami meghatározza a bemeneti vektor hossza alapján a kimeneti vektor hosszát. Ezt a függvényt paraméterül adva a streamAsVectorSize függvénynek ez a probléma is megoldódik, ez utóbbi függvény definíciója a következő:

```
streamAsVectorSize :: (Type a, Type b) =>
  (Stream (Data a) -> Stream (Data b))
  -> (Data Length -> Data Length)
  -> (Vector (Data a) -> Vector (Data b))
streamAsVectorSize f s v =
  unfreezeVector $ take (s $ length v) $ f $ cycle v
```

Látható, hogy kétparaméteres függvények használata igen kényelmetlen, de ha belegondolunk, már az egyszerűbb példák esetén is nehézkes a használata.

Konstans Streamek fordíthatóvá tételéről a take függvénnyel gondoskodhatunk. Tegyük fel, hogy az előbb használt Fibonacci sorozatot előállító C kódot szeretnénk generálni, ekkor a következő kódot kell írunk:

```
take 4 $ unfold \(s1,s2)->(s1,(s2,s2+s1))
  (0::Data Int32,1::Data Int32))
```

A másik nagy hiányossága a könyvtárnak, hogy nem definiál konkatenációt. Ennek alapvető akadálya az, hogy a Stream végtelen. Ez jelentősen csökkenti a típus használhatóságát, hisz így két Stream konkatenáláshoz Vectorrá kell konvertálni a Streameket majd vissza Streammé, elvégezni a további Stream műveleteket, és az előbb kifejtett okok miatt megint Vecotr-rá. Mint láttuk így a konkatenáció előtti és utáni műveletekből külön ciklus generálódik a C kódba, nem is beszélve arról, hogy a sok konverzió mennyire memóriapazarló.

Tekintsük azt az egyszerű esetet, mikor egy Streammel implementált algoritmus kimenetének első és második felét fel akarjuk cserélni, majd ezen egy újabb Streammel implementált algoritmust alkalmazunk (például egy hashelés

részeként). Ekkor Vectorrá kell konvertálni, de így sajnos a ciklus összevonás nem valósulhat meg. Tekintsük a következő példát:

```
exampleForConcatStr :: V.DVector Int32 -> V.DVector Int32
exampleForConcatStr v =
  scanByStream (*) $ scanByStream (+) (V.drop len v)
  V.++ (scanByStream (+) (V.take len v))
  where
    scanByStream f v =
      (S.streamAsVectorSize (S.scan f 0) (\len->len+1) v
       len= (V.length v) 'div' 2
```

Az ebből generált C kód az alábbiakban olvasható. Vegyük észre, hogy a szorzást és az összeadást külön ciklusban végzi, ez két művelet esetén nem jelentős, de egy összetettebb esetben nyilvánvalóan hatékonyság szempontjából ez nem a legoptimálisabb megoldás. Míg ha csak Vector műveleteket használtunk volna ezek a ciklusok összevonódtak volna.

Az átláthatóság kedvéért csak a számunkra releváns kódrészleteket láthatjuk alább, a teljes generált kód a A.1.2 függelékben olvasható. Megfigyelhető, hogy az összeadás és a szorzás külön ciklusba került, pedig ez a két ciklus összevonható lenne.

```
...
setLength(&at(struct array,mem,0), (v3 + 1));
{
  uint32_t i12;
  for(i12 = 0; i12 < (v3 + 1); i12 += 1)
  {
    int32_t temp11_1;
    int32_t temp11_2_1;
    uint32_t temp11_2_2;

    temp11_1 = x10_2_1;
    temp11_2_1 = (x10_2_1 + at(int32_t,in0,(x10_2_2 + v6
    )));
    temp11_2_2 = ((x10_2_2 + 1) % v3);
    x10_2_1 = temp11_2_1;
    x10_2_2 = temp11_2_2;
    at(int32_t,at(struct array,mem,0),i12) = temp11_1;
  }
}
...
setLength(&at(struct array,mem,4), (v35 + 1));
{
  uint32_t i40;
  for(i40 = 0; i40 < (v35 + 1); i40 += 1)
  {
    int32_t temp39_1;
    int32_t temp39_2_1;
    uint32_t temp39_2_2;

    temp39_1 = x38_2_1;
    temp39_2_1 = (x38_2_1 + at(int32_t,in0,x38_2_2));
    temp39_2_2 = ((x38_2_2 + 1) % v35);
    x38_2_1 = temp39_2_1;
```

```

        x38_2_2 = temp39_2_2;
        at(int32_t, at(struct array, mem, 4), i40) = temp39_1;
    }
}
x41_2_1 = 0;
x41_2_2 = 0;
setLength(out1, (v2 + 1));
{
    uint32_t i43;
    for(i43 = 0; i43 < (v2 + 1); i43 += 1)
    {
        int32_t temp42_1;
        int32_t temp42_2_1;
        uint32_t temp42_2_2;
        int32_t w44;

        temp42_1 = x41_2_1;
        if((x41_2_2 < v17))
        {
            w44 = at(int32_t, at(struct array, mem, 3), x41_2_2)
                ;
        }
        else
        {
            w44 = at(int32_t, at(struct array, mem, 4), (x41_2_2
                - v17));
        }
        temp42_2_1 = (x41_2_1 * w44);
        temp42_2_2 = ((x41_2_2 + 1) % v2);
        x41_2_1 = temp42_2_1;
        x41_2_2 = temp42_2_2;
        at(int32_t, (* out1), i43) = temp42_1;
    }
}

```

3. fejezet

Seq

3.1. Alapötlet

Az előző fejezetben láthattuk, hogy a Feldspar jelenlegi megoldásainak igen sok hátránya van, de azt is megfigyelhettük, hogy a két bemutatott könyvtár hátrányai diszjunktak. Így joggal merül fel az igény egy olyan könyvtárra, mely rendelkezik mindkét könyvtár jó tulajdonságaival, azaz feloldja az előző fejezetben említett problémákat. Ezzel a céllal írtam meg a *Seq* könyvtárat. Ebben a fejezetben ezt a könyvtárat és az ezzel elért eredményeimet mutatom be.

A *Seq* típus alapötlete, hogy a *Vector* típusba csempésszük be az állapotot, és a rákövetkezési függvény szerinti értékszámítást úgy, hogy a *Vector* jó tulajdonságai megmaradjanak. Tehát a *Seq* is egy algebrai adattípus lesz. Hogy konkatenálható legyen, a többszegmenses adatszerkezetet kell használnunk, így két konstruktora lesz. Az egyik az üres *Seq*, ez paraméter nélküli. A másik konstruktornak kell tartalmaznia a szegmens hosszát, a kezdő állapotot, a rákövetkezési függvényt, és valamilyen formában a következő szegmenst, tehát ez egy 4 paraméteres konstruktor lesz.

Az elképzelés az, hogy egy *Seq* értékei úgy állíthatók elő, hogy az első szegmens kezdőértékére alkalmazom a rákövetkezési függvényt, ami itt is egy értéket és egy állapotot ad vissza. Az így kapott állapotot fel lehet használni a következő érték előállítására a rákövetkezési függvény újbóli alkalmazásával. De a *Stream*-mel ellentétben itt a szegmens véges, így csak annyi értéket állítunk elő, amennyi a szegmens hossza, utána elkezdjük a következő szegmens értékeinek előállítását, feldolgozását. Tehát véges adatfolyamot kell definiálni.

3.2. Megvalósítás

3.2.1. A *Seq* típus definíciója

A könyvtár implementálásához először a *Seq* típust kell definiálni. Elsőre két-paraméteres típusnak tűnik, hisz az állapotnak is van típusa. Első megközelítésben a típus definíciója a következő lehetne:

```
data Seq st a
    = EmptySeq
```

```

| Seq
  { segmentLength :: Data Length
  , segmentStartSt :: st
  , segmentValue   :: st -> (st,a)
  , continuation  :: Seq st a
  }

```

Ez a definíció megfelel az elképzelésnek, hogy hogyan is működik a Seq, de technikai akadályokba ütközünk. Alapvetően a típushoz szeretnénk instance-t készíteni, hogy tagja legyen a **Syntactic** osztálynak. Mint ahogy azt a Vectornál láttuk, ehhez szükséges, hogy le tudjuk tárolni Feldspar tömbbé, és Feldspar tömböt tudjuk Seq-ként kezelni. Erre szolgálnak a freezeSeq és az unfreezeSeq függvények. Sajnos azonban itt nem tudunk akármilyen típusú Feldspar tömböt Seq-é alakítani, hisz kell adnunk egy kezdőállapotot, amivel megkötjük annak típusát. Ezért egyparaméteres típust kell készíteni a Seqból. Sőt, hogy könnyebb legyen Feldspar tömböt Seq-é alakítani, a rákövetkezési függvényből indexről és állapotról, állapot és érték párjára képező függvényt kell készíteni. Ehhez általános algebrai adatszerkezetre van szükség, hisz így elrejtethető az állapot típusparamétere. Tehát, így a Seq típus egy jobb implementációja a következő:

```

data Seq a where
  EmptySeq :: Seq a
  Seq :: (Syntactic st) => Data Length -> st ->
    (Data Index -> st -> (st,a)) -> Seq a -> Seq a

```

Ez a definíció már nem ütközik technikai akadályokba, de a szegmensenkénti feldolgozása nem olyan egyszerű, mint a Vector esetében. Tekintsük ugyanis például a scan függvény működését. Nyilván úgy szeretnénk implementálni, hogy a részeredményeket az állapotban tároljuk, de ekkor valahogy meg kell oldani, hogy a szegmensek közt is át lehessen vinni ezt az értéket. Látható, hogy ezzel az implementációval ezt megoldani igen nehéz, ezért szükséges még egy módosítás, a Seq konstruktor utolsó paramétere nem Seq a, hanem állapotról Seq-re képező függvény. Ekkor az állapotot tovább tudjuk adni a következő szegmensnek, és így tovább tudjuk vinni a szükséges értékeket a következő szegmensbe. Ez első ránézésre úgy tűnhet, elbonyolíthatja a Seq használatát, de a továbbiakban látni fogjuk, hogy ez egyáltalán nincs így. Ezek alapján a következőképpen került implementálásra a típus:

```

data Seq a where
  EmptySeq :: Seq a
  Seq :: (Syntactic st) => Data Length -> st ->
    (Data Index -> st -> (st,a)) -> (st -> Seq a) -> Seq
    a

```

Mivel általában a véges adatfolyamot Seq (Data a) szerkezetű típusként akarjuk alkalmazni, készítünk egy típuszsinonímát, a DSeq-et, mely e használati módot egyszerűsíti. Az implementációja az alábbi.

```

type DSeq a = Seq (Data a)

```

3.2.2. A Seq könyvtár függvényeinek implementációja

A Seq könyvtár a Vectorhoz hasonlóan a funkcionális programozásban gyakran használt listafüggvényeket veszi alapul. Tehát a cél az, hogy a programozó a

lehető legkényelmesebben tudja a típust úgy használni, hogy közben nem gondol a mögöttes reprezentációra, hanem úgy tud vele dolgozni, mint egy átlagos funkcionális nyelvben a listákkal. Ez a cél teljesen sosem lesz elérhető a Feldspar sajátosságai miatt.

