1. Bevezetés

Euego Moggi [6] cikkében javasolta először a kategóriaelmélet egyik fogalmának a monádnak (egyes irodalmi források triple-nek nevezik) használatát funkcionális programnyelvekben. Az ötlete az volt, hogy bizonyos számítástípusok jól modellezhetők a fogalom segítségével. Természetesen a gondolat nem volt előzmények nélküli, ugyanis a kategóriaelmélet fogalmai nagyon erős eszközöket adnak különböző struktúrák modellezésére. Ezek között szerepelnek az olyan alapvető algebrai struktúrák, mint például a félcsoportok és a monoidok, de sokkal összetettebb, érdekesebb fogalmak is. A különböző algebrai struktúrák kapcsán bevezethető szabad struktúrák például általánosan definiálhatók kategóriaelméleti eszközökkel, de a direktszorzat általános definíciójához is eljuthatunk velük, vagy egy összetettebb példa: lehetőség van az úgynevezett cratesian-closed kategóriákban a típusos lambda-kalkulus modellezésére is. Ezek azonban csak kiragadott elemek [1]-ből, egy nagyon jó kategóriaelméleti bevezetőből.

Visszakanyarodva Moggi cikkéhez, a funkcionális programnyelvek különböző fogalmai is kezelhetők az elmélet keretein belül, ezek voltak tehát ötletének előzményei.

Dolgozatunk célja kettős, egyrészt megpróbáljuk a funkcionális programnyelvek egy olyan modelljét adni, amin keresztül Moggi gondolata is bemutatható. Ehhez a kategóriaelmélet különböző alapfogalmainak ismertetésén keresztül jutunk el. Az egyre absztraktabb fogalmak bevezetése során, igyekszünk megfelelő példákkal alátámasztani a definíciókat, ezzel könnyítve megértésüket. Több segédanyagunk is lesz ebben: a mindenki által ismert algebrai struktúrák, a korábban bevezetett fogalmak és saját modellünk a funkcionális programnyelvekről. A dolgozat első fejezetének végére elérkezünk a monádok definíciójához, ezután modellünkben is megvizsgáljuk bevezetésüket. Ezen a ponton azonban szemléletet kell váltanunk, ugyanis második célunk a monádok alkalmazási lehetőségeinek feltárása. Ehhez ugyan alkalmas lenne a modell továbbfejlesztett változata, azonban maga a fogalom annyira absztrakt, hogy nekünk is absztraktabb eszközhöz kell nyúlnunk, és elhagyjuk a kategóriaelméleti párhuzam folytatását. Ezután egyszerűen azt mondjuk, hogy látható, hogy a modell megállja a helyét, a különböző irodalmi hivatkozások további finomítási lehetőségeket adnak, és fontos hiányosságokra hívják fel a figyelmet, de az alkalmazási lehetőségek megértéséhez, például a Moggi cikkében javasolt számítástípusok példányosításához, jóval egyszerűbb lesz egy valódi funkcionális nyelvben gondolkodni. Hiszen képzeljük csak el: elméletileg lehet assemblyben is elosztott objektum orientált programokat írni, de mégsem szokás, hiszen az absztrakció pontosan azt jelenti, hogy a számunkra érdektelen részletektől (például a konkrét architektúrától) eltekinthessünk, és az igazán fontos fogalmakra koncentrálhassunk.

2. Kategóriaelméleti összefoglaló

2.1. Alapfogalmak

Mivel egy matematikai elmélet alapfogalmainak ismertetését tűztük ki részcélként, mindenképpen szükségünk lesz néhány alapfogalomra, ezeket [1] alapján vezetjük be. A kategóriaelmélet tulajdonképpen speciális tulajdonságú gráfokkal, és ezek különböző leképezéseivel foglalkozik, így az ezekkel kapcsolatos fogalmak rögzítésével kell kezdenünk.

2.1.1. Definíció Gráf
A gráfok definíciójában nem [3]-t követjük, mert nemcsak olyan gráfokat fogunk használni, aminek a pontjait elképzelhetjük egy halmaz elemeiként. Helyette [1] megközelítését alkalmazzuk, és azt mondjuk, hogy egy gráf megadása a pontjainak és éleinek rögzítését jelenti. Minden élhez tartozik egy kezdő- és egy végpont. Ha f egy él a kezdő-, és b végponttal, azt $f : a \to b$-vel jelöljük. Az él két végére a $kezd(f)$ és $veg(f)$ jelöléseket használjuk. Megengedjük a többszörös éleket, azaz "a" és "b" végpontokkal rendelkezhet több különböző él is. Továbbá megengedjük, hogy egy él kezdőpontja és végpontja megegyezzen (hurok él).
Egy G gráf pontjaira G_0-lal, éleire G_1-el fogunk hivatkozni, ezeket [1] terminológiájával gyűjteménynek nevezzük.

2.1.2. Megjegyzés Russell-paradoxon
Felmerül a kérdés, hogy miért nem halmazokat használtunk a fenti definícióban. Ennek oka a klasszikus halmazelméletből ismert Russell-paradoxon problémája. Vegyük az alábbi kifejezést:

$$T = \{S | S \text{ halmaz és } S \notin S\}$$

Azaz legyen T azoknak a halmazoknak a halmaza, amik nem tartalmazzák önmagukat elemként. Ezzel a definícióval csak egy baj van: T nem halmaz. Ugyanis indirekt tegyük fel, hogy T mégis egy halmaz. Most az a kérdés, hogy tartalmazza-e önmagát. Ha $T \in T$-ből indulunk ki, akkor a definíció szerint azt kapjuk, hogy $T \notin T$, hiszen T pontosan az önmagukat nem tartalmazó halmazokból áll. Megfordítva $T \notin T$-ből viszont $T \in T$-t kapjuk. Így sem $T \in T$ sem $T \notin T$ nem teljesülhet, tehát ellentmondásra jutunk azzal, hogy T-t halmaznak tekintettük.

2.1.3. Definíció G gráfot *kis* gráfnak nevezzük, ha G_0 és G_1 halmazok, egyébként G *nagy* gráf.

2.1.4. Definíció Adott egy G gráf és két csúcsa legyen x és y. x és y közti k *hosszú út*nak nevezzük azt az $(f_1, f_2, ..., f_k)$ élsorozatot, amelyre teljesül, hogy:

(i) $kezd(f_k) = x$,

(ii) $veg(f_i) = kezd(f_{i-1})$, ha $i = 2, ..., k$ és

(iii) $veg(f_1) = y$.

A definíció nem köti ki, hogy az élsorozat tagjai különbözők. Figyeljük meg azt is, hogy a sorozat tagjait, a várakozással talán ellentétesen, az élsorozat végpontjától kezdve számozzuk. Erre a kompozícióval való egységesség miatt van szükségünk.

$$x \xrightarrow{f_k} \bullet \xrightarrow{f_{k-1}} \dots \xrightarrow{f_2} \bullet \xrightarrow{f_1} y$$

A gráf minden pontjához rendelhetünk egy nulla hosszú utat, amit üres útnak nevezünk, és ()-el fogunk jelölni. Ha f a gráf egy éle, x kezdő-, és y végponttal, akkor (f) egy 1 hosszú út x és y között.

2.1.5. Jelölés Egy G gráf k hosszú útjainak halmazát G_k-val fogjuk jelölni. Ez ugyan ütközik a gráf definíciójánál bevezetett G_0-al és G_1-el, de egyrészt a későbbiekben a kontextusból mindig kiderül, hogy éppen melyikről beszélünk, másrészt kölcsönösen egyértelmű hozzárendelés képezhető az élek és az egy hosszú utak, valamint a gráf csúcsai és üres útjai között.

Szükségünk lesz még gráfok közötti, valamilyen értelemben, struktúra őrző leképezésekre is, amit az irodalom gráf homomorfizmusnak nevez.

2.1.6. Definíció Adott egy G és egy H gráf. A $\phi : G \to H$ leképezést *gráf homomorfizmus*nak nevezzük, ha ϕ egy függvénypár, egy $\phi_0 : G_0 \to H_0$ és egy $\phi_1 : G_1 \to H_1$ függvény együttese, azzal a tulajdonsággal, hogy ha $u : m \to n$ G-nek egy éle, akkor $\phi_1(u) : \phi_0(m) \to \phi_0(n)$ egy él H-ban.