emptySeq. Mivel a `Seq` konstruktor utolsó paramétere függvény vagyis nem `Seq`, így kényelmi okokból definiáltam egy `emptySeq` nevű függvényt, mely egy egyparaméteres konstans üres `Seq`-re képező függvény. A Feldspar kódja a következő:

```
emptySeq = (\ _->EmptySeq)
```

seq'. Általában egyparaméteres `Seq`-eket akarunk definiálni, a többszegmensek különböző műveletek során jönnek létre. Ezért bevezetjük a `seq'` függvényt, mely a `Seq` konstruktorral megegyezik, csak nem tartalmaz következő szegmenst, vagyis az `emptySeq`. Általános konvenció, hogy az egyszegmenses műveletek vesszősek, ez a konvenció a `Vector` könyvtárban is megfigyelhető. Definíciója:

```
seq' :: (Syntactic st) =>
  Data Length -> st -> (Data Index -> st -> (st,a)) ->
    Seq a
seq' l st tr = Seq l st tr emptySeq
```

seq. Mivel nem elegáns közvetlenül a konstruktorokat használni, definiálok a `seq` függvényt is, mely teljes mértékben megegyezik a `Seq` konstruktorral.

```
seq :: (Syntactic st) =>
  Data Length
-> st
-> (Data Index -> st -> (st,a))
->(st -> Seq a)
-> Seq a
seq = Seq
```

empty. A `Seq` konstruktorhoz hasonlóan, az `EmptySeq` konstruktorhoz is készítettem egy olyan függvényt, az `empty`-t, mely teljesen megegyezik a konstruktorral.

```
empty = EmptySeq
```

length. Az egyik legalapvetőbb függvény konténer típusok esetén, azok hosszának lekérdezése. Ezért a `Seq` esetén is definiálok erre a célra a `length` függvényt. Ez alapvetően egy rekurzív függvény, mely összeadja az egyes szegmensek hosszát, melyeket mintaillesztéssel kapunk meg. Itt jegyezném meg, hogy a Feldspar nem támogatja a rekurzív függvényeket, amennyiben az azok által feldolgozandó elemek száma fordítási időben nem áll rendelkezésre. Szerencsére a szegmensek számát ismerjük fordítási időben is.

```
length :: Seq a -> Data Length
length EmptySeq = 0
length (Seq l st _ con) = 1 + length (con st)
```

freezeSeq. Mint, ahogy azt a Vectornál is említettem, hogy egy típust a Syntactic osztály tagjává tudjuk tenni, szükségünk van a freeze függvényre, mely az elemeket képes letárolni, azaz a Seq-et Feldspar tömbbé konvertálni. Erre szolgál a freezeSeq függvény. Ez egymás után kiszámítja az értékeket, miközben letárolja azokat. Erre létezik magnyelv konstrukció, mégpedig a sequential, ami rekurzív függvényeket számoló ciklusok leírását teszi lehetővé. Itt csak arra kell figyelni, hogy a függvény érték és állapot párját várja, míg a Seq-et úgy definiáltam, hogy a rákövetkezési függvény állapot és érték párját adja vissza. Ennek kezelésére jól használható a swap függvény. Rekurzióval gondoskodom arról, hogy az összes szegmens feldolgozásra kerüljön.

```
freezeSeq :: (Type a) => Seq (Data a) -> Data [a]
freezeSeq EmptySeq = value []
freezeSeq (Seq l ini fun con) =
    sequential l ini (\i st -> swap $ fun i st)
    (\s-> freezeSeq (con s))
```

unfreezeSeq. A másik függvény, amire szükségünk van az instance-ok elkészítéséhez, az az unfreeze, ami jelen esetben ez a unfreezeSeq nevet viseli, mely egy fizikai tömböt Seq-ként kezelhetővé tesz, azaz Feldspar tömböt Seq-é konvertál. Ez a függvény definiál egy Seq-et, melynek a rákövetkezési függvénye tetszőleges állapothoz és indexhez, az adott indexű tömbelemet rendeli értékül, és unit állapotot, a kezdőértéke szintén unit. A unit egy olyan típus, ami nincs hatással a generált C kódra, csak technikai jellegű információ, ez esetben üres állapotot jelent. Ezt Haskellben a () szimbólummal jelöljük.

```
unfreezeSeq :: (Type a) => Data [a] -> Seq (Data a)
unfreezeSeq arr =
    Seq (getLength arr) (value ())
    (\i _->(value (), getIx arr i)) emptySeq
```

Konkatenáció. Ez egy igen fontos függvény amit, mint ahogy láttuk a Stream könyvtár nem valósít meg. Viszont a Seq típus esetén triviális a megvalósítás módja, hisz a típus definíciójánál is törekedtünk arra, hogy ez a művelet könnyen megvalósítható legyen. Csak annyit kell tenni, hogy az utolsó szegmenst (ami üres Seq-re képző konstans függvény), lecseréljük a hozzákonkatenálandó Seq-re képző konstans függvényre. Ehhez rekurzióval végig kell mennünk a Seq szegmensein, hogy eljussunk az utolsóig.

```
(++) :: Seq a -> Seq a -> Seq a
EmptySeq      ++ sq          = sq
sq            ++ EmptySeq    = sq
Seq l s tr con ++ sq          = Seq l s tr (\s-> con s
    ++ sq)
```

take. A take függvény úgy működik, ahogy azt a neve alapján elvárjuk. Paraméterben kapott mennyiségű értéket hagy meg a Seq-ből, a többit eldobja. Az implementációja a következő:

```
take :: (Syntactic a,
        Role a ~ ()),
```

```

Info a ~ EdgeSize () (Internal a))
=> Data Length
-> Seq a
-> Seq a
take _ EmptySeq = EmptySeq
take n (Seq l st tr con) = condition (n==0) (empty)
  (condition (n<=1) (sq emptySeq) (sq cont))
  where
    sq = seq nHead st tr
    nHead = condition (n<1) n l
    cont = (\s->(take (n-1) (con s)))

```

Látható, hogy egy elágazással van definiálva, erre azért van szükség, mert ha a `take` kisebb-egyenlő értéket kap, mint a szegmens hossza, akkor az új szegmens hossza a kapott érték, a többi szegmenst elhagyjuk. Ha nagyobb, akkor a szegmenst változatlanul hagyjuk, és rekurzívan meghívjuk a `take`-t a következő szegmensére. Nyilván az első szegmens hosszával csökkentve adjuk át az első paraméter értékét.

Azt is megfigyelhetjük, hogy a 0 paraméter külön van kezelve, amire azért van szükség, hogy ez esetben `EmptySeq`-et kapjunk és ne 0 hosszú szegmenst, mert annak nem definiált a viselkedése.

A függvény műveletigénye $O(n)$, ahol n az első paraméter értéke, és a `Seq` hosszának minimuma, hisz a C-kód csak tömbökkel dolgozik, és az elemeket át kell másolni a kisebb tömbbe. Ha megfelelő kompozícióban használjuk, nem feltétlen jelenik meg közvetlenül a C kódban. Azaz ha elhagyunk vele szegmenseket, akkor az azok által tárolt értékeket már nem kell kiszámítani, így a másolás összevonható a kiértékeléssel (fusion). Viszont ez a műveletigény megegyezik a `take` függvény listára, Vectorra és Streamre megírt változatainak műveletigényével. Tehát a generált C kód:

```

void test(struct array mem,
          uint32_t in0,
          struct array in1,
          struct array * out2)
{
    uint32_t v4;
    uint32_t x9_2;

    v4 = length(in1);
    if((in0 == 0))
    {
        setLength(&at(struct array,mem,0), 0);
    }
    else
    {
        uint32_t x6_2;
        uint32_t w5;

        x6_2 = 0;
        if((in0 < v4))
        {
            w5 = in0;
        }
        else

```

```

{
    w5 = v4;
}
setLength(&at(struct array,mem,0), w5);
{
    uint32_t i8;
    for(i8 = 0; i8 < w5; i8 += 1)
    {
        int32_t temp7_1;
        uint32_t temp7_2;

        temp7_1 = at(int32_t,in1,i8);
        temp7_2 = x6_2;
        x6_2 = temp7_2;
        at(int32_t,at(struct array,mem,0),i8) =
            temp7_1;
    }
}
x9_2 = 0;
setLength(out2, length(at(struct array,mem,0)));
{
    uint32_t i11;
    for(i11 = 0; i11 < length(at(struct array,mem,0));
        i11 += 1)
    {
        int32_t temp10_1;
        uint32_t temp10_2;

        temp10_1 = at(int32_t,at(struct array,mem,0),i11
            );
        temp10_2 = 0;
        x9_2 = temp10_2;
        at(int32_t,(* out2),i11) = temp10_1;
    }
}
}

```

drop. Ez a függvény is az elvárásoknak megfelelően működik, a paraméterben kapott mennyiségű értéket levágja a Seq elejéről. Definiál egy Seq-et, melynek hossza a kívánt értékkel kevesebb, és kezdőállapota az eredeti Seq megfelelő állapota. Ehhez ki kell számítani az $(m - 1)$ -edik értéket, ahol m a levágandó értékek száma. És a felesleges szegmenseket el kell hagyni, ami rekurzívan történik, de ezeken a szegmenseken is végig kell számolni az állapotokat, hisz elképzelhető, hogy azt használja a következő szegmens, és ez esetben szükséges az új állapot kiszámításához az elhagyni kívánt szegmensek végállapota. Tehát az implementációja:

```

drop :: (Syntactic a,
        Role a ~ (),
        Info a ~ EdgeSize () (Internal a))
=> Data Length
-> Seq a

```

```

-> Seq a
drop _ EmptySeq = EmptySeq
drop n sq@(Seq l st tr con) = condition (n<1)
  (newSeq) (cont (forLoop l st (\i s-> fst(tr i s))))
  where
    newSeq = seq (1-n)
      (forLoop n st (\i s-> fst(tr i s))) (\i s->tr (i+n) s)
    con
    cont = (\s->(drop (n-1) (con s)))

```

A műveletigény tekintetében egyrészt ki kell számítani az új kezdőállapotot, illetve a többi elemet át kell másolni a kisebb tömbbe, így a műveletigénye $O(m + (n - m)) = O(n)$, ahol n a Seq hossza, m továbbra is a levágandó értékek száma és n minimuma. Ezt láthatjuk a következő generált C kódban is, de sajnós az is látható, hogy az eredményt egy temporális változóba írja és azt másolja az eredmény tömbbe, természetesen aszimptotikusan vizsgálva ez nem befolyásolja a műveletigényt:

```

void test(struct array mem,
          uint32_t in0,
          struct array in1,
          struct array * out2)
{
    uint32_t v4;
    uint32_t x10_2;

    v4 = length(in1);
    if((in0 < v4))
    {
        uint32_t x5_2;
        uint32_t temp8;

        x5_2 = 0;
        {
            uint32_t i9;
            for(i9 = 0; i9 < in0; i9 += 1)
            {
                temp8 = x5_2;
                x5_2 = temp8;
            }
        }
        setLength(&at(struct array,mem,0), (v4 - in0));
        {
            uint32_t i7;
            for(i7 = 0; i7 < (v4 - in0); i7 += 1)
            {
                int32_t temp6_1;
                uint32_t temp6_2;

                temp6_1 = at(int32_t,in1,(i7 + in0));
                temp6_2 = x5_2;
                x5_2 = temp6_2;
                at(int32_t,at(struct array,mem,0),i7) =
                    temp6_1;
            }
        }
    }
}

```

```

    }
}
else
{
    setLength(&at(struct array, mem, 0), 0);
}
x10_2 = 0;
setLength(out2, length(at(struct array, mem, 0)));
{
    uint32_t i12;
    for (i12 = 0; i12 < length(at(struct array, mem, 0));
        i12 += 1)
    {
        int32_t temp11_1;
        uint32_t temp11_2;

        temp11_1 = at(int32_t, at(struct array, mem, 0), i12);
        temp11_2 = 0;
        x10_2 = temp11_2;
        at(int32_t, (* out2), i12) = temp11_1;
    }
}
}

```

Összehasonlítva más implementációkkal, láthatjuk, hogy a műveletigénye megegyezik a Streames implementációval, de a Vectorokra implementált `drop` $O(n - m)$ es, míg a listákra akár $O(m)$ es implementáció is adható, ahol m és n továbbra is az elhagyandó értékek száma és a vektor hossza. Azaz a hatékonysága kissé elmarad a várttól.

Véletlen elérés. A `Seq` is tagja a `RandomAcces` osztálynak, azaz implementáljuk rá a (!) függvényt. Ahhoz, hogy egy elérjük egy elemét, az előző elemeket is mind elő kell állítanunk, csakúgy, mint a `drop` esetében, azzal, hogy most minket csak egy elem érdekel. Így az ott látott módon végigszámítjuk az elemeket, azonban itt nem az állapotot használjuk fel, hanem az értéket. Az implementációban így felhasználom a `drop` és `head` függvényeket.

```

instance (Syntactic a, Role a ~ ()),
    Info a ~ EdgeSize () (Internal a)
    => RandomAccess (Seq a)
where
    type Element (Seq a) = a
    sq!ind = head (drop ind sq)

```

Műveletigény tekintetében $O(n)$, hisz végig ki kell számolni az adott indexű elemig az összes őt megelőzőt. Legrosszabb esetben az utolsót kérdezzük le, tehát a műveletigény a lekérdezett elem indexétől függ. Legyen ez a szám i , akkor a műveletigény $\Theta(i)$. Csakúgy, mint a Streamekre implementált változatánál, vagy a listák véletlen elérése esetén, ez utóbbi esetben is végig kell iterálni az összes a kívánt elem előtti elemén. A Vector könyvtár ennél jobb implementációt ad, hiszen ez ott konstans idejű. De az adatfolyamokra elvi okok miatt nem is adható konstans idejű.

last. A `last` függvény a `Seq` utolsó elemét adja vissza. Ez könnyen definiálható, hisz a `Seq` hossza mínusz egyedik indexű elemet kell lekérdezni. A Feldspar kódja a következő:

```
last :: (Syntactic a,
        Role a ~ (),
        Info a ~ EdgeSize () (Internal a))
      => Seq a -> a
last sq = sq ! (length sq - 1)
```

A műveletigénye ennek megfelelően $O(n)$, ahogy azt a véletlen elérésnél ki is fejtettem.

head. A `head` függvény visszaadja a `Seq` első elemét. Ehhez egyszerűen alkalmazni kell egyszer a rákövetkezési függvényt a kezdőállapotra. Így ennek műveletigénye konstans, mint ahogy az a Streamekre, Vectorokra és a Haskell lista függvényeire implementált `head` esetén is megállapítható. Feldspar kódja a következő:

```
head :: Seq a -> a
head (Seq l st tr _) = snd (tr 0 st)
```

tail. A `tail` függvény elhagyja a `Seq` első elemét. Ezt implementálhatnánk `drop 1`ként is, de nem tesszük, ezzel megspórolva a C kódba generálódó felesleges elágazást többszegmenses `Seq` esetén. Ehelyett két esetet veszünk: az első, amikor az első szegmens 1 hosszú, ekkor ezt el kell hagyni. A másik eset az, amikor nem 1 hosszú, ekkor a hosszát csökkenteni kell eggyel, a kezdőállapotot pedig úgy kaphatjuk, ha alkalmazzuk egyszer a rákövetkezési függvényt. Tehát a Feldspar kódja a következő:

```
tail :: Seq a -> Seq a
tail (Seq l st tr con) = con (fst (tr 0 st))
tail (Seq l st tr con) =
    seq (l-1) (fst (tr 0 st)) (\i s-> tr (i+1) s) con
```

A műveletigénye $O(n)$, hisz ez esetben is át kell másolni a megmaradó elemeket egy kisebb tömbbe.

init. Az `init` függvény elhagyja a `Seq` utolsó elemét, ami a `take` segítségével könnyen lekódolható. A műveletigénye is megegyezik a `take` műveletigényével. A Feldspar kódja a következő:

```
init :: (Syntactic a,
        Role a ~ (),
        Info a ~ EdgeSize () (Internal a)) => Seq a -> Seq
      a
init sq = take (length sq - 1) sq
```

tails. Egy `Seq` összes posztfixét megkaphatjuk a `tails` függvény segítségével, tehát ez a függvény `Seq`-ek `Seq`-jét állítja elő.

```
tails :: (Syntactic a,
         Role a ~ (),
```

```

      Info a ~ EdgeSize () (Internal a))=> Seq a -> Seq
      (Seq a)
tails sq@(Seq _ st _ _) =
  seq' ((length sq) + 1) (st) (\i s->(s, (drop i sq)))

```

inits. Egy Seq összes prefixét megkaphatjuk az `inits` függvény segítségével. Tehát ez a függvény Seq-ek Seq-jét állítja elő. Így a műveletigénye $O(n^2)$, hisz ennek tömbbé konvertálása csak két egymásba ágyazott ciklussal oldható meg.

```

inits :: (Syntactic a,
         Role a ~ (),
         Info a ~ EdgeSize () (Internal a))
      => Seq a
      -> Seq (Seq a)
inits sq@(Seq _ st _ _) =
  seq' ((length sq) + 1) (st) (\i s->(s, (take i sq)))

```

inits1. Az `inits1` abban különbözik az `inits`től, hogy csak a valódi prefixeket állítja elő, azaz az üres Seq-et nem. Tehát csak el kell hagyni az `inits` által előállított Seq első elemét. Tehát a kódja

```

inits1 :: (Syntactic a,
          Role a ~ (),
          Info a ~ EdgeSize () (Internal a))
       => Seq a
       -> Seq (Seq a)
inits1 = tail . inits

```

Nyilván az `inits` művelet igényére leírtak az `inits1`-re is igazak.

map. A `map` függvény a paraméterében kapott függvényt alkalmazza a Seq minden elemére. Ezt legegyszerűbb úgy implementálni, hogy az új rákövetkezési függvény a régi által adott értékre alkalmazza a kívánt függvényt. Tehát egy új Seq-et definiál, aminek a letárolási költsége a C kódban eddig látott esetekhez hasonlóan $O(n)$. Ez megegyezik a Vectorra és a Streamre implementált `map` műveletigényével. A Seq könyvtár pedig az alábbi implementációt tartalmazza:

```

map :: (a->b) -> Seq a -> Seq b
map _ EmptySeq = EmptySeq
map f (Seq l st tr con) =
  seq l st (\i s -> (fst (tr i s), f $ snd $ tr i s))
  $ (\s->map f (con s))

```

fold. A `fold` függvény a paraméterben kapott művelettel és kezdőértékkel összehajtogatja a Seq elemeit. Erre jól használható a `forLoop` magnyelvi függvény, amivel összegezzük az egyes szegmenseket, a szegmensek összegzése pedig itt is rekúzió segítségével valósul meg. Az előző szegmens részeredménye pedig kezdőértékként adódik tovább. A Feldspar kódja a következő:

```

fold :: Syntactic a => (a -> b -> a) -> a -> Seq b -> a
fold _ init EmptySeq = init
fold f init (Seq l st tr cont) = fold f (thisRe) (cont $
  conSt)

```

```

where
  tr' idx (seqState, foldState) = (s', f foldState a)
  where
    (s', a) = tr idx seqState
  (conSt, thisRe) = forLoop 1 (st, init) tr'

```

A műveletigénye lineáris, csakúgy mint a Vectorra implementált változat esetében. A Streamekre pedig nem lehetett implementálni, mivel ott végtelen sok elemet kellett volna összehajtogatni.

fold1. A `fold1` a `fold` egy olyan változata, mely kezdőértéknek a `Seq` első elemét használja. A definiálására a `fold` függvényt használok, mint ahogy az alábbiakban látható.

```

fold1 :: Syntactic a => (a -> a -> a) -> Seq a -> a
fold1 f s = fold f (head s) (tail s)

```

sum. A `sum` függvény egy `Seq` elemeit összegzi. Azaz a `fold` függvény olyan változata, ahol az első paraméter a `(+)` függvény, a második pedig a `0` konstans. Tehát az implementáció:

```

sum :: (Numeric a) => Seq (Data a) -> Data a
sum = fold (+) 0

```

A műveletigénye nyilván megegyezik a `fold` függvényével.

maximum. A `maximum` függvény kiválasztja egy `Seq` legnagyobb elemét (nyilván ez csak olyan `Seq`-ekre értelmes, melynek elemei összehasonlíthatók). Az implementációjához itt a `fold1`-et használok fel, az első paramétere a `max` függvény, mely két elem közül eldönti, melyik a nagyobb.

```

maximum :: (Ord a) => Seq (Data a) -> Data a
maximum = fold1 max

```

minimum. A `minimum` függvény kiválasztja egy `Seq` legkisebb elemét, az implementációja hasonló a `maximum`-hoz.

```

minimum :: (Ord a) => Seq (Data a) -> Data a
minimum = fold1 min

```

zipWith. A `zipWith` függvény paraméterben kap egy kétparaméteres függvényt, melyet páronként alkalmaz a paraméterben kapott `Seq` elemeire. Ezzel egy új `Seq`-et hoz létre.

A feladat első megközelítésben nem túl bonyolult. Definiáljuk az új `Seq`-et oly módon, hogy a két input `Seq` állapotainak párja az új állapot. A rákövetkezési függvényt is ennek megfelelően definiáljuk. A rákövetkezési függvény által visszaadandó értéket megkaphatjuk, ha a paraméterben kapott `Seq`-ek által adott eredményre alkalmazzuk a paraméterben kapott függvényt.

Első probléma, amibe beleütközünk, hogy milyen hosszú legyen az új `Seq`, hisz a két `Seq` hossza nem biztos, hogy egyenlő. Ennek a problémának egyszerű feloldása, ha mondjuk mindig a rövidebb `Seq` hosszával megegyezőre állítjuk az

új Seq hosszát, ahogy ez a Vector és a Haskell listái esetén is van. Ez viszont előrevetíti azt a problémát, hogy a Seq szegmenseinek hossza sem biztos, hogy megegyezik. Ha ezt a megoldást alkalmaznánk a szegmensek hosszára is, akkor elhagynánk elemeket, sőt, az sem biztos hogy a szegmensek száma megegyezik.

Összefoglalva, egy függvénnyel csak egy Seq szegmensein tudunk végig ite-
rálni, ha a másik nem pont ugyanilyen vagy ugyanennyi szegmensből áll, akkor
a rekurzió nem a kívánt szegmenseket hozza létre az új Seq-ben. Hogyan is
nézzen ki az új Seq szerkezete? Ha a két bemenet felépítése megegyezne, akkor
a probléma nem állna fenn.