2.1.7. Definíció Definiáljuk még a *gráf homomorfizmusok kompozícióját* is. Ehhez legyenek $\mathcal{G}$, $\mathcal{H}$ és $\mathcal{I}$ gráfok, $\phi : \mathcal{G} \to \mathcal{H}$ és $\psi : \mathcal{H} \to \mathcal{I}$ gráf homomorfizmusok. Legyen ekkor $(\psi \circ \phi)_0 = \psi_0 \circ \phi_0$ és $(\psi \circ \phi)_1 = \psi_1 \circ \phi_1$ utasításokkal $\psi \circ \phi = ((\psi \circ \phi)_0, (\psi \circ \phi)_1)$
Egyszerűen belátható, hogy az így kapott leképezés maga is gráf homomorfizmus, sőt a kompozíció asszociatív.

2.2. Kategóriák

A rövid bevezetés után elkezdhetünk kategóriaelméleti fogalmakkal foglalkozni. Mindenekelőtt természetesen a kategóriákat fogjuk definiálni, majd bevezetjük a modell számára nagyon fontos funktorokat, később megismerkedünk a természetes transzformációkkal (ezek ddd funktorok közötti leképezések), végül rátérünk a monádokra, az ekvivalens Kleisli triple-re és az ezekkel rokon adjungáltakra.

2.2.1. Definíció Legyen G egy gráf, és vegyünk egy $c : G_2 \to G_1$ és egy $id : G_0 \to G_1$ függvényt. Nevezzük c-t *kompozíciónak*, és ha $(g, f) \in \mathcal{D}(c)$ ($\mathcal{D}$ az értelmezési tartományt jelenti), akkor g és f kompozícióját, azaz $c((g, f))$-et, jelöljük $g \circ f$-el. Ha $A \in G_0$, akkor $id(A)$-t id_A-val jelöljük, és A identitás élének nevezzük.

Akkor mondjuk, hogy a G gráf *kategória*, ha teljesülnek az alábbi feltételek:

(C-1) $g \circ f$ kezdőpontja azonos f kezdőpontjával, végpontja pedig g végpontja;

(C-2) $(h \circ g) \circ f = h \circ (g \circ f)$, ha az egyenlőség bármelyik oldala értelmezett;

(C-3) id_A kezdő- és végpontja egyaránt A;

(C-4) ha $f : A \to B$, akkor $f \circ id_A = id_B \circ f = f$.

Az irodalomban id_A-t gyakran csak A-val jelölik. Mivel a kompozíció értelmezési tartománya G_2, így $g \circ f$ pontosan akkor definiált, ha f végpontja és g kezdőpontja megegyezik. Ebből és C-1-ből következik, hogy ha C-2 valamelyik oldala definiált, akkor a másik oldala is az.

2.2.2. Megjegyzés A gráfok és a kategóriák közti különbség hangsúlyozására a továbbiakban a kategóriák pontjait objektumoknak nevezzük, és a kategóriák jelölésére az írott nagybetűket használjuk ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, stb.).

A fogalom bevezetése után nézzünk néhány példát kategóriára.

2.2.3. Példa Egyszerű kategóriák
A legegyszerűbb kategória az üres gráf, aminek nincsenek sem objektumai sem élei. A következő legkisebb kategóriának egy objektuma és egy éle van (az objektum identitás éle). Ezt a kategóriát a továbbiakban **1**-el jelöljük. Két objektummal rendelkező kategóriában már több lehetőségünk van, ugyanis dönthetünk arról, hogy az objektumok között legyen-e él, így kapjuk az **1**+**1** és **2** kategóriákat.

2.2.4. Példa Halmazok kategóriája
Halmazok kategóriája az a kategória, aminek objektumai a halmazok, élei függvények, a c kompozíció megegyezik a függvénykompozícióval, és egy S halmazra id_S az S-ről S-re képező identitás függvény. Ahhoz, hogy belássuk, hogy ez valóban kategória azt a két állítást kell megvizsgálnunk, hogy a függvénykompozíció asszociatív, és bármely $id_S : S \to S$ identitás függvényre teljesül, hogy $f \circ id_S = id_S \circ f = f$. Ezek nyilvánvalóan fennállnak, így bizonyításukkal itt nem foglalkozunk, megemlítjük viszont, hogy ez a kategória, amit az irodalom **Set**-el jelöl, tipikus példája az úgynevezett *nagy* kategóriáknak, abban az értelemben, hogy a kategóriát tartó gráf *nagy*, hiszen pontjai ($\mathbf{Set_0}$) nem alkotnak halmazt ([1]).

2.2.5. Példa Gráfok kategóriája
Azt a kategóriát, aminek objektumai kis gráfok, élei pedig gráfok közötti homomorf leképezések **Grf**-el jelöljük. Egyszerűen belátható, hogy **Grf** valóban kategória, hiszen a gráfok identitás homomorfizmusai tekinthetők a **Grf**-ben megkövetelt identitás éleknek, az asszociatív tulajdonság pedig a gráf homomorfizmusok asszociativitása miatt áll fenn.

Algebrai struktúrák és kategóriák

A kategóriaelmélet egyik nagy ereje, hogy gyakran alkalmazható, általános eszközt ad különböző struktúrák modellezésére, ezért mindenképpen érdemes megvizsgálnunk néhány egyszerű algebrai struktúra és a kategóriák közti kapcsolatot. Ráadásul ezek a példák hasznosak lesznek a későbbiekben ismertetetésre kerülő fogalmak megértésénél is.

2.2.6. Példa Előrendezett és rendezett halmazok
Ha S egy halmaz, akkor $\alpha \subseteq S \times S$ egy *bináris reláció* S-en. Sokszor $(x, y) \in \alpha$ helyett $x\alpha y$-t szoktak írni. Azt mondjuk, hogy α *reflexív*, ha minden $x \in S$-re teljesül, hogy $x\alpha x$; α *tranzitív*, ha minden $x, y, z \in S$ esetén $x\alpha y$ és $y\alpha z$ fennállásából $x\alpha z$ következik.

Akkor mondjuk, hogy az S halmaz egy α bináris relációval előrendezett halmaz, ha α tranzitív és reflexív. Ez a struktúra az alábbi $C(S, \alpha)$ kategóriát határozza meg:

CO-1 legyenek $C(S, \alpha)$ objektumai S elemei;

CO-2 ha $x, y \in S$ és $x\alpha y$, akkor $C(S, \alpha)$ pontosan egy élet tartalmaz x-ből y-ba;

CO-3 ha x és y nem állnak relációban, akkor $C(S, \alpha)$-ban nincs él x és y között.

Ha α antiszimmetrikus is, azaz $x\alpha y$ és $y\alpha x$ teljesüléséből $x = y$ következik, akkor (S, α) parciálisan rendezett. Két jó példa erre a struktúrára a valós számok halmaza a szokásos rendezéssel, jelölésben $(\mathbf{R}, \leq)$, és $(\mathcal{P}(S), \subseteq)$, azaz S részhalmazai a tartalmazás relációval.

Egy másik példa parciálisan rendezett halmazokra a funkcionális programnyelvekben használatos osztály fogalom. Például a Haskellben minden típushoz hozzárendelhetünk osztályokat úgy, hogy az osztályok definíciójában rögzített műveleteket implementáljuk az adott típushoz. A típusosztályokat a Haskell fejlesztői ellátták az öröklődés és a többszörös öröklődés lehetőségével is, például az `Num` osztály elemei (mint az `Int`, `Integer`, `Float` és `Double`) egyúttal a `Show` osztálynak is elemei, mivel a `Num` a `Show` leszármazottja.
Az öröklődési reláció megfelel egy parciális rendezésnek, így egyszerűen rendelhetünk a Haskell osztály-hierarchiájához is egy kategóriát.

2.2.7. Definíció Félcsoportok
Egy S halmaz egy asszociatív bináris $m : S \times S \to S$ művelettel *félcsoportot* alkot. Jelölésben csak st-t írunk az $(s, m) \in S$, $m(s, t)$ helyett, m-et szorzásnak is nevezzük, bár nem követeljük meg, hogy kommutatív legyen (így a *kommutatív félcsoportokhoz* jutnánk).

2.2.8. Definíció Egységelem, monoid
Ha S egy félcsoport, és valamilyen $e \in S$ elemre teljesül, hogy tetszőleges $s \in S$-re $se = es = s$, akkor e-t *egységelem*nek, S-et a bináris műveletével és e-vel *monoid*nak nevezzük.

2.2.9. Példa Egy egyszerű példa félcsoportra a pozitív egész számok halmaza az összeadással. Ha a halmazhoz a nullát is hozzávesszük monoidot kapunk.

2.2.10. Példa Monoidok mint kategóriák
Minden M monoidhoz hozzárendelhető egy $C(M)$ kategória a következőképpen:

CM-1 $C(M)$-nek mindössze egy objektuma van, ezt jelöljük mondjuk *-gal.;

CM-2 $C(M)$ élei M elemei, * kezdő- és végponttal;

CM-3 a kategória kompozíciójának válasszuk M bináris műveletét.

Definiáljuk még monoidok és félcsoportok homomorfizmusait a későbbi példák kedvéért. Ezek, a homomorfizmusokra általában jellemző módon, struktúratartó leképezések lesznek.

2.2.11. Definíció Félcsoport és monoid homomorfizmusok
Legyen S és T két félcsoport. A $h : S \to T$ leképezést *félcsoport homomorfizmusnak* nevezzük, ha minden $s, s' \in S$ esetén $h(ss') = h(s)h(s')$. Ha S és T monoidok, e az S egységeleme és $h(e)$ a T egységeleme (azaz a leképezés egységelemhez egységelemet rendel), akkor h-t *monoid homomorfizmusnak* nevezzük.

Egy másik jólismert fogalom a Kleene-lezárás, ami tulajdonképpen egy halmaz által definiált szabad monoid:

2.2.12. Definíció Az A halmaz **Kleene-lezártját** A^*-gal jelöljük. Ez az A-beli elemekből alkotott véges hosszú sorozatok halmaza. A konkatenáció művelete alatt a sorozatok egymás után írását értjük:

$$(a,b,c,d)(e,f,g) = (a,b,c,d,e,f,g)$$

A konkatenáció nyilván asszociatív, egységeleme az üres sorozat: $()$, ebből következik, hogy A^* ezzel a művelettel monoidot alkot, ezt $F(A)$-val jelölik és **szabad monoid**nak is nevezik.
Az A halmazt szokás ábécének, a A^* elemeit sztringeknek is nevezni, és sokszor felteszik, hogy A véges halmaz.

A Kleene-lezárással rokon a gráf által generált szabad kategória fogalma.

2.2.13. Definíció Adott G gráfhoz, rendeljük a következő $F(G)$ kategóriát. A kategória objektumai legyenek G pontjai, élei legyenek a G-beli utak. A kompozíciót definiáljuk a következőképpen:

$$(f_1, f_2, ..., f_k) \circ (f_{k+1}, f_{k+2}, ..., f_n) = (f_1, f_2, ..., f_n)$$

A kompozíció asszociatív és minden A objektumra id_A az üres út A-ból A-ba. $F(G)$-t a *G által generált szabad kategóriának* nevezzük.

Az egyetlen csúcsból álló él mentes gráf által generált szabad kategóriának egy csúcsa és egy éle van (a csúcs identitás éle). Az egy csúccsal és egy éllel rendelkező gráfhoz rendelt szabad kategóriának egy csúcsa van, és végtelen sok éle. Tulajdonképpen izomorf egy monoidhoz rendelt kategóriával, de egy egyelemű halmaz Kleene-lezártjaként is gondolhatunk rá.

Egy általános kis G gráf szabad kategóriáját is érdemes halmazok Kleene-lezártjához hasonlóan megközelíteni. A gráf útjai megfelelnek a Kleene-lezárt sztringjeinek. A kettő között az a különbség, hogy míg a halmazból kapott sztringekben egymás után bármilyen betűk állhatnak, addig G útjait csak az élek megfelelő módon történő egymás után fűzésével kaphatjuk, ami megszorítás jelent a generálható sztringekre nézve. Egy olyan gráfban, amiben minden csúcs minden csúccsal összekötött, a csúcsok halmazának Kleene-lezártja egyenlő az utakból nyerhető sztringek halmazával.

Szükségünk lesz még néhány, a kategóriák éleivel kapcsolatos, tulajdonságra:

2.2.14. Definíció Ha a $\mathcal{C}$ kategória $f : A \to B$ éléhez létezik olyan $g : B \to A$ él, amire teljesül, hogy

$$f \circ g = id_B$$

$$g \circ f = id_A$$

akkor azt mondjuk, hogy A **izomorf B-vel**, amit $A \cong B$-vel jelölünk. f és g a két objektum közötti **izomorfizmus**ok.

A kategóriák identitás élei izomorfizmusok.

Ha egy monoidhoz tartozó kategória minden éle izomorfizmus, akkor a monoid egy csoport (group), hiszen ez pontosan azt jelenti, hogy minden elemnek van inverze. Ebből kiindulva azokat a kategóriákat, amikben minden él izomorfizmus groupoidnak szokták nevezni.

Egyszerűen belátható, hogy **Set** izomorfizmusai a bijektív függvények.

2.2.15. Definíció A $\mathcal{C}$ kategória T objektumát *terminális*nak nevezzük, ha igaz, hogy minden $A \in \mathcal{C}$ objektumhoz létezik pontosan egy A $\to$ T él. A terminális objektumokat 1-el fogjuk jelölni.

A *kezdeti elem* fogalma az előző duálisa. Az ilyen objektumból a kategória minden objektumába pontosan egy él mutat. Az irodalom általában 0-val jelöli őket.

A halmazok és függvények kategóriájában az üres halmaz a kezdeti elem, hiszen az $f : \emptyset \to A$ függvény egyértelmű: maga is az üres halmaz. Hasonlóan minden egy elemű halmaz terminális, ugyanis az $f : A \to \{t\}$ függvényt csak úgy lehet definiálni, hogy minden elemet t-be vigyen. A parciálisan rendezett halmazok kategóriáiban a kezdeti elem a halmaz egy abszolút minimuma, a terminális elemek abszolút maximumok.
Egyszerűen belátható, hogy bármely két terminális elem egymással izomorf, hasonló igaz az állítás duálisára: a kezdeti elemek is izomorfak egymással.

2.2.16. Definíció Legyen $\mathcal{C}$ és $\mathcal{D}$ két kategória. Definiáljuk a $\mathcal{C} \times \mathcal{D}$ direktszorzatot.
Egy (C, D) $\mathcal{C} \times \mathcal{D}$-beli objektum a $\mathcal{C}_0$-beli C és $\mathcal{D}_0$-beli D objektumok rendezett párja. Egy $(f, g) : (C, D) \to (C', D')$ $\mathcal{C} \times \mathcal{D}$-beli él az $f : C \to C'$ $\mathcal{C}$-beli és $g : D \to D'$ $\mathcal{D}$-beli élekből alkotott pár. (C, D) identitás éle legyen (id_C, id_D), végül ha $(f', g') : (C', D') \to (C'', D'')$ egy másik él, akkor a kompozíció legyen:
$$(f', g') \circ (f, g) = (f' \circ f, g' \circ g) : (C, D) \to (C'', D'')$$
Egyszerűen belátható, hogy az így definiált $\mathcal{C} \times \mathcal{D}$ is kategória. Ehhez hasonlóan definiálható a kettőnél több komponensű direktszorzat is, és ha a direktszorzat komponensei ugyanazok a kategóriák, használjuk a szokásos $\mathcal{C}^2$, $\mathcal{C}^3$, ... , $\mathcal{C}^n$ jelöléseket.

Funkcionális programnyelvek, mint kategóriák

Ebben a részben megmutatjuk, hogy a funkcionális programnyelvek, bizonyos feltételek teljesülése mellett, tulajdonképpen kategóriákként is felfoghatók. Nem célunk a teljesség, csak egy egyszerű, modellt adunk, azonban ezt tovább finomítjuk a későbbiekben bemutatásra kerülő kategóriaelméleti fogalmak segítségével. A modell bevezetésének fő oka, hogy [Moggi 69] egy hasonló modellre támaszkodott, mikor a monádokat a számítások absztrakciójaként közelítette meg.

Az alábbi modell csak egy példa a kategóriaelmélet és a számítástudomány összefonódására, sok hasonlót lehetne találni, példaként említjük a dedukciós rendszerek kategóriaelméleti megfogalmazását, amit [1] ötödik fejezete is tárgyal.

Egy funkcionális programnyelvet durván úgy határozhatunk meg, mint egy olyan programnyelvet, ami néhány beépített adattípust tartalmaz, ezek közt műveleteket definiál, és vannak olyan konstruktorai, amivel a meglevő műveletekből újakat hozhatunk létre, illetve újakkal bővíthetjük a meglevő típusok halmazát. A tisztán funkcionális nyelvekből hiányzik a változók és az értékadások fogalma.

Egy funkcionális programnyelv, tehát a következő összetevőkkel rendelkezik:

FPL-1: a nyelv által definiált egyszerű típusok

FPL-2: minden típushoz léteznek konstansok

FPL-3: típusok közötti műveletek, függvények

FPL-4: típusokra, és műveletekre alkalmazható konstruktorok a származtatott adattípusok és műveletek definiálásához.

A nyelv felfogható a primitív adattípusokból származtatható típusok, és a primitív műveletekből származtatható műveletek összességeként. Itt a primitív szót a nyelv által definiált alaptípusokra és műveletekre használjuk a konstruktorokkal származtatottakkal szemben. Ha három feltételezéssel élünk a funkcionális programnyelvekről, és teszünk egy apró változtatást a fentiekben, azonnal látható, hogy egy L funkcionális nyelvhez hogyan rendelhetünk egy $C(L)$ kategóriát.