A probléma megoldása tehát az, hogy azonos felépítésűre kell hozni a két
bemeneti Seq-et. Két lehetőség van, az egészet egy szegmensűvé tenni, vagy
ahol valamely bemenetben új szegmens kezdődik, ott az új Seqban is kezdőd-
jön új szegmens. Az első nem implementálható Seq-re, mivel a szegmensek
állapotai különböző típusúak is lehetnek, így nem lehetne meghatározni az új
Seq állapotának típusát. Így a másik megoldást választottam. Definiáltam egy
előfeldolgozó függvényt, a `cats`-t. Ez azt teszi, hogy az első paraméterében
kapott Seq szerint feldarabolja a második paraméterében kapott Seq-et, ehhez
a `cat` segédfüggvényt használja. A `Feldspar` kódja a következő:

```
cat 0 sq = sq
cat _ EmptySeq = EmptySeq
cat len (Seq l st tr con) = condition (l>len) (cated) (rec)
  where
    cated = seq len st tr (\s->seq (l-len) s tr' con)
    tr' i s = tr (i+len) s
    rec = seq l st tr (\s->cat (len-1) (con s))

cats EmptySeq sq = sq
cats (Seq l st _ con) sq = (cats (con st) (cat l sq))
```

A `zipWith` ezek után könnyen definiálható, a bemenetekre váltott szereposzás-
ban meg kell hívni a `cats`-t, így alkalmazható a fent leírt algoritmus. Az így
kapott eredményt még a megfelelő méretűre kell vágni, amit a `take`-kel könnyen
meg lehet tenni. Tehát az implementációja:

```
zipWith f sq1 sq2 =
  take (min (length sq1) (length sq2)) $ h (cats sq2 sq1) (
    cats sq1 sq2)
  where
    h EmptySeq _ =
      EmptySeq
    h _ EmptySeq =
      EmptySeq
    h (Seq l1 st1 tr1 con1) (Seq l2 st2 tr2 con2) =
      seq l1 (st1,st2) tr' (\(s1,s2)->h (con1 s1) (con2
        s2))
    where
      tr' i (s1,s2) = ((s1',s2'), a 'f' b)
    where
      (s1',a) = tr1 i s1
      (s2',b) = tr2 i s2
```

zip. A `zip` függvény két `Seq`-ből készít párok `Seq`-jét úgy, hogy az eredmény `Seq` elemeinek első komponensét az első, a második komponensét a második `Seq`-ből veszi. Ezt a `zipWith` segítségével könnyen megtehetjük, hisz a `pair` függvény két `Data` értékből előállítja azok `Data` párját, így ezt adva a `zipWith` első paraméternek, a kívánt működést kapjuk.

Tehát a `Feldspar` kódja a következő:

```
zip :: (Type a, Type b) => DSeq a -> DSeq b -> DSeq ((a,b))
zip sq1 sq2 = zipWith pair sq1 sq2
```

Mivel a `zipWith`-del definiáltam, a műveletigénye megegyezik annak műveletigényével.

fromVector. Mivel jelenleg a `Feldspar`-ban az algoritmusok többsége `Vector` segítségével van implementálva, fontos, hogy a `Seq` kompatibilis legyen a `Vector`-tal. Ezért definiáltam konverziós függvényeket. A `fromVector` segítségével tudunk `Vector`-ot `Seq`-é konvertálni. Ezt futásidejű költség nélkül tehetjük meg, hisz a `Vector` indexelő függvénye egy indexről képez egy értékre, a `Seq` rákövetkezési függvénye pedig egy indexről és egy állapotról képez egy állapot és egy érték párjára. Tehát megállapíthatjuk, hogy a `Vector` egy általánosítása a `Seq`, így a `Vector` és a `Seq` közti konverzióknak ez az iránya valóban lehet költség nélküli. A `Feldspar` kódja alább látható.

Egy olyan `Seq`-et kell létrehozni, ami nem foglalkozik az állapottal, vagyis az indexelő függvény az állapotot helyben hagyja, az értéket pedig a `Vector` indexelő függvényével számítja ki. A szegmensek száma és hossza megmarad. Az üres `Vector` az üres `Seq`-re képződik le. Tehát az implementáció:

```
fromVector :: Vec.Vector a -> Seq a
fromVector Vec.Empty = EmptySeq
fromVector (Vec.Indexed l t c) =
    (seq' l (value ()) (\i s->(s,t i))) ++ fromVector c
```

toVector. Mivel a `Seq` a `Vector` általánosítása, a `Seq` `Vector`-rá való konvertálása nem lehet költség nélküli. Az állapotokkal való számítás adta a motivációt a `Seq` bevezetéséhez, így ezt a plusz információt nem tudjuk átvinni a `Vector`-ra. Ezért a konverzióhoz le kell tárolni az adatokat. Tehát, míg egy `Vector`-ral implementált algoritmus lehet úgy bemenete egy `Seq`-kel implementált algoritmusnak, hogy ha van rá mód, a két algoritmus ciklusai összevonódjanak. Fordított esetben nincs rá mód, de ha csak olyan vektor műveleteket használunk, amik `Seq`-re is implementálva vannak, akkor érdemes kihagyni a konverziót. Az implementációja a következő:

```
toVector :: (Type a) => Seq (Data a) -> Vec.Vector (Data a)
toVector a = (Vec.unfreezeVector) (freezeSeq a)
```

scan. A könyvtár tartalmazza a `scan` függvény implementációját (definícióját lásd 1. definíció) `Seq`-re. Mint ahogy azt az előző fejezetben kifejtettem, ezt a függvényt nem lehet megvalósítani hatékonyan `Vector`-ra. A `Seq` egyik motivációja is az volt, hogy hatékonyan lehessen konkatenálni és `scan` függvényt implementálni. Tehát a `scan` függvényt úgy kell megvalósítani, – mint ahogy azt az előző fejezetben kifejtettem – hogy $O(n)$ műveletigényű legyen.

Ezt a Seq tulajdonságai miatt könnyen meg lehet tenni. A Seq állapotában el tudom tárolni a részletösszegeket. Tehát fel kell építeni egy új Seq-et, ami az állapotában részeredményeket tárol, majd ezek alapján számítja ki az elemeket.

A kezdőállapota a bemeneti Seq kezdőállapota, és a paraméterben kapott kezdőérték rendezett párja. Ezzel az állapot típusát is meghatároztuk. A következő lépés a rákövetkezési függvény feladatának meghatározása. Tudjuk, hogy a kimeneti Seq első eleme minden esetben a kezdőérték, tehát a rákövetkezési függvénynek ezt kell visszaadnia értékként, hogy teljesüljön az a feltétel, hogy az első elem megegyezik a paraméterben kapott kezdőértékkel. Ekkor már egyértelmű, hogy a rákövetkezési függvény, hogyan számítja ki az új állapotot: a paraméterben kapott függvényt alkalmazom a régi állapotra és a bemeneti Seq soron következő elemére.

Az eddigi részösszeg szegmensek közti átadására kihasználom, hogy a Seq konstruktor utolsó paramétere egy állapotról képez Seq-re. Az aktuális szegmens állapotából úgy készítjük el az új, következő szegmenst, hogy a részösszeget továbbadjuk a scan-nek kezdőértékként (mely előállítja a következő szegmenst is), az eredeti állapotot pedig a következő szegmensre képező függvénynek.

Hogyan válasszuk meg az új Seq hosszát? A kérdés könnyű, legyen a bemeneti Seq hossza n , ekkor a kimenetnek $n + 1$ hosszúnak kell lennie. Ekkor az állapotban kiszámítódna az $(n + 2)$. elem is, de ehhez szükség lenne a bemeneti Seq $(n + 1)$. elemére, ami nincs. Ezért n -re állítjuk a hosszát, így elvileg az utolsó értéket elveszítenénk. De a bemeneti Seq szegmensein végig kell rekurzálni, könnyen módosíthatjuk a rekurzió leállítását úgy, hogy a kimenet egyel több szegmensű lesz. Az utolsó szegmens egy hosszú, és az állapotot már nem módosítja, csak az utolsó értéket olvassa ki belőle. Így az új Seq összességében a kívánt $n + 1$ hosszúságú lesz.

```
scan :: (Syntactic a) => (a -> b -> a) -> a -> Seq b -> Seq a
scan _ ini EmptySeq = Seq (1) (0::Data Int32) (\_ s -> (s,
    ini)) emptySeq
scan f ini (Seq l1 ini1 f1 cont1) = Seq l1 (ini1,ini) tr'
    cont'
  where
    tr' i (s,s1) = ((z,b),s1)
    where
      (z,a) = f1 i s
      b = f s1 a
    cont' (s1,s) = scan f s $ cont1 s1
```

3.3. Az előző fejezetben felvetett problémák megoldása, mérési eredmények

A dolgozat célkitűzését sikerült elérni, miszerint létrehoztam egy olyan absztrakciót Feldspar magas szintű könyvtárként, mely a *kívánt hatékonysággal* definiálja a funkcionális nyelvekben megszokott, digitális jelfeldolgozáshoz szükséges lista függvényeket. Úgy, mint a scan-t,map-et,zip-et,fold-ot és a konkatenációt. Azaz megvalósíthatók vele rekurzívan definiálható sorozatot előállító függvények és a csak véges adatszerkezetre értelmezhető függvények is, és ez által a kettő kompozíciójával definiált függvények is.

3.3.1. A könyvtár teljessége

Az általam megvalósított könyvtár tartalmazza az eddigi megoldásokban, azaz a Streamben és a Vectorban megvalósított listákra szokásos függvények unióját Seq-re.

Mint ahogy azt láthattuk a 3.2.2 szakaszban, a scan megvalósítása $O(n)$ -es, és megegyezik a Stream-ekkel definiált scan aszimptotikus viselkedésével, ami jóval hatékonyabb, mint a Vectorral definiálható scan függvény, mely $O(n^2)$ -es. Ez általánosságban is elmondható a rekurzívan definiálható sorozatokról. Tegyük fel, hogy $n-k = c_k, n-k+1 = c_{k-1} \dots n_0 = c_0$ és $n_{i+1} = f(i+1, n_i, n_{i-1} \dots n_{i-k})$, ahol f tetszőleges, k konstans, i pedig tetszőleges természetes szám. Ekkor az n sorozat előállítható Seq segítségével úgy, hogy az állapotában tároljuk $n_i, n_{i-1} \dots n_{i-k}$ értékeket, a rákövetkezési függvény pedig az f .

Nyilván, ha az f felhasznál m db bemenő sorozatot, úgy, hogy az n_i meghatározásához a bemenő sorozatoknak maximum az i . tagjáig használ fel fix mennyiségű értéket, vagy az i db érték valamilyen aggregációját, akkor az állapotban tároljuk a bemenő Seq-ek állapotait is, így minden lépésben elő tudjuk állítani az f értéket konstans idő alatt. Tehát $O(n)$ hatékonyságú kódot kapunk, ha egy-egy lépés műveletigénye konstans. Láttuk, hogy ezek a típusú függvények kellően általánosak, gondoljunk csak az iir algoritmusra.

A konkatenáció, mint ahogy azt bemutattam, futásidejű költség nélkül hajtható végre, a vektorhoz hasonlóan. A Seq, véges adatszerkezetként, alkalmas csak véges adatszerkezetekre definiálható algoritmusok implementálására.

3.3.2. Felhasználóbarát interfész

A 2.2.1 fejezetben bemutattam többek közt, hogy Stream-ek, végtelenségük miatt, nehezen használhatók. Láttuk, hogy mivel a streamek végtelenek, így nem lehetnek a Syntactic példányai (közvetlenül nem fordíthatók), ezért streamAsVector függvényt kell használni, ami egy Streamről Streamre képező függvényt Vectorról Vectorra képezővé konvertál. Ha több, Streamről Streamre képező függvényt akarunk fordítható formára konvertálni, akkor ezt igen nehézkesen tudjuk megtenni, átláthatatlanná teheti a kódot. A Seq viszont példánya a Syntactic osztálynak, így a Seq-kel definiált függvények közvetlenül fordíthatók. Az alábbi egyszerű példák is jól illusztrálják, hogy a Seq mennyivel egyszerűbben használható. Konstans Stream, melyet Feldspar tömbbé kell konvertálni, ahhoz, hogy fordítható legyen:

```
take 4 $ unfold (\(s1,s2)->(s1,(s2,s2+s1)))
(0::Data Int32,1::Data Int32))
```

Közvetlenül fordítható konstans Seq definíciója:

```
seq' 4 (0::Data Int32,1::Data Int32 )
(\i (s1,s2)->((s2, s1+s2),s2))
```

A Seq könnyen alkalmazható olyan esetekben is, amikor az index is szükséges egy elem kiszámításához. Ha csak az indexen múlik, akkor azt kényelmes lehet vektorral kódolni, ha csak a megelőző értékektől, akkor stream is kényelmes lehet, de ha mindkettőt használni akarjuk, akkor a Seq sokkal kényelmesebb, mint a stream, mivel nem kell az állapotban karbantartani az aktuális indexet.