A-1: Feltételezzük, hogy létezik egy identitás művelet (id_A) minden A típushoz, ami alkalmazásakor visszaadja a paraméterül kapott értéket.

A-2: Bevezetünk egy további típust, amit 1-el jelölünk. $C(L)$-ben minden A típusú c konstanshoz bevezetünk egy $c : 1 \to A$ irányított élet.

A-3: Feltételezzük, hogy a nyelv tartalmaz egy kompozíció konstruktort, ami egy olyan f műveletből, ami A típusú értékekhez B típusú értékeket rendel, és egy olyan g-ből, ami B típusú értékekhez C típusúakat rendel, létrehoz egy A-ról C-re képező műveletet (programot), f-nek és g-nek a szokásos értelemben vett kompozícióját, jelölésben $f; g$-t.

A-4: Végül feltételezzük, hogy L műveletei teljesen definiált függvények.

A funkcionális programnyelvek általában teljesítik az A-1 és A-3 kikötéseket, azaz vannak identikus műveleteik és kompozíció konstruktoraik. Az A-2 feltételre csak technikai okokból van szükségünk, a nyelv kifejezőerején ez nyilván nem változtat, ráadásul összhangban van [Csorny] meglátásával, miszerint a funkcionális világban minden függvény, és csak az emberi képzelet láttat velünk konstansokat a függvények mögé. Egyedül A-4 jelentene komoly megszorítást, hiszen így például a nullával való osztást is értelmeznünk kellene, azonban a modell úgynevezett bottom-elemek ($\perp$) bevezetésével finomítható, így ezt a feltételt ki is tudjuk váltani.

A függvénykompozíció asszociatív művelet, abban az értelemben, hogy ha valamely f, g, h-ra $(f; g); h$ definiált, akkor $f; (g; h)$ is definiált, és a két művelet megegyezik. Az A-1 és A-3 feltételek együtt azt jelentik, hogy tetszőleges $f : A \to B$ esetén $f; id_B$ és $id_A; f$ is definiált, és mindkettő megegyezik f-fel. Azaz $f = f; id_B$ és $id_A; f = f$.

A fenti feltételek mellett egy L funkcionális nyelvhez a következő $C(L)$ kategóriát rendelhetjük:

FPC-1: $C(L)$ objektumai legyenek L típusai - pontosabban az objektumok legyenek a típusértékek halmazai.

FPC-2: L primitív és származtatott műveletei legyenek $C(L)$ irányított élei.

FPC-3: Az irányított élek forrása legyen a megfelelő művelet paraméterének típusa, a végpontja pedig a művelet eredményének típusa. Maga az él legyen egy függvény a két halmaz között, hasonlóan **Set** éleihez.

FPC-4: A kategória kompozíciója legyen L kompozíció konstruktora (fordított sorrendben írva a paramétereket).

FPC-5: Az identitás élek legyenek az identitás műveletek.

Megjegyezzük, hogy ha a funkcionális nyelv tartalmazza a függvénydefiníció konstruktorát, akkor $C(L)$ **Set**-hez hasonlóan a típusok közötti összes leképezést tartalmazza. A továbbiakban gyakran feltesszük, hogy L-re teljesül ez a feltétel is.

Figyeljük meg, hogy $C(L)$ csak egy modellje L-nek, hiszen míg $C(L)$-ben $f; id_B = f$, addig L-ben ez két különböző programot jelentene.

2.2.17. Példa Nézzünk egy konkrét példát. Legyen L-nek három alaptípusa: a természetes számok típusa (NAT), a logikai értékek típusa ($BOOLEAN$) és a karaktereké ($CHAR$). A műveletek:

(i) NAT-nak van egy konstansa $0 : 1 \to NAT$ a természetes számok 0 eleme, és a halmaz szokásos rákövetkező művelete $succ : NAT \to NAT$.

(ii) A $BOOLEAN$ típusban van két konstans $true, false : 1 \to BOOLEAN$, és egy negáció művelet $\neg : BOOLEAN \to BOOLEAN$ a következő definícióval: $\neg \circ true = false$ és $\neg \circ false = true$.

(iii) $CHAR$ minden karakterhez tartalmaz egy $c : 1 \to CHAR$ konstanst.

(iv) Legyen két típuskonverziós műveletünk is. $ord : CHAR \to NAT$ és $chr : NAT \to CHAR$. Ezekre érvényes, hogy: $chr \circ ord = id_{CHAR}$ (chr-t felfoghatjuk úgy, hogy a konverziót modulo a karakterek száma szerint végzi, így értelmes minden természetes számra.)

Egy példaprogram lehetne a rákövetkező karakter meghatározása: $next : CHAR \to CHAR$. $next = chr \circ succ \circ ord$. Az L-hez rendelt $C(L)$ kategóriában $next$ egy irányított él lesz csakúgy, mint az összes függvénykompozíció, de nézzük meg pontosan, mi lesz az L-hez kapcsolódó kategória:

F-1 $C(L)$ négy objektumot tartalmaz: a három beépített típust, azaz NAT-ot, $CHAR$-t, $BOOLEAN$-t, valamint az 1-et.

F-2 $C(L)$ irányított élei a nyelv programjai, ahol az ekvivalens programokhoz azonos élek tartoznak, azaz például a fenti $next = chr \circ succ \circ ord : CHAR \to CHAR$ és a $chr \circ succ \circ ord \circ chr \circ ord : CHAR \to CHAR$ egyenlők (iv) miatt.

NAT konstansai felírhatók $succ \circ succ \circ ... \circ succ \circ succ \circ 0$ alakban, ahol $succ$ véges sokszor fordul elő. Ezeket el is nevezhetjük a természetes számok halmazának elemeivel. Legyen $1 = succ \circ 0$, és indukcióval, ha az n természetes számhoz már rendeltünk értéket, akkor legyen $n + 1 = succ \circ n$.

Felhívjuk a figyelmet arra, hogy C(L) minden L-ben megírható programhoz tartalmaz egy élet, így például a következőhöz is:

$$(+2) : NAT \to NAT$$

$$(+2)(m) = succ \circ succ \circ m$$

Felmerül a kérdés, hogy mit kezdjünk a többváltozós függvényekkel, hiszen jelen pillanatban még az összeadást sem tudjuk kifejezni. Első ötletünk a direktszorzatok bevezetése lenne, tehát feltételezhetnénk, hogy C(L) tartalmazza L alaptípusainak direktszorzatait is, így a többváltozós függvények a matematikában szokásos módon direktszorzatokon lennének értelmezve, azaz például:

$$f : A \times B \to C$$

alakban. Egészítsük is ki a modellünket:

FPC-6 bármely két A,B C(L)-beli objektumhoz tegyük fel, hogy létezik az A × B objektum, a halmazok szokásos direktszorzata.

Ezzel tulajdonképpen a később bevezetésre kerülő "direktszorzat" típuskonstrukciós műveletet is megalapoztuk. Itt jegyezzük meg, hogy a fenti konstrukcióval még mindig **Set**-en belül maradunk: az objektumok megfeleltethetők **Set**-beli objektumoknak, tehát nem fordulhat elő, hogy az eredmény esetleg "túl nagy" lesz.

Szükségünk lesz az élek bővítésére is, két élhez bevezetjük a kezdőpontjaik direktszorzatából a végpontjaik direktszorzatába vezető élet a következőképpen:

FPC-7 Egy $f : A \to B$ és egy $g : C \to D$ C(L)-beli élre legyen az

$$f \times g : A \times B \to C \times D$$

él az a függvény C(L)-ben, amire tetszőleges $(a, b) \in A \times B$ elem esetén:

$$(f \times g)(a, b) = (f(a), g(b)) \tag{2.1}$$

A funkcionális programnyelveket ismerők, azonban vitatkoznának ezzel a megoldással, létezik ugyanis a "currying"-nek nevezett eljárás, ami lényegében arról szól, hogy az

$$f : A \times B \to C$$

értelmezés helyett f-et olyan egyparaméteres függvénynek tekintjük, aminek az értéke egy $a \in A$ helyen maga is egy függvény, ami jelen esetben $B \to C$ típusú:

$$f' : A \to (B \to C)$$

erre teljesül, hogy minden $a \in A$, $b \in B$ esetén

$$f(a, b) = f'(a)(b)$$

A mi megoldásunkkal az a baj, hogy nem tudunk benne függvény típusú értékeket használni, így hiányzik például az f'(a) lehetősége is. Ráadásul a funkcionális programnyelveket használók gyakran adnak át függvényeket paraméterként, tehát mindenképpen tovább kell bővítenünk modellünket.