3.3.3. Fusion

A Seq alapgondolata egy olyan absztrakció létrehozása volt, mely ötvözi a stream és a vektor tulajdonságait. Erre azért volt szükség, hogy azokon az algoritmusokon is működjön a ciklusösszevonás, melyek eredetileg streammel és vektorral definiált függvények kompozíciójaként adhatók meg.

Eddig ez azért nem működhetett, mivel a Stream és a Vector közti konverzió úgy történik, hogy letárolja az elemeket, így az elemek manipulálása egy másik ciklusban folytatódik. A Seq esetén viszont nem szükséges konverzió, mivel ezek az algoritmusok felépíthetők hatékonyan felépíthetők tisztán Seq-re definiált függvények kompozíciójaként. Mivel a Seq-re hatékonyan megvalósíthatók azon függvények, melyek eddig vagy csak Vectorral, vagy csak Streammel voltak hatékonyak implementálhatók.

Elemi, csak Streammel implementált függvény a scan, csak vektorral megvalósítható alapvető függvény a konkatenáció. Ezért ezek kompozícióján jól látható ez a probléma. Alább látható ennek megvalósítása eddigi Feldspar eszközökkel és Seq-kel:

```
exampleForConcatStr :: V.DVector Int32 -> V.DVector Int32
exampleForConcatStr v =
  scanByStream (*) $ scanByStream (+) (V.drop 1 v)
  V.++ (scanByStream (+) (V.take 1 v))
  where
    scanByStream f v =
      (Stream.streamAsVectorSize (Stream.scan f 0)
       (\l->l+1) v)
    l = (V.length v) `div` 2
```

```
exampleForConcatSeq :: DSeq Int32 -> DSeq Int32
exampleForConcatSeq s =
  scan (*) 0 $ (scan (+) 0 (drop 1 s))
  ++ (scan (+) 0 (take 1 s))
  where
    l = (length s) `div` 2
```

Mindkét kód úgy működik, hogy megcseréli a bemenő sorozat első és második felét, majd mindkét felének veszi a részletösszegeinek sorozatát, végül az így kapott sorozat részsorzatainak sorozatát állítja elő. Vegyük észre, hogy a Seq-es kód átláthatóbb is. Az ezekből generált C kódok, a A.1.2 és a A.1.3 függelékben láthatók. Ezekből a következő részleteket érdemes megnézni:

Az első megoldásból generált kód részletei:

```
...
  setLength(&at(struct array,mem,0), (v3 + 1));
  {
    uint32_t i12;
    for(i12 = 0; i12 < (v3 + 1); i12 += 1)
```

```

    {
        int32_t temp11_1;
        int32_t temp11_2_1;
        uint32_t temp11_2_2;

        temp11_1 = x10_2_1;
        temp11_2_1 = (x10_2_1 + at(int32_t,in0,(x10_2_2
            + v6)));
        temp11_2_2 = ((x10_2_2 + 1) % v3);
        x10_2_1 = temp11_2_1;
        x10_2_2 = temp11_2_2;
        at(int32_t,at(struct array,mem,0),i12) =
            temp11_1;
    }
}
...
setLength(&at(struct array,mem,4), (v35 + 1));
{
    uint32_t i40;
    for(i40 = 0; i40 < (v35 + 1); i40 += 1)
    {
        int32_t temp39_1;
        int32_t temp39_2_1;
        uint32_t temp39_2_2;

        temp39_1 = x38_2_1;
        temp39_2_1 = (x38_2_1 + at(int32_t,in0,x38_2_2))
            ;
        temp39_2_2 = ((x38_2_2 + 1) % v35);
        x38_2_1 = temp39_2_1;
        x38_2_2 = temp39_2_2;
        at(int32_t,at(struct array,mem,4),i40) =
            temp39_1;
    }
}
x41_2_1 = 0;
x41_2_2 = 0;
setLength(out1, (v2 + 1));
{
    uint32_t i43;
    for(i43 = 0; i43 < (v2 + 1); i43 += 1)
    {
        int32_t temp42_1;
        int32_t temp42_2_1;
        uint32_t temp42_2_2;
        int32_t w44;

        temp42_1 = x41_2_1;
        if((x41_2_2 < v17))
        {
            w44 = at(int32_t,at(struct array,mem,3),
                x41_2_2);
        }
        else

```

```

    {
        w44 = at(int32_t, at(struct array, mem, 4), (
            x41_2_2 - v17));
    }
    temp42_2_1 = (x41_2_1 * w44);
    temp42_2_2 = ((x41_2_2 + 1) % v2);
    x41_2_1 = temp42_2_1;
    x41_2_2 = temp42_2_2;
    at(int32_t, (* out1), i43) = temp42_1;
}
}

```

A második megoldásból generált kód részletei:

```

...
{
    uint32_t i40;
    for(i40 = 0; i40 < v3; i40 += 1)
    {
        int32_t temp39_1;
        uint32_t temp39_2_1_1;
        int32_t temp39_2_1_2;
        int32_t temp39_2_2;
        int32_t v41;

        v41 = (x38_2_2 * (x38_2_1_2 + at(int32_t, at(
            struct array, mem, 2), i40)));
        temp39_1 = v41;
        temp39_2_1_1 = 0;
        temp39_2_1_2 = (x38_2_1_2 + at(int32_t, at(struct
            array, mem, 2), i40));
        temp39_2_2 = v41;
        x38_2_1_1 = temp39_2_1_1;
        x38_2_1_2 = temp39_2_1_2;
        x38_2_2 = temp39_2_2;
        at(int32_t, at(struct array, mem, 0), (i40 + 1)) =
            temp39_1;
    }
}
...
{
    uint32_t i47;
    for(i47 = 0; i47 < v20; i47 += 1)
    {
        int32_t temp46_1;
        uint32_t temp46_2_1_1;
        int32_t temp46_2_1_2;
        int32_t temp46_2_2;
        int32_t v48;

        v48 = (x45_2_2 * (x45_2_1_2 + at(int32_t, at(
            struct array, mem, 4), i47)));
        temp46_1 = v48;
    }
}

```

```

        temp46_2_1_1 = 0;
        temp46_2_1_2 = (x45_2_1_2 + at(int32_t, at(struct
            array, mem, 4), i47));
        temp46_2_2 = v48;
        x45_2_1_1 = temp46_2_1_1;
        x45_2_1_2 = temp46_2_1_2;
        x45_2_2 = temp46_2_2;
        at(int32_t, at(struct array, mem, 0), (((i47 + 1) +
            v3) + 1)) = temp46_1;
    }
}
x49_2_1_1 = 0;
x49_2_1_2 = 0;
x49_2_2 = 0;
setLength(out1, 1);
{
    uint32_t i51;
    for(i51 = 0; i51 < 1; i51 += 1)
    {
        int32_t temp50_1;
        uint32_t temp50_2_1_1;
        int32_t temp50_2_1_2;
        int32_t temp50_2_2;

        temp50_1 = 0;
        temp50_2_1_1 = 0;
        temp50_2_1_2 = 0;
        temp50_2_2 = 0;
        x49_2_1_1 = temp50_2_1_1;
        x49_2_1_2 = temp50_2_1_2;
        x49_2_2 = temp50_2_2;
        at(int32_t, (* out1), i51) = temp50_1;
    }
}
copyArrayPos(out1, 1, at(struct array, mem, 0));

```

A konkatenáció miatt több ciklus jön létre. Azt kell észrevenni, hogy a második (Seq-kel implementált) megoldás megfelelő ciklusaiban egy utasításban hajtódnak végre a szorzás és összeadás műveletek (itt az utolsó ciklus magja csak egyszer fut le, és nem tartalmaz számítást), míg az első megoldás esetén a szorzás egy külön ciklusban hajtódik végre (utolsó ciklus). Ez nyilván egy ilyen egyszerű példa esetén nem csökkenti jelentősen a hatékonyságot, viszont a felhasználók valószínűleg nem ilyen egyszerű példákra fogják használni a nyelvet, a memóriaigényt pedig már ez is növeli. Sejthető, hogy minél bonyolultabb algoritmusról van szó, ez a különbség annál jelentősebb. Ezt a sejtést a következő (3.3.4) alfejezetben mérési eredményekkel támasztom alá.

Az utóbbi példa arra is alkalmas, hogy megfigyeljük, a konkatenáció valóban futásidejű költség nélkül hajtódik-e végre, azaz a ciklusok mindig az eredmény megfelelő részét töltik-e fel.

3.3.4. Mérési eredmények

Ebben az alfejezetben mérési eredményekkel is alátámasztom fenti állításaimat. A futási időket két esetre mértem, a scan-re, melyről azt állítottam, hogy a $O(n)$ viselkedésű, és közel olyan hatékony, mint a Streammel megvalósított változata, illetve a konkatenáció és a scan kompozícióra, amiről azt mondtam, optimálisabban megvalósítható Seq-kel mint a jelenlegi megoldásokkal. A konkatenációról azt állítottam, hogy elméletben nincs futásidejű költsége, kivéve, ha önmagában használják, hisz ekkor a tömböket át kell másolni egy nagyobbba, aminek viszont van futásidejű költsége.

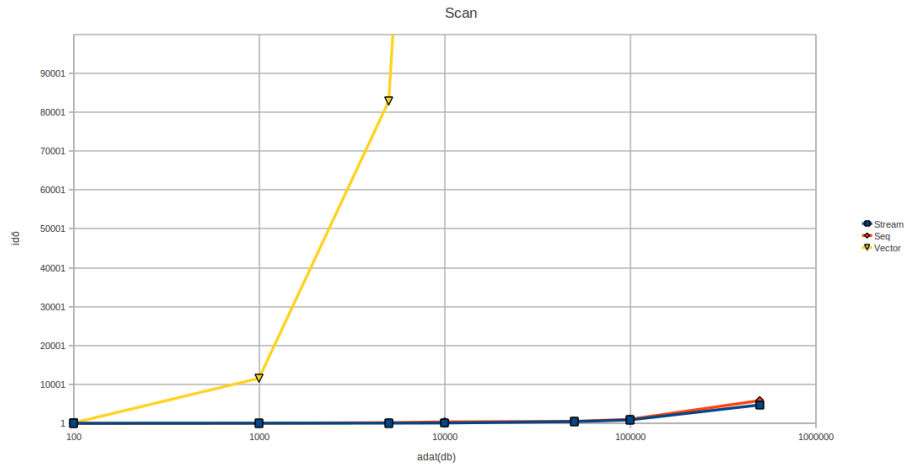
A méréseket egy Intel(R) Core(TM)2 Duo CPU T9600 2.8GHz-es kétmagos processzorú, 4 GB memóriával rendelkező gépen végeztem, melyen 10.4-es Ubuntu operációs rendszer fut. A C kódokat gcc 4.4.5 -ös fordítóval fordítottam (-std=c99 kapcsolóval) futtatható állománnyá.

Scan

Mint ahogy az alábbi (3.1-es) táblázatban látható, a Vectorral megvalósított scan valóban $O(n^2)$ -es, a konkrét mérési eredmények értékei a $0,003n^2$ függvénnyel közelíthetők. A Seq-kel és a Streammel megvalósított változata pedig lineáris. Látható, hogy a Seq-kel implementált változat kissé lassabb, a konkrét mérési eredményeket tekintve 1.2 konstans szorzóval becsülhető felülről a Streames változattal Seq-es.

3.1. táblázat. Mérési eredmények:scan

adat	Idő (mikroszekundum)		
	Vector	Stream	Seq
100	120	4	5
1000	11591	35	39
5000	82862	46	186
10000	311433	91	381
50000	7471161	445	525
100000		885	1046
500000		4734	5858



Kompozíció

A scan és konkatenáció kompozíciójának mérésére szintén azt az egyszerű példát vettem, melyet az előző (3.3.3) alfejezetben mutattam be. A 2.2.1 fejezetben kifejtettem, hogy ez akár egy hash algoritmus része is lehetne. Emlékeztetőül, a példa Feldspar kódja jelenlegi megoldásokkal és Seq használatával az alábbi:

```
exampleForConcatStr :: V.DVector Int32 -> V.DVector Int32
exampleForConcatStr v =
  scanByStream (*) $ scanByStream (+) (V.drop 1 v)
  V.++ (scanByStream (+) (V.take 1 v))
  where
    scanByStream f v =
      (Stream.streamAsVectorSize (Stream.scan f 0)
       (\l->l+1) v)
    l = (V.length v) `div` 2

exampleForConcatSeq :: DSeq Int32 -> DSeq Int32
exampleForConcatSeq s =
  scan (*) 0 $ (scan (+) 0 (drop 1 s))
  ++ (scan (+) 0 (take 1 s))
  where
    l = (length s) `div` 2
```

A mérési eredmények az alábbi, 3.2-es táblázatban láthatók. Megfigyelhető, hogy mindkét megoldás lineáris, ami nem meglepő. A mérési eredményekből látszik, hogy a *Seq*-kel implementált kompozíció 12% -al hatékonyabb, a 3.3.3 alfejezetben kifejtett okok miatt, miszerint a ciklusösszevonás az eddigi eszközökkel nem olyan hatékony, mint Seq-kel.

Azt állítottam a 3.3.3 alfejezetben, hogy minél összetettebb a kompozíció, annál hatékonyabb a Seq a jelenlegi megoldásokhoz képest. Ezt bizonyítandó az előző példa tovább bonyolított változatára is végeztem méréseket. Ezek Feldspar kódja a következő:

3.2. táblázat. Mérési eredmények:Kompozíció (hash)

adat	Idő (mikroszekundum)	
	Stream	Seq
100	16	14
1000	40	27
5000	154	181
10000	366	300
50000	1714	1522
100000	3909	3165
500000	18294	15777

```

exampleForConcatStr2 :: V.DVector Int32 -> V.DVector Int32
exampleForConcatStr2 s = scanByStream f
  (scanByStream (*) $ (scanByStream (+) (V.take 1 s))
   V.++ (scanByStream (+) (V.drop 1 s)))
  V.++ ((scanByStream (+) (V.drop 1 s))
   V.++ (scanByStream (+) (V.take 1 s)))
  where
    scanByStream f v =
      (Stream.streamAsVectorSize (Stream.scan f 0) (\l
        ->l+1) v)
    l = (V.length s) `div` 2
    f a b = (a*b)^3

exampleForConcatSeq2 :: DSeq Int32 -> DSeq Int32
exampleForConcatSeq2 s = scan f 10
  (scan (*) 0 $ (scan (+) 0 (take 1 s))
   ++ (scan (+) 0 (drop 1 s)))
  ++ ((scan (+) 0 (drop 1 s))
   ++ (scan (+) 0 (take 1 s)))
  where
    l = (length s) `div` 2
    f a b = (a*b)^3

```

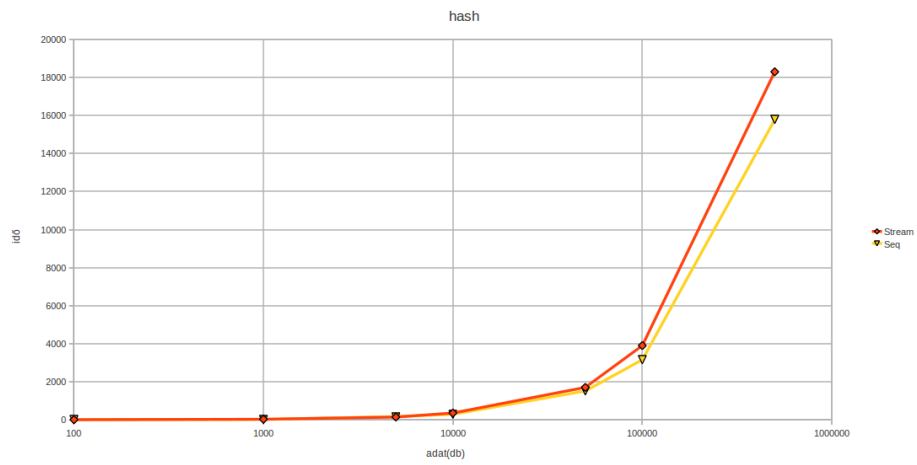
Az alábbi (3.3) táblázatban szereplő mérési eredményekből jól látható, hogy a Seq-kel implementált megoldás 180% -al hatékonyabb mint az eddigi eszközökkel adott megoldás. Azaz ezen a példán a Seq 2,8-szor hatékonyabb, mint az eddigi megoldások.

Összegzés. A scan és a konkatenáció elég általános algoritmus ahhoz, hogy a fenti példákból következtetni lehessen arra, hogy a Seq egyszerű példák esetén lehet, hogy nem olyan hatékony, mint az eddigi megoldások, de a gyakorlatban, ahol összetett algoritmusokat alkalmazunk, jóval hatékonyabb. Ezt láthatjuk a 3.1 és a 3.2 diagramokon is.

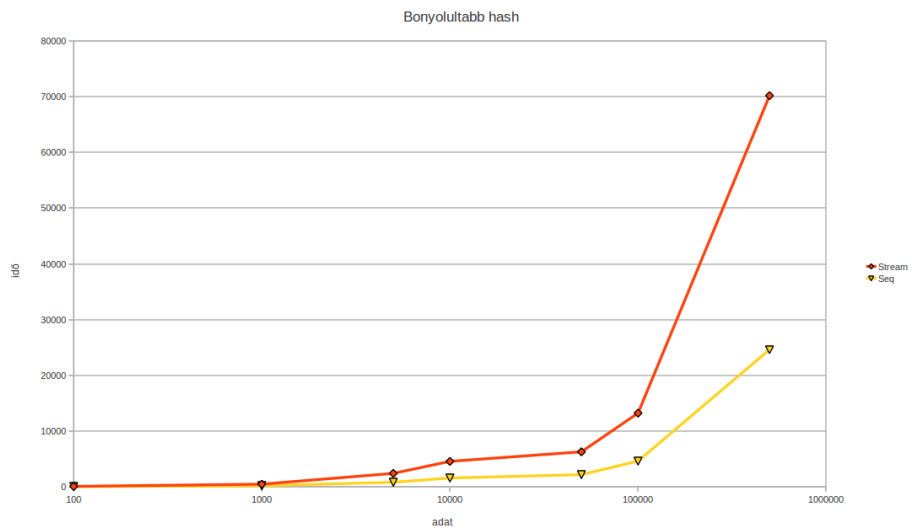
3.3. táblázat. Mérési eredmények: összetettebb kompozíció (hash)

adat	Idő (mikroszekundum)	
	Stream	Seq
100	55	21
1000	450	156
5000	2385	793
10000	4544	1558
50000	6257	2158
100000	13208	4594
500000	70212	24613

3.1. ábra.



3.2. ábra.



4. fejezet

Összefoglalás

A dolgozatomban bemutatam a Feldspar beágyazott, digitális jelfeldolgozásra specializált, funkcionális programozási nyelv jelenlegi magas szintű ciklus optimalizációs lehetőségeit. Részletesen bemutatam az ezekhez tartozó Vector és Stream könyvtárakat, illetve ezek hiányosságait és elméleti korlátjait. Ezt két alapvető algoritmus segítségével mutattam be, a scan-nel és a konkatenációval.

Munkám során implementáltam egy további lehetőséget biztosító könyvtárat a Seq-et, példákkal mutatom be az új megoldás előnyeit. A dolgozatban részletezem ezen könyvtár tulajdonságait, többek közt azt, hogy nem rendelkezik a Vector és a Stream elméleti korlátaival. A Seq hatékonyságát mérési eredményekkel támasztottam alá, amihez szintén a két alapvető algoritmust használtam, a scan-t és a konkatenációt.

4.1. Eredmények

A munkám során megterveztem és megvalósítottam egy új absztrakciós lehetőséget a Feldspar nyelvhez, amit egy magas szintű könyvtárként implementáltam. Ez a Seq nevet kapta. A könyvtárban megvalósítottam egy véges adatfolyamok leírására alkalmas típust, és a hozzá kapcsolódó függvényeket. Ez a könyvtár a következő előnyökkel rendelkezik a nyelv jelenlegi megoldásaihoz képest:

- A Seq egy könyvtáron belül implementálja az összes jelenlegi megoldások által megvalósított függvényt, míg az utóbbiak egyikével sem lehet az összeset implementálni.
- A Seq képes hatékonyan ($O(n)$ -es hatékonysággal) definiálni a *rekurzív sorozatokat definiáló függvényeket*, mint a scan-t, vagy a jelfeldolgozásban gyakran használt ir szűrő algoritmust. Ellentétben a Vector könyvtárral, ahol $\Omega(n^2)$ -es hatékonysággal tehattük ezt meg.
- A Seq véges adatszerkezet, így definiálható rá a konkatenáció is. A típus többszegmenses felépítése miatt a konkatenáció művelete általában futásidejű költség nélkül használható, ellentétben a Stream könyvtárral, mely végtelen adatszerkezetként nem alkalmas konkatenáció megvalósítására.

- A megvalósított könyvtár – az eddigi megoldásokkal ellentétben – képes egyszerre megvalósítani csak véges adatszerkezetekre értelmezett függvényeket és rekurzív sorozatot definiáló függvényeket is. Ez azt eredményezi, hogy ha ezeket vegyesen használó algoritmust kódolunk, nem szükséges konvertálni. Így a ciklusösszevonások is megtörténhetnek, hatékonyabb kódot eredményezve. A mérési eredmények szerint ez a hatékonyság javulás a kód összetettségével arányos. Azaz bonyolultabb kód esetén nagyobb különbség mutatkozik az eddigi megoldásokhoz képest.
- A Vector futásidejű költség nélkül konvertálható Seq-é. Ez azért lényeges, mert a Vectort használják a Feldsparban leggyakrabban azonos típusú elemek sorozatának tárolására.
- A könyvtár jól illeszkedik a Feldspar nyelvbe. A vele implementált algoritmusok konverzió nélkül fordíthatók, mivel a Seq példányosítja a `Syntactic` osztályt, míg a Stream könyvtár esetén nehézkesen használható konverziós függvényeket kell használnunk ahhoz, hogy fordítható kódot kapjunk, hisz a végtelensége miatt nem lehet a `Syntactic` példánya.