2.2.18. Definíció Egy $\mathcal{C}$ kategóriát *cratesian-closed*-nak nevezünk, ha:

CCC-1: létezik terminális eleme (1).

CCC-2: minden $A, B \in \mathcal{C}_0$ objektumhoz létezik ezek $A \times B$ direktszorzata.

CCC-3: bármely két B, C $\mathcal{C}$-beli objektumhoz létezik a $[B \to C]$ objektum és egy $eval : [B \to C] \times B \to C$ él $\mathcal{C}$-ben, amire teljesül, hogy minden $f : A \times B \to C$ él esetén egyértelműen létezik egy $\lambda f : A \to [B \to C]$ él úgy, hogy

$$f = eval \circ \lambda f \times id_B$$

amit ábrával úgy szemléltethetnénk, hogy az alábbi kompozíció éppen f:

$$A \times B \xrightarrow{\lambda f \times id_B} [B \to C] \times B \xrightarrow{eval} B$$

Megjegyezzük, hogy a definíció nem teljesen egzakt, precízen a direktszorzat általános definíciójára kellene építeni, amire terjedelmi okokból nem térünk ki. Érdekességként megemlítjük, viszont, hogy azért kell a direktszorzatot általánosan meghatározni, mert több struktúra is rendelkezik ugyanazokkal a tulajdonságokkal. (Például az $A \times B$ rendezett párjait megadhatnánk fordított sorrendben is.)

Most az a kérdés, hogy mi köze a cratesian-closed kategóriáknak C(L)-hez. Ha megnézzük a feltételeket, látható, hogy C(L) teljesíti a CCC-1 és CCC-2 kikötéseket, hiszen van terminális eleme és direktszorzatai. Csak a CCC-3 feltétel teljesülését kell megoldanunk valahogy, és ez vezet el a "currying" probléma megoldásához.

Vegyük hozzá C(L) objektumaihoz tetszőleges B és C objektum esetén az $[B \to C]$ halmazt úgy, hogy elemei legyenek az C(L)-beli $B \to C$ függvények.

Mivel a $C(L)$-beli élek függvények, így az $eval : [B \to C] \times B \to C$ él adott $B, C \in C(L)_0$ esetén maga is függvény lesz. Ezt nagyon egyszerűen definiálhatjuk, hiszen $[B \to C]$-t éppen az előbb adtuk meg a $B \to C$ függvények halmazaként. Legyen tehát $b \in B$ esetén

$$eval(f, b) = f(b) \tag{2.2}$$

Ez a választás meg is felel az él ($eval$) típusára tett követelményeknek. Végül meg kell adnunk minden $f : A \times B \to C$ élhez az $\lambda f : A \to [B \to C]$ élet. Mivel ez egy olyan függvény, ami A-beli elemekhez $B \to C$ függvényeket rendel, így azt is meg kell mondanunk, hogy az $a \in A$ ponthoz rendelt függvény hogy működik egy $b \in B$-ben. Legyen tehát:

$$(\lambda f)(a)(b) = f(a, b) \tag{2.3}$$

(A zárójelet csak a hangsúlyozás kedvéért írtuk λf köré.) Az nyilvánvaló, hogy az így definiált λf egyértelmű, tehát csak az a kérdés, hogy teljesül-e CCC-3 utolsó feltétele, azaz:

$$f \quad \stackrel{?}{=} \quad eval \circ \lambda f \times id_B$$

Nézzük elemenként, tetszőleges $a \in A$, $b \in B$-re:

$$(eval \circ (\lambda f \times id_B))(a, b) =$$

a kompozíció a szokásos függvénykompozíció C(L)-ben, így:

$$= eval((\lambda f \times id_B)(a, b)) =$$

a $\times$-t komponensenként definiáltuk 2.1-ben függvényekre, tehát:

$$= eval((\lambda f(a), id_B(b))) =$$

id_B az identitás függvény:

$$= eval((\lambda f(a), b)) =$$

$eval$ 2.2-beli definíciója miatt:

$$= (\lambda f)(a)(b) =$$

végül $(\lambda f)(a)$ 2.3 definícióját felhasználva:

$$= (\lambda f)(a)(b) = f(a, b)$$

Ezzel az állítást bizonyítottuk: a fent leírtaknak megfelelően kibővített C(L) egy cratesian-closed kategória. Most nézzük meg mi történik, ha feltesszük, hogy L-ben felírható a természetes számok összeadása:

$$+ : NAT \times NAT \to NAT$$

Egyrészt ennek közvetlenül megfelel egy + függvény C(L)-ben, másrészt létezik a $\lambda+$ függvény is, és $(\lambda+)(n)$ valamilyen $n \in NAT$ természetes számra az a függvény lesz, ami a paraméteréhez n-et ad hozzá.

2.3. Funktorok

2.3.1. Funktorok

A funktor lényegében egy struktúra megőrző leképezés két kategória között. Nekünk elsősorban a típuskonstruktorok bevezetéséhez lesz rá szükségünk, de ismertetjük egyéb tulajdonságaikat is. Mivel a kategóriákat gráfokból származtattuk, a struktúra őrzés részben azt jelenti, hogy a funktoroknak a két kategória közötti gráf homomorfizmusoknak kell lenniük, de nézzük a pontos definíciót.

2.3.1. Definíció Funktor
Legyen $\mathcal{C}$ és $\mathcal{D}$ két kategória. Jelölje $F : \mathcal{C} \to \mathcal{D}$ azt az $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$ és $F_1 : \mathcal{C}_1 \to \mathcal{D}_1$, függvénypárt, amelyre teljesül, hogy:

F-1 ha $f : A \to B$ $\mathcal{C}$-ben, akkor $F_1(f) : F_0(A) \to F_0(B)$ $\mathcal{D}$-ben;

F-2 $\mathcal{C}$ minden A objektumára: $F_1(id_A) = id_{F_0(A)}$;

F-3 ha $g \circ f$ definiált $\mathcal{C}$-ben, akkor $F_1(g) \circ F_1(f)$ is definiált $\mathcal{D}$-ben, és $F_1(g \circ f) = F_1(g) \circ F_1(f)$;

ekkor F-et funktornak nevezzük.

A definícióban F-1 pontosan azt mondja ki, hogy F gráf homomorfizmus. Az F-2 kikötés miatt F megőrzi az identitás éleket, míg F-3 szerint F az élek kompozícióját a képek kompozíciójába viszi.

A szokásos jelölésmódnak megfelelően F-et használni fogjuk F_0 és F_1 helyett, azaz A objektum és f él esetén használjuk az $F(A) := F_0(A)$ és $F(f) := F_1(f)$ jelöléseket. Ez nem okoz zavart a későbbiekben, és egyszerűsíti a jelölésrendszerünket. További egyszerűsítésként néha a zárójeleket is elhagyjuk és a funktort operátor alakban használjuk: F A, F f ...

2.3.2. Példa Egyszerűen belátható, hogy egy $f : M \to N$ monoid homomorfizmus funktort generál $C(M)$ és $C(N)$ kategóriák között. Mivel $C(M)$-nek és $C(N)$-nek csak egy-egy objektuma van, így a keresett F homomorfizmus az objektumokon csak úgy definiálható, hogy egyiket a másikra képezi. F-et az élekre f-nek megfelelően definiáljuk, azaz egy tetszőleges "a" $C(M)$-beli élhez az $f(a)$ $C(N)$-beli élet rendeljük, ami létezik, hiszen f homomorfizmus, és a kategóriák élei a megfelelő monoidok elemeivel címkézettek. Ekkor az objektumok egyértelműségéből következik, hogy F teljesíti az F-1 kikötést, és f homomorfia tulajdonsága miatt fennáll F-2 és F-3 is.
A dolgot megfordítva az is igaz, hogy a két kategória közötti funktorok monoid homomorfizmusokat generálnak.

2.3.3. Példa Legyenek $C(S, \alpha)$ és $C(T, \beta)$ az S és T parciálisan rendezett halmazok által generált kategóriák. Egy $F : C(S, \alpha) \to C(T, \beta)$ funktorra F-1 miatt fennáll, hogy bármely $x, y \in S$ esetén, ha létezik él x-ből y-ba, akkor létezik él $F(x)$-ből $F(y)$-ba $C(T, \beta)$-ban. Ez az élek definíciója miatt más szóval azt jelenti, hogy ha $x\alpha y$ teljesül, akkor $F(x)\beta F(y)$ is fennáll.
Megjegyezzük, hogy az ilyen tulajdonságú F-et *monoton*nak szokták nevezni, továbbá az F-2 és F-3 kikötéseknek ebben az esetben nincs megszorító ereje, hiszen a szóbanforgó kategóriákban két objektum között csak egyetlen él lehet, így a feltételek arra egyszerűsödnek, hogy egy-egy él önmagával egyenlő.