A. Függelék

Függelék

A.1. Generált kódok

A.1.1. Iir filter megvalósítása Vectorral

```
vIir :: V.DVector Float
-> V.DVector Float
-> V.DVector Float
-> V.DVector Float
vIir a b x@(V.Indexed l ixf cont) = res l
  where
    res l = V.indexed l ixf'
    ixf' j = snd $ forLoop (j+1) (initState) tr'
    where
      initState = (((V.freezeVector $ V.indexed (V.
        length a)
        (\i -> 0.0::Data Float)) , (0::Data Index)) ,
        ((V.freezeVector $
          V.indexed (V.length b) (\i -> 0.0::
            Data Float)) ,
            (0::Data Index)) ) , 0::Data Float)
      tr' i ((firstVec, secVec), _) = ( (firstVec',
        secVec') , newa)
    where
      (s', olda) = (s', ixf i)
      firstVec' = insert firstVec newa
      secVec' = insert secVec olda
      insert (s2, s3) val =
        (setIx s2 s3 val, (s3 + 1::Data Index) `mod`
          (getLength s2))
      newa =
        ((sumVec b secVec' secVec') - (sumVec a
          firstVec firstVec))
      sumVec v (s2, s3) vec = V.sum $ V.zipWith (*) v
        ( V.reverse $ V.rotateVecR
          ((getLength s2) - s3) $ V.unfreezeVector (
            fst vec))
      uf = V.unfreezeVector $ V.freezeVector x
```

```

void testIirV(struct array mem,
             struct array in0,
             struct array in1,
             struct array in2,
             struct array * out3)
{
    uint32_t v8;
    uint32_t v9;

    setLength(&at(struct array,mem,0), length(in0));
    {
        uint32_t i5;
        for(i5 = 0; i5 < length(in0); i5 += 1)
        {
            at(float,at(struct array,mem,0),i5) = 0.0f;
        }
    }
    setLength(&at(struct array,mem,1), length(in1));
    {
        uint32_t i7;
        for(i7 = 0; i7 < length(in1); i7 += 1)
        {
            at(float,at(struct array,mem,1),i7) = 0.0f;
        }
    }
    v8 = length(in1);
    v9 = length(in0);
    setLength(out3, length(in2));
    {
        uint32_t i10;
        for(i10 = 0; i10 < length(in2); i10 += 1)
        {
            uint32_t v11_1_1_2;
            uint32_t v11_1_2_2;
            float v11_2;
            uint32_t temp12_1_1_2;
            uint32_t temp12_1_2_2;
            float temp12_2;

            copyArray(&at(struct array,mem,2), at(struct array,
                mem,0));
            v11_1_1_2 = 0;
            copyArray(&at(struct array,mem,3), at(struct array,
                mem,1));
            v11_1_2_2 = 0;
            v11_2 = 0.0f;
            {
                uint32_t i13;
                for(i13 = 0; i13 < (i10 + 1); i13 += 1)
                {
                    float v14;
                    uint32_t v15;
                    uint32_t v17;
                    uint32_t v18;

```

```

uint32_t v19;
uint32_t v20;
uint32_t v21;
float w22;
float temp24;
uint32_t w23;
float w26;
float temp28;
uint32_t w27;

v15 = length(at(struct array,mem,3));
copyArray(&at(struct array,mem,6), at(struct
array,mem,3));
at(float,at(struct array,mem,6),v11_1_2_2) =
at(float,in2,i13);
v17 = (v15 - 1);
v18 = (v15 - ((v11_1_2_2 + 1) % v15));
v19 = length(at(struct array,mem,2));
v20 = (v19 - 1);
v21 = (v19 - v11_1_1_2);
if((v15 < v8))
{
    w23 = v15;
}
else
{
    w23 = v8;
}
w22 = 0.0f;
{
    uint32_t i25;
    for(i25 = 0; i25 < w23; i25 += 1)
    {
        temp24 = (w22 + (at(float,in1,i25) *
at(float,at(struct array,
mem,6)
,(v17 - (((v17 - (v17 - i25
)) + v18) % v15)))));
        w22 = temp24;
    }
}
if((v19 < v9))
{
    w27 = v19;
}
else
{
    w27 = v9;
}
w26 = 0.0f;
{
    uint32_t i29;
    for(i29 = 0; i29 < w27; i29 += 1)
    {

```

```

        temp28 = (w26 + (at(float,in0,i29) *
                        at(float,at(struct array,
                                mem,2),
                                (v20 - (((v20 - (v20 - i29)
                                            ) + v21) % v19))))));
        w26 = temp28;
    }
}
v14 = (w22 - w26);
copyArray(&at(struct array,mem,4), at(struct
    array,mem,2));
at(float,at(struct array,mem,4),v11_1_1_2) =
    v14;
temp12_1_1_2 = ((v11_1_1_2 + 1) % length(at(
    struct array,mem,2)));
copyArray(&at(struct array,mem,5), at(struct
    array,mem,3));
at(float,at(struct array,mem,5),v11_1_2_2) =
    at(float,in2,i13);
temp12_1_2_2 = ((v11_1_2_2 + 1) % length(at(
    struct array,mem,3)));
temp12_2 = v14;
copyArray(&at(struct array,mem,2), at(struct
    array,mem,4));
v11_1_1_2 = temp12_1_1_2;
copyArray(&at(struct array,mem,3), at(struct
    array,mem,5));
v11_1_2_2 = temp12_1_2_2;
v11_2 = temp12_2;
    }
}
    at(float,(* out3),i10) = v11_2;
}
}
}

```

A.1.2. Vector és Stream függvényeinek kompozíciója

```

exampleForConcatStr :: V.DVector Int32 -> V.DVector Int32
exampleForConcatStr v =
    scanByStream (*) $ scanByStream (+) (V.drop 1 v)
    V.++ (scanByStream (+) (V.take 1 v))
    where
        scanByStream f v =
            (Stream.streamAsVectorSize (Stream.scan f 0) (\l->l
                +1) v)
        l = (V.length v) `div` 2

```

```

void testhashStr(struct array mem,
                 struct array in0,
                 struct array * out1)
{
    uint32_t v2;
    uint32_t v3;
    uint32_t v4;
    uint32_t v5;
    uint32_t v6;
    uint32_t v7;
    uint32_t v8;
    int32_t x10_2_1;
    uint32_t x10_2_2;
    int32_t x14_2_1;
    uint32_t x14_2_2;
    uint32_t v17;
    uint32_t v18;
    uint32_t v19;
    uint32_t v20;
    uint32_t v21;
    int32_t x23_2_1;
    uint32_t x23_2_2;
    uint32_t v27;
    uint32_t v28;
    uint32_t v29;
    uint32_t v30;
    int32_t x31_2_1;
    uint32_t x31_2_2;
    uint32_t v35;
    uint32_t v36;
    uint32_t v37;
    int32_t x38_2_1;
    uint32_t x38_2_2;
    int32_t x41_2_1;
    uint32_t x41_2_2;

    v4 = (length(in0) >> 1);
    v5 = length(in0);
    if((v4 > v5))
    {
        v3 = 0;
    }
    else
    {
        v3 = (v5 - v4);
    }
    v6 = (length(in0) >> 1);
    v8 = length(in0);
    if((v6 < v8))
    {
        v7 = v6;
    }
    else
    {

```

```

        v7 = v8;
    }
    x10_2_1 = 0;
    x10_2_2 = 0;
    setLength(&at(struct array,mem,0), (v3 + 1));
    {
        uint32_t i12;
        for(i12 = 0; i12 < (v3 + 1); i12 += 1)
        {
            int32_t temp11_1;
            int32_t temp11_2_1;
            uint32_t temp11_2_2;

            temp11_1 = x10_2_1;
            temp11_2_1 = (x10_2_1 + at(int32_t,in0,(x10_2_2
                + v6)));
            temp11_2_2 = ((x10_2_2 + 1) % v3);
            x10_2_1 = temp11_2_1;
            x10_2_2 = temp11_2_2;
            at(int32_t,at(struct array,mem,0),i12) =
                temp11_1;
        }
    }
    x14_2_1 = 0;
    x14_2_2 = 0;
    setLength(&at(struct array,mem,1), (v7 + 1));
    {
        uint32_t i16;
        for(i16 = 0; i16 < (v7 + 1); i16 += 1)
        {
            int32_t temp15_1;
            int32_t temp15_2_1;
            uint32_t temp15_2_2;

            temp15_1 = x14_2_1;
            temp15_2_1 = (x14_2_1 + at(int32_t,in0,x14_2_2))
                ;
            temp15_2_2 = ((x14_2_2 + 1) % v7);
            x14_2_1 = temp15_2_1;
            x14_2_2 = temp15_2_2;
            at(int32_t,at(struct array,mem,1),i16) =
                temp15_1;
        }
    }
    v2 = (length(at(struct array,mem,0))
        + length(at(struct array,mem,1)));
    v19 = (length(in0) >> 1);
    v20 = length(in0);
    if((v19 > v20))
    {
        v18 = 0;
    }
    else
    {

```

```

        v18 = (v20 - v19);
    }
    v21 = (length(in0) >> 1);
    x23_2_1 = 0;
    x23_2_2 = 0;
    setLength(&at(struct array,mem,2), (v18 + 1));
    {
        uint32_t i25;
        for(i25 = 0; i25 < (v18 + 1); i25 += 1)
        {
            int32_t temp24_1;
            int32_t temp24_2_1;
            uint32_t temp24_2_2;

            temp24_1 = x23_2_1;
            temp24_2_1 = (x23_2_1 + at(int32_t,in0,(x23_2_2
                + v21)));
            temp24_2_2 = ((x23_2_2 + 1) % v18);
            x23_2_1 = temp24_2_1;
            x23_2_2 = temp24_2_2;
            at(int32_t,at(struct array,mem,2),i25) =
                temp24_1;
        }
    }
    v17 = length(at(struct array,mem,2));
    v28 = (length(in0) >> 1);
    v29 = length(in0);
    if((v28 > v29))
    {
        v27 = 0;
    }
    else
    {
        v27 = (v29 - v28);
    }
    v30 = (length(in0) >> 1);
    x31_2_1 = 0;
    x31_2_2 = 0;
    setLength(&at(struct array,mem,3), (v27 + 1));
    {
        uint32_t i33;
        for(i33 = 0; i33 < (v27 + 1); i33 += 1)
        {
            int32_t temp32_1;
            int32_t temp32_2_1;
            uint32_t temp32_2_2;

            temp32_1 = x31_2_1;
            temp32_2_1 = (x31_2_1 + at(int32_t,in0,(x31_2_2
                + v30)));
            temp32_2_2 = ((x31_2_2 + 1) % v27);
            x31_2_1 = temp32_2_1;
            x31_2_2 = temp32_2_2;
        }
    }

```

```

        at(int32_t,at(struct array,mem,3),i33) =
            temp32_1;
    }
}
v36 = (length(in0) >> 1);
v37 = length(in0);
if((v36 < v37))
{
    v35 = v36;
}
else
{
    v35 = v37;
}
x38_2_1 = 0;
x38_2_2 = 0;
setLength(&at(struct array,mem,4), (v35 + 1));
{
    uint32_t i40;
    for(i40 = 0; i40 < (v35 + 1); i40 += 1)
    {
        int32_t temp39_1;
        int32_t temp39_2_1;
        uint32_t temp39_2_2;

        temp39_1 = x38_2_1;
        temp39_2_1 = (x38_2_1 + at(int32_t,in0,x38_2_2))
            ;
        temp39_2_2 = ((x38_2_2 + 1) % v35);
        x38_2_1 = temp39_2_1;
        x38_2_2 = temp39_2_2;
        at(int32_t,at(struct array,mem,4),i40) =
            temp39_1;
    }
}
x41_2_1 = 0;
x41_2_2 = 0;
setLength(out1, (v2 + 1));
{
    uint32_t i43;
    for(i43 = 0; i43 < (v2 + 1); i43 += 1)
    {
        int32_t temp42_1;
        int32_t temp42_2_1;
        uint32_t temp42_2_2;
        int32_t w44;

        temp42_1 = x41_2_1;
        if((x41_2_2 < v17))
        {
            w44 = at(int32_t,at(struct array,mem,3),
                x41_2_2);
        }
        else

```

```

    {
        w44 = at(int32_t, at(struct array, mem, 4), (
            x41_2_2 - v17));
    }
    temp42_2_1 = (x41_2_1 * w44);
    temp42_2_2 = ((x41_2_2 + 1) % v2);
    x41_2_1 = temp42_2_1;
    x41_2_2 = temp42_2_2;
    at(int32_t, (* out1), i43) = temp42_1;
}
}
}

```

A.1.3. Seq függvényeinek kompozíciója

```

exampleForConcatSeq :: DSeq Int32 -> DSeq Int32
exampleForConcatSeq s =
    scan (*) 0 $ (scan (+) 0 (drop 1 s)) ++ (scan (+) 0 (take
        1 s))
    where
        l = (length s) `div` 2

```

```

void testhashS(struct array mem,
               struct array in0,
               struct array * out1)
{
    uint32_t v3;
    uint32_t v4;
    uint32_t v5;
    uint32_t v13;
    uint32_t v14;
    uint32_t v20;
    uint32_t v21;
    uint32_t v22;
    uint32_t v29;
    uint32_t v30;
    uint32_t x35_2_1_1;
    int32_t x35_2_1_2;
    int32_t x35_2_2;
    uint32_t x38_2_1_1;
    int32_t x38_2_1_2;
    int32_t x38_2_2;
    uint32_t x42_2_1_1;
    int32_t x42_2_1_2;
    int32_t x42_2_2;
    uint32_t x45_2_1_1;
    int32_t x45_2_1_2;
    int32_t x45_2_2;
}

```

```

uint32_t x49_2_1_1;
int32_t x49_2_1_2;
int32_t x49_2_2;

v4 = (length(in0) >> 1);
v5 = length(in0);
if((v4 < v5))
{
    uint32_t x7_2;
    uint32_t temp10;

    x7_2 = 0;
    {
        uint32_t i11;
        for(i11 = 0; i11 < v4; i11 += 1)
        {
            temp10 = 0;
            x7_2 = temp10;
        }
    }
    setLength(&at(struct array,mem,1), (v5 - v4));
    {
        uint32_t i9;
        for(i9 = 0; i9 < (v5 - v4); i9 += 1)
        {
            int32_t temp8_1;
            uint32_t temp8_2;

            temp8_1 = at(int32_t,in0,(i9 + v4));
            temp8_2 = 0;
            x7_2 = temp8_2;
            at(int32_t,at(struct array,mem,1),i9) =
                temp8_1;
        }
    }
}
else
{
    setLength(&at(struct array,mem,1), 0);
}
v3 = length(at(struct array,mem,1));
v13 = (length(in0) >> 1);
v14 = length(in0);
if((v13 < v14))
{
    uint32_t x15_2;
    uint32_t temp18;

    x15_2 = 0;
    {
        uint32_t i19;
        for(i19 = 0; i19 < v13; i19 += 1)
        {
            temp18 = 0;

```

```

        x15_2 = temp18;
    }
}
setLength(&at(struct array,mem,2), (v14 - v13));
{
    uint32_t i17;
    for(i17 = 0; i17 < (v14 - v13); i17 += 1)
    {
        int32_t temp16_1;
        uint32_t temp16_2;

        temp16_1 = at(int32_t,in0,(i17 + v13));
        temp16_2 = 0;
        x15_2 = temp16_2;
        at(int32_t,at(struct array,mem,2),i17) =
            temp16_1;
    }
}
else
{
    setLength(&at(struct array,mem,2), 0);
}
v21 = (length(in0) >> 1);
v22 = length(in0);
if((v21 == 0))
{
    setLength(&at(struct array,mem,3), 0);
}
else
{
    uint32_t x25_2;
    uint32_t w24;

    x25_2 = 0;
    if((v21 < v22))
    {
        w24 = v21;
    }
    else
    {
        w24 = v22;
    }
    setLength(&at(struct array,mem,3), w24);
    {
        uint32_t i27;
        for(i27 = 0; i27 < w24; i27 += 1)
        {
            int32_t temp26_1;
            uint32_t temp26_2;

            temp26_1 = at(int32_t,in0,i27);
            temp26_2 = 0;
            x25_2 = temp26_2;
        }
    }
}

```

```

        at(int32_t,at(struct array,mem,3),i27) =
            temp26_1;
    }
}
v20 = length(at(struct array,mem,3));
v29 = (length(in0) >> 1);
v30 = length(in0);
if((v29 == 0))
{
    setLength(&at(struct array,mem,4), 0);
}
else
{
    uint32_t x32_2;
    uint32_t w31;

    x32_2 = 0;
    if((v29 < v30))
    {
        w31 = v29;
    }
    else
    {
        w31 = v30;
    }
    setLength(&at(struct array,mem,4), w31);
    {
        uint32_t i34;
        for(i34 = 0; i34 < w31; i34 += 1)
        {
            int32_t temp33_1;
            uint32_t temp33_2;

            temp33_1 = at(int32_t,in0,i34);
            temp33_2 = 0;
            x32_2 = temp33_2;
            at(int32_t,at(struct array,mem,4),i34) =
                temp33_1;
        }
    }
}
x35_2_1_1 = 0;
x35_2_1_2 = 0;
x35_2_2 = 0;
setLength(&at(struct array,mem,0), 1);
{
    uint32_t i37;
    for(i37 = 0; i37 < 1; i37 += 1)
    {
        int32_t temp36_1;
        uint32_t temp36_2_1_1;
        int32_t temp36_2_1_2;
        int32_t temp36_2_2;
    }
}

```

```

        temp36_1 = 0;
        temp36_2_1_1 = 0;
        temp36_2_1_2 = 0;
        temp36_2_2 = 0;
        x35_2_1_1 = temp36_2_1_1;
        x35_2_1_2 = temp36_2_1_2;
        x35_2_2 = temp36_2_2;
        at(int32_t,at(struct array,mem,0),i37) =
            temp36_1;
    }
}
x38_2_1_1 = 0;
x38_2_1_2 = 0;
x38_2_2 = x35_2_2;
increaseLength(&at(struct array,mem,0), v3);
{
    uint32_t i40;
    for(i40 = 0; i40 < v3; i40 += 1)
    {
        int32_t temp39_1;
        uint32_t temp39_2_1_1;
        int32_t temp39_2_1_2;
        int32_t temp39_2_2;
        int32_t v41;

        v41 = (x38_2_2 * (x38_2_1_2 +
            at(int32_t,at(struct array,mem,2),i40)));
        temp39_1 = v41;
        temp39_2_1_1 = 0;
        temp39_2_1_2 = (x38_2_1_2 +
            at(int32_t,at(struct array,mem,2),i40));
        temp39_2_2 = v41;
        x38_2_1_1 = temp39_2_1_1;
        x38_2_1_2 = temp39_2_1_2;
        x38_2_2 = temp39_2_2;
        at(int32_t,at(struct array,mem,0),(i40 + 1)) =
            temp39_1;
    }
}
x42_2_1_1 = 0;
x42_2_1_2 = 0;
x42_2_2 = x38_2_2;
increaseLength(&at(struct array,mem,0), 1);
{
    uint32_t i44;
    for(i44 = 0; i44 < 1; i44 += 1)
    {
        int32_t temp43_1;
        uint32_t temp43_2_1_1;
        int32_t temp43_2_1_2;
        int32_t temp43_2_2;

        temp43_1 = 0;

```

```

        temp43_2_1_1 = 0;
        temp43_2_1_2 = 0;
        temp43_2_2 = 0;
        x42_2_1_1 = temp43_2_1_1;
        x42_2_1_2 = temp43_2_1_2;
        x42_2_2 = temp43_2_2;
        at(int32_t,at(struct array,mem,0),((i44 + 1) +
            v3))=temp43_1;
    }
}
x45_2_1_1 = 0;
x45_2_1_2 = 0;
x45_2_2 = x42_2_2;
increaseLength(&at(struct array,mem,0), v20);
{
    uint32_t i47;
    for(i47 = 0; i47 < v20; i47 += 1)
    {
        int32_t temp46_1;
        uint32_t temp46_2_1_1;
        int32_t temp46_2_1_2;
        int32_t temp46_2_2;
        int32_t v48;

        v48 = (x45_2_2 * (x45_2_1_2 +
            at(int32_t,at(struct array,mem,4),i47)));
        temp46_1 = v48;
        temp46_2_1_1 = 0;
        temp46_2_1_2 = (x45_2_1_2 +
            at(int32_t,at(struct array,mem,4),i47));
        temp46_2_2 = v48;
        x45_2_1_1 = temp46_2_1_1;
        x45_2_1_2 = temp46_2_1_2;
        x45_2_2 = temp46_2_2;
        at(int32_t
            ,at(struct array,mem,0),(((i47 + 1) + v3) + 1)
            )=temp46_1;
    }
}
x49_2_1_1 = 0;
x49_2_1_2 = 0;
x49_2_2 = 0;
setLength(out1, 1);
{
    uint32_t i51;
    for(i51 = 0; i51 < 1; i51 += 1)
    {
        int32_t temp50_1;
        uint32_t temp50_2_1_1;
        int32_t temp50_2_1_2;
        int32_t temp50_2_2;

        temp50_1 = 0;
        temp50_2_1_1 = 0;

```

```

        temp50_2_1_2 = 0;
        temp50_2_2 = 0;
        x49_2_1_1 = temp50_2_1_1;
        x49_2_1_2 = temp50_2_1_2;
        x49_2_2 = temp50_2_2;
        at(int32_t, (* out1), i51) = temp50_1;
    }
}
copyArrayPos(out1, 1, at(struct array, mem, 0));
}

```

Irodalomjegyzék

- [1] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegard, Anders Persson, Mary Sheeran, Josef Svenningsson and András Vajda. Feldspar: A Domain Specific Language for Digital Signal Processing algorithms. On 2010 8th IEEE/ACM International Conference on, page 169 - 178, 2010.
- [2] Duncan Coutts. Stream Fusion: Practical shortcut fusion for coinductive sequence types. A thesis submitted for the degree of Doctor of Philosophy. University of Oxford.
- [3] Duncan Coutts, Roman Leshchinskiy, Don Stewart. Stream Fusion From Lists to Streams to Nothing at All. In ICFP '07 Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, 2007, page 315-326.
- [4] Lennart Augustsson, Howard Mansell, Ganesh Sittampalam. Paradise: A two-stage DSL embedded in Haskell. In ICFP '08 Proceedings of the 13th ACM SIGPLAN international conference on Functional programming, 2008, 225-228.
- [5] Paul Hudak. Modular Domain Specific Languages and Tools. In Software Reuse, 1998. Proceedings. Fifth International Conference on, 1998, page 134 - 142.
- [6] Feldspar. Functional Embedded Language for DSP and PARallelism. <http://feldspar.inf.elte.hu/>.
- [7] Emil Axelsson, Anders Persson, Mary Sheeran, Gergely Dévai, Josef Svenningsson. A Tutorial on Programming in Feldspar. feldspar.inf.elte.hu/feldspar/documents/FeldsparTutorial.pdf.
- [8] Nurhan Karaboga. Digital IIR Filter Design Using Differential Evolution Algorithm. EURASIP Journal on Applied Signal Processing, 2005, page 1269-1276.