2.3.4. Példa Kategóriák kategóriája
Ebben a példában definiálunk egy kategóriát, **Cat**-ot, aminek a kis kategóriák az objektumai, és élei a kategóriák közötti funktorok. Legyen egy **Cat**-beli $\mathcal{C}$ objektumra $id_{\mathcal{C}}$ az $id : \mathcal{C} \to \mathcal{C}$ identitás funktor. Funktorok kompozíciója pedig legyen a gráf homomorfizmusként vett kompozíciójuk, azaz ha $F : \mathcal{C} \to \mathcal{D}$ és $G : \mathcal{D} \to \mathcal{E}$, akkor legyen $G \circ F(C) = G(F(C))$ minden $\mathcal{C}$-beli C objektumra, és $G \circ F(f) = G(F(f))$ minden $\mathcal{C}$-beli f élre. Egyszerűen belátható, hogy ez a kompozíció funktorokból funktorokat képez, és az így definiált **Cat** valóban kategória.

2.3.2. Diagramok

Nem térünk ki a diagramok formális definíciójára, ezzel kapcsolatban [1]-re utalunk, de használni fogjuk a diagram érvényességének fogalmát. Lényegében arról van szó, hogy felrajzolunk egy gráfot, és azt mondjuk, hogy az így kapott "diagram" érvényes, ha akárhogy választunk benne két objektumot, a köztük levő irányított utak egyenlők. Tipikus példa a négyzetes diagram, ehhez vegyük a **Set** kategóriát és e, f, g, h függvényeket:

$$\begin{tikzcd} A \arrow[r, "e"] \arrow[d, "g"'] & B \arrow[d, "f"] \\ C \arrow[r, "h"'] & D \end{tikzcd}$$

Ez a diagram akkor érvényes, ha $f \circ e = h \circ g$.

2.3.3. Bifunktorok

A funkcionális nyelvek típusainak modellezésénél szükségünk lesz a bináris funktorok (bifunktorok) fogalmára. Ez tulajdonképpen nem más, mint a funktorok egy speciális esete, de felfogható kétparaméteres funktorként is.

2.3.5. Definíció Legyen $\mathcal{C}$ egy kategória. Ekkor az $F : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ leképezést **bifunktornak** nevezzük, ha:

BF-1: minden f és g $\mathcal{C}$-beli él esetén
$F(f, g) : F(kezd(f), kezd(g)) \to F(veg(f), veg(g))$ $\mathcal{C}$-nek egy éle;

BF-2: F megőrzi a kompozíciót, azaz $F(f \circ g, h \circ k) = F(f, h) \circ F(g, k)$, ha $f \circ g$ és $h \circ k$ értelmezett.

BF-3: F megőrzi az identitás éleket: $F(id_A, id_B) = id_{F(A,B)}$, ha A és B $\mathcal{C}_0$ elemei.

2.3.6. Megjegyzés A feltételek pontosan azt jelentik, hogy F egy funktor a $\mathcal{C} \times \mathcal{C}$ és $\mathcal{C}$ kategóriák között, de fontossága miatt külön nevet adtak neki.

Két egyszerű kapcsolatra mutat rá az alábbi állítás.

2.3.7. Állítás Ha $\oplus$ egy bifunktor valamilyen $\mathcal{C}$ kategória fölött, és $A \in \mathcal{C}_0$, akkor az

$$(A\oplus)B := A \oplus B$$

$$(A\oplus)f := id_A \oplus f$$

definícióval értelmezett $A\oplus$ leképezés $\mathcal{C}$ endofunktora. (Az állítás párja is érvényes, azaz a második argumentumot is rögzíthetnénk.)

Továbbá legyen adott két endofunktor F és G $\mathcal{C}$ fölött, valamint legyen $\oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ egy bifunktor, ekkor az

$$(F \oplus' G)A = (FA) \oplus (GA)$$

$$(F \oplus' G)f = (Ff) \oplus (Gf)$$

utasításokkal értelmezett $\oplus'$ $\mathcal{C}$ endofunktora. Ezt a tulajdonságot az angol irodalom *lifting*-nek nevezi.

Bizonyítás

A bizonyítás a definíciók egyszerű következménye, itt most nem foglalkozunk vele.

Mielőtt a típusok modellezésére rátérnénk, nézzünk meg két, a továbbiakban fontos szerepet játszó, bifunktort.

2.3.8. Példa $C(L)$-ben bevezettük a típusok direktszorzatait, és megadtuk az élekhez rendelt direktszorzatot is, ezzel tulajdonképpen egy $\times$-tel jelölt bifunktort adtunk meg, idézzük fel a definíciót. Ha A és B két típus, $f : A \to C$ és $g : B \to D$ két függvény, akkor:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

$$f \times g : A \times B \to C \times D$$

$$(f \times g)(a, b) = (f(a), g(b))$$

A BF-1 feltétel pontosan a definíció második sora. A BF-2 feltétel abból következik, hogy függvények direktszorzatát komponensenkénti végrehajtással értelmeztük, és a kompozíció $C(L)$-ben a függvények szokásos kompozíciója. Végül BF-3 azért teljesül, mert egy él pontosan akkor $A \times B$ identitás éle, ha komponensenként identitásként működik. Ezzel beláttuk, hogy

$$\times : C(L) \times C(L) \to C(L)$$

valóban bifunktor. (A jelölés azért lehet furcsa, mert a kategóriák direktszorzatait is $\times$-el jelöltük.)

2.3.9. Példa Vezessünk be egy másik bifunktort is, ez a direktszorzathoz hasonlóan szintén egy fontos típuskonstrukciós eszköz lesz. Vegyük hozzá $C(L)$ objektumaihoz a következő "diszjunkt unió" objektumokat:

$$A + B = \{inl(a)|a \in A\} \cup \{inr(b)|b \in B\}$$

Ahol inl és inr függvények arra szolgálnak, hogy elkülöníthessük egymástól a két halmaz elemeit. Definiálhatjuk őket például így:

$$inl_{A+B}(a) = (0, a)$$

$$inr_{A+B}(b) = (1, b)$$

ahol $a \in A$, $b \in B$, 0 és 1 pedig a NAT típus két konstansa, de bármilyen két különböző konstans megteszi, sőt tulajdonképpen csak az az érdekes, hogy inl_{A+B} és inr_{A+B} értékkészlete diszjunkt legyen. A továbbiakban az $A + B$ alsóindexet elhagyjuk, mert a környezetből mindig világos lesz, hogy éppen melyik diszjunkt unióhoz kapcsolódó inl, inr függvényről van szó.

Vezessünk be további éleket is. Legyen $f : A \to C$ és $g : B \to D$ és

$$(f + g) : A + B \to C + D$$

$$(f+g)(x) = \begin{cases} inl(f(a)), & \text{ha } x = inl(a) \text{ valamely } a \in A\text{-ra} \\ inr(g(b)), & \text{ha } x = inr(b) \text{ valamely } b \in B\text{-re} \end{cases}$$

Az így kapott

$$+ : C(L) \times C(L) \to C(L)$$

leképezés bifunktor, ami az előző példával analóg módon egyszerűen belátható. Végül vezessünk be egy-egy élet minden $C + C$ ($C \in C(L)_0$) objektumhoz:

$$un : C + C \to C$$

$$un(x) = \begin{cases} a, & \text{ha } x = inl(a) \text{ valamely } a \in C\text{-re} \\ b, & \text{ha } x = inr(b) \text{ valamely } b \in C\text{-re} \end{cases}$$

Ezzel egy $f : A \to C$ és egy $g : B \to C$ függvényhez az

$$f \ \triangledown \ g : A + B \to C$$

függvényt rendelhetjük a következőképpen:

$$f \ \triangledown \ g = un \circ (f + g)$$

A definíciónak a későbbiekben még nagy hasznát vesszük. Egyelőre annyit jegyezzünk meg róla, hogy a diszjunkt unió aktuális eme alapján kiválasztott függvény végrehajtása után nem $C + C$-ből veszük az eredményt, hanem elhagyjuk a direkt összeget és egyszerűen C-be képezünk (a két függvényt mintegy összefésülve).

2.3.4. Funktorok és típusok

Eddig elsősorban L műveleteivel foglalkoztunk, és azt mondtuk, hogy ezek a modellező $C(L)$ kategória élei. Tettünk bizonyos megfontolásokat $C(L)$ objektumairól is. Kikötöttük, hogy a kategóriának tartalmaznia kell egy kitüntetett objektumot, egy terminális elemet, aminek segítségével a nyelv konstansait definiáltuk. Ezt az elemet jelöltük 1-el. Most megvizsgáljuk, hogy még milyen feltételeknek kell ahhoz teljesülni, hogy a kategória összetett típusok modellezésére is alkalmas legyen.

Egyszerű típusok

Kezdjük a **konstans**okkal. Az alábbi példa Haskellben definiálja a Constant típust, aminek egyetlen típusértéke Const.

data `Constant = Const`

A gondolatmenet mindig az lesz, hogy azonosítjuk a kérdéses típus szerkezetét, azaz keresünk egy olyan objektumot, amivel izomorf az új típus. Fel kell tennünk azt, hogy a kategória elég sok egymással izomorf objektumot tartalmaz, ugyanis a típusokat a nevük segítségével azonosítjuk. Feltételezzük, hogy a nevek valamilyen rögzített ábécé fölötti véges hosszú sorozatok elemei, így a rendszer elég gazdag lesz típusokban, ha feltesszük, hogy minden objektumához tartalmaz megszámlálhatóan végtelen sok másik, ezzel izomorf objektumot. A továbbiakban ezzel nem is foglalkozunk, mert mindegy, hogy a sok egyforma objektum közül melyiket választjuk ki egy-egy típus bevezetésekor, a lényeges kérdés a típus belső szerkezete.

A konstansok esetén különösen egyszerű dolgunk van, hiszen nyilvánvaló, hogy az új típus izomorf a rögzített terminális elemünkkel, 1-el. Tehát a példában szereplő `Constant` típus egy tetszőleges ezzel izomorf objektum $C(L)$-ben. Mivel a kategória előre tartalmaz minden szóbajöhető élet az objektumai között, így tartalmaz egy élet 1-ből `Constant`-ba is. Másrészt, mivel mindkét objektum egyelemű halmaz, csak egyetlen ilyen él létezik, válasszuk ezt `Constant` adatkonstruktorának, `Const`-nak.

A második lépés a **felsorolási típus**ok bevezetése. Nézzünk egy Haskellben megírt példát:

data `Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun`

Feltettük, hogy a kategória tartalmazza az objektumainak diszjunkt unióit, amit a $+$ (bi)funktor segítségével határoztuk meg. A bifunktoroknál elmondottak általánosíthatók kettőnél több változós funktorokra is, tetszőleges n-re bevezethető az n-változós diszjunkt unió funktor, ehhez azonban fel kell tennünk $C(L)$-ről, hogy objektumai között szerepelnek az így létrehozható halmazok is.

Ha a hetedrendű diszjunkt unió funktort úgy alkalmazzuk, hogy minden argumentumának az 1 objektumot választjuk, nyilvánvaló, hogy a keresett `Napok` típussal izomorf objektumot kapunk. Az 1 elemből a típusba hét különböző él vezet, ezek adják az új típus adatkonstruktorait.

Az unió típuskonstrukció

Az a feltételezés, hogy $C(L)$-ben definiálható az n-változós funktor, többek között azt jelenti, hogy $C(L)$ zárt az objektumok diszjunkt uniójának képzésére nézve, hiszen a funktor típusa

$$C(L)^n \to C(L)$$

Ebből következik, hogy a felsorolási típusok bevezetésénél jóval többet tettünk az előző lépésben, ugyanis gyakorlatilag bevezettük a diszjunkt unió típuskonstruktort. Az alábbi példa Haskellben szemlélteti a konstrukció használatát:

data `MaybeInt = Just Int | Nothing`

A Haskell programozók jól ismerik ezt a típust, pontosabban azt a változatát, amikor az `Int` helyén egy típusparaméter áll. Az eredeti változat egyfajta kivételkezelési lehetőséget ad, amit a későbbiekben használni is fogunk. Így típusparaméter nélkül meglehetősen redukált az alkalmazhatósági köre, de hamarosan lesznek eszközeink arra is, hogy a teljes változatot is leírhassuk.

Egyelőre azonban elégedjünk meg ezzel, ha megnézzük a szerkezetét, látható, hogy izomorf az

$$Int + 1$$

típussal, amiről a felsorolási típusok bevezetésekor feltettük, hogy ha az Int egy L-beli típus, akkor ez is eleme $C(L)_0$-nak.

A direktszorzat típuskonstrukció

A Haskell nyelvben a következő szintaxissal hozhatunk létre direktszorzatot:

```
data ComplexInt  = Cons Int Int
```

A `ComplexInt` típust komplex egészek tárolására használhatjuk.

A bifnuktorok tárgyalásánál bevezettük a $\times$ műveletet. Ehhez feltettük, hogy $C(L)$ tartalmazza a típusaiból készített rendezett párokat. Nyilvánvaló, hogy a keresett típus izomorf az alábbival:

$$Int \times Int$$

így $C(L)$-ben megadható a `ComplexInt` típus is. A $\times$ művelet is általánosítható n-változós esetre, ehhez csak azt a további feltételezést kell tennünk, hogy a típusok rendezett n-esei is $C(L)$ objektumai. Így többtényezős direktszorzatokkal is kibővíthetjük a kategóriát.

Rekurzív struktúrák

Ahhoz, hogy igazán hasznos típusokat definiálhassunk szükségünk van olyan eszközökre, amivel rekurzív struktúrákat írhatunk le. Ezért két lépésben erre is bemutatunk egy egyszerű módszert [13] nyomán.

Vegyük példaként az alábbi két Haskell-ben megírt típusdefiníciót:

```
data IntList    = Nil | Cons Int IntList
data List a     = Nil | Cons a (List a)
```

Mindkét definíció listákkal kapcsolatos. `IntList` elemei `Int` típusúak. `IntList`-et monomorfnak nevezzük ellentétben a másikkal, ahol a típuskonstruktornak van egy típusparamétere, ami megjelenik az adatkonstruktorokban is. `List` tulajdonképpen a tetszőleges elemű listák összegyűjtésére szolgál, és ha az "`a`" helyére `Int`-et írunk, `IntList`-el izomorf típust kapunk. A paraméterezhetőség miatt `List`-et polimorfnak nevezzük, és mivel csak egy paramétere van azt mondjuk, hogy ez egy unáris típuskonstruktor.

Monomorf adattípusok

Kezdjük a monomorf esettel, mert ez az egyszerűbb, és segítségünkre lesz a polimorf eset kapcsán is. Az eddigiekben bemutatott példák mindegyike monomorf típus volt, így az egyetlen újdonság a rekurzió jelenléte.

```
data IntList    = Nil | Cons Int IntList
```

`IntList` azért rekurzív, mert a `Cons` adatkonstruktor második paramétere `IntList`, így a típus definiálásához önmagát is felhasználjuk. Mielőtt azonban erre rátérnénk, első feladatunk, hogy az általános esetben előforduló típusdefiníciókat valamilyen közös alakra hozzuk. A probléma az, és ez vezet a megoldás kulcsához, a konstruktorok változatossága. A funkcionális nyelvek megengedik, hogy egyszerre több konstruktort használjunk, ezek lehetnek konstansok, vagy függvények is, sőt a függvényeknek eltérő lehet az aritása.

A konkrét példát tekintve az `IntList` konstruktorai a következők:

```
Nil :: IntList
Cons :: Int → (IntList → IntList)
```

Haladjunk visszafelé. A harmadik problémát úgy oldhatjuk meg, hogy a "currying" eljárást megfordítjuk, azaz a függvények argumentumait összecsomagoljuk egyetlen rendezett n-esbe. Például a fenti `Intlist`ben a `Cons` típuskonstruktorból a következő unáris konstruktort kapjuk:

```
Cons :: Int × IntList → IntList
```

A második probléma a konstans konstruktorokkal van. Ezeket olyan függvényekké alakítjuk, amiknek értelmezési tartománya az "1" típus. Például:

```
Nil :: 1 → IntList
```

Végül az első probléma a konstruktorok tetszőleges száma volt. Ennek kiküszöböléséhez a diszjunkt unió típust fogjuk felhasználni, és egyetlen konstruktorba fogjuk össze őket.

Nézzük a konkrét esetet. Most csak két konstruktorunk van, így a formálisan is definiált + bifunktort használhatjuk. Az eredmény típusával, azonban újabb probléma merül fel.

```
Nil + Cons :: 1 + (Int × IntList) → IntList + IntList
```

Ennek az az oka, hogy + egy bifunktor, így függvényekre csak úgy definiálhattuk, hogy az eredmény is egy diszjunkt unió legyen. A bifunktorok definíciójának első pontja ugyanis:

BF-1: minden f és g $\mathcal{C}$-beli él esetén
$F(f,g) : F(kezd(f), kezd(g)) \to F(veg(f), veg(g))$ $\mathcal{C}$-nek egy éle;

Szerencsére a + bevezetésekor definiáltuk a ∇ műveletet is:

$$f \ \nabla \ g = un \circ (f + g)$$

ahol:

$$un(x) = \begin{cases} a, & \text{ha } x = inl(a) \text{ valamely } a \in C\text{-re} \\ b, & \text{ha } x = inr(b) \text{ valamely } b \in C\text{-re} \end{cases}$$

Így + helyett a ∇ művelettel összekapcsolva a konstruktorokat az eredmény típusa:

```
Nil ∇ Cons :: 1 + (Int × IntList) → IntList
```

Végül egyetlen konstruktorunk maradt `Nil ∇ Cons`. Az értelmezési tartomány valamilyen típuskifejezés, ami tulajdonképpen egy funktornak az `IntList`-re való alkalmazásaként fogható fel. A funktor konkrétan:

$F_{IntList}$ `Y = 1 + (Int × Y)`

és

$$F_{IntList}\ \texttt{f}\ \texttt{=}\ id_1\ \texttt{+}\ \texttt{(}id_{Int}\ \times\ \texttt{f)}$$

Az általános esetben ugyanígy járhatunk el, csak be kell vezetnünk a ∇ művelet többváltozós megfelelőjét, de ez nem okoz különösebb problémát. Ezzel megoldottuk, hogy rekurzív monomorf adattípusok definíciójához egyetlen konstruktort használjunk. Ráadásul a konstruktor értelmezési tartománya egy típus képe valamilyen funktorra nézve. Ez a funktor úgynevezett polinomiális tulajdonsággal rendelkezik, ami azt jelenti, hogy a konstans funktor, a diszjunkt unió és a direktszorzat funktorok kombinációjával írható fel. A konstans funktor számunkra egy olyan a funktor, ami minden objektumot ugyanabba a típusba képez, az éleket pedig identitás függvényekbe. Belátható ([13], [14], [1]), hogy az ilyen funktoroknak van fixpontja $C(L)$-ben. Ez röviden azt jelenti, hogy létezik olyan T adattípus, amire teljesül, hogy FT izomorf T-vel. Ennek a formális definiálásához azonban be kell vezetnünk néhány további fogalmat az algebrákkal kapcsolatosan.

2.3.10. Definíció Legyen $\mathcal{C}$ egy kategória, és $F : \mathcal{C} \to \mathcal{C}$ endofunktor. Egy **F-algebra** egy olyan (A, f) pár, amiben $A \in \mathcal{C}$ és $f : FA \to A$. (Az elnevezés a szóbanforgó funktor nevéből származik, így például, ha a kérdéses funktort R-nek hívják, akkor R-algebrákról beszélünk.)

2.3.11. Definíció A $\mathcal{C}$ kategória $h : A \to B$ élét **F-algebrák közötti homomorfizmus**nak nevezzük, ha F $\mathcal{C}$-nek egy endofunktora, (A, f) és (B, g) két F-algebra, továbbá

$$h \circ f = g \circ Fh$$

diagrammal:

$$\begin{tikzcd}
FA \arrow[r, "f"] \arrow[d, "Fh"'] & A \arrow[d, "h"] \\
FB \arrow[r, "g"'] & B
\end{tikzcd}$$

Egyszerűen belátható, hogy az F-algebrák és a köztük definiált homomorfizmusok kategóriát alkotnak. Ezt (F : $\mathcal{C}$)-vel jelöljük.

2.3.12. Definíció (F : $\mathcal{C}$) egy (A, a) objektuma **F fixpont**ja, ha "a" egy izomorfizmus F A és A között.

2.3.13. Definíció Az $F : \mathcal{C} \to \mathcal{C}$ funktor legkisebb fixpontja $(F : \mathcal{C})$ egy kezdeti elme.

Természetesen ahhoz, hogy ez a definíció értelmes legyen szükséges, hogy a kezdeti elemekhez tartozó élek egyúttal izomorfizmusok is legyenek, de szerencsére ez belátható, és bizonyítása megtalálható [1]-ben.

Az új fogalmakkal, most már pontosan megadhatjuk mit értünk az F polinomiális funktor által generált T adattípus alatt. Ehhez egész egyszerűen meg kell keresnünk a $F_{IntList}$-algebrák kategóriájának kezdeti elemét, ami az izomorfia erejéig egyértelmű (hiszen a kezdeti elemek egymással mindig izomorfak). Az így definiált $(T, in_T) \in (F_{IntList} : C(L))$ elem első tagját tekintjük a T adattípusnak. A továbbiakban ezt egyszerűen így fogjuk jelölni:

```
T = DATA F
```

Azaz konkrétan:

`IntList = DATA` $F_{IntList}$

Adósak vagyunk még az $in_T : FT \to T$ él azonosításával. Erre igaz, hogy az F T és T közötti izomorfizmus egyik élével egyezik meg. Nézzük meg, mi lenne ez konkrétan $F_{IntList}$-ben:

$$in_{IntList} : F_{IntList} IntList \to IntList$$

$F_{IntList}$ definícióját felhasználva:

$$in_{IntList} : 1 + (Int, IntList) \to IntList$$

Korábban sem Nil-t, sem Cons-t nem definiáltuk, ugyanakkor mivel

```
Cons :: Int × IntList → IntList
Nil :: 1 → IntList
```

és

```
Nil ∇ Cons :: 1 + (Int × IntList) → IntList
```

ezért választhatjuk `Nil ∇ Cons`-et $in_{IntList}$-nek. Ezzel egy lépésben definiáljuk mindkét konstruktort, hiszen a művelet értelmezési tartománya egy diszjunkt unió, amiből visszakövetkeztethetünk a két függvény működésére az unió komponensein. $in_{IntList}$-et konstruktornak nevezik, aminek az az oka, hogy az az $F_{IntList}(IntList)$ és $IntList$ objektumok közti izomorfizmus egyik ágaként, bijektív megfeleltetést létesít $F_{IntList}$ elemei és $IntList$ elemei között. Ezt felhasználva lehetőségünk nyílik `Nil` és `Cons` segítségével felírni `IntList` elemeit, hiszen `Nil` is kiválaszt egy elemet, a `Cons` segítségével pedig felírhatjuk a többit. A bijektív megfeleltetés garantálja, hogy valóban minden elemet fel tudunk írni.

Most definiálhatjuk például egy IntList-beli lista elmeit összegző `sum :: IntList → Int` függvényt:

```
sum (Nil)        = 0
sum (Cons (a, x)= a + sum x
```

A funkcionális nyelvekből megismert `fold` függvénynek is adhatunk kategória-elméleti megfelelőt. Mivel $(IntList, Nil \nabla Cons)$ az $F_{IntList}$ funktor által meghatározott kezdeti elem, így a kezdeti elem definíciója miatt, minden más (T, a) algebrába létezik pontosan egy $h : (IntList, Nil \nabla Cons) \to (T, a)$ homomorfizmus.
Egyszerűen belátható, hogy $(Int, zero \nabla plus)$ egy másik $F_{IntList}$-algebra, ahol $zero : 1 \to Int$ az egészek nulla konstansát kijelölő függvény, és $plus : Int \times Int \to Int$ két egészhez azok összegét rendelő függvény.

Ekkor igaz az, hogy

$$sum \circ Nil \nabla Cons = zero \nabla plus \circ F_{IntList}(sum) \tag{2.4}$$

Ábrával:

$$\begin{tikzcd}
FIntList \arrow[r, "Nil \nabla Cons"] \arrow[d, "F_{IntList}(sum)"'] & IntList \arrow[d, "sum"] \\
FInt \arrow[r, "zero \nabla plus"'] & Int
\end{tikzcd}$$

Ennek belátáshoz vizsgáljuk meg, mit jelent az $F_{IntList}(sum)$ kifejezés. Mivel $F_{IntList}$ funktor így ez definiált, hiszen sum egy függvény. Az is nyilvánvaló, hogy $F_{IntList}(sum) : F_{IntList}(IntList) \to F_{IntList}(Int)$, abból, hogy $sum : IntList \to Int$.

Emlékeztetünk a diszjunkt unió definíciójára:

Legyen f és g $C(L)$ két éle, ekkor:

$$(f + g) : A + B \to C + D$$

$$(f+g)(x) = \begin{cases} inl(f(a)), & \text{ha } x = inl(a) \text{ valamely } a \in A\text{-ra} \\ inr(g(b)), & \text{ha } x = inr(b) \text{ valamely } b \in B\text{-re} \end{cases} \tag{2.5}$$

$F_{IntList}$-et sum-ra alkalmazva:

$$F_{IntList}(sum) = id_1 + (id_{Int}, sum)$$

$F_{IntList}(IntList)$ elemei alapján két eset lehetséges:

1. Vegyük az inl(1)-et, mint $F_{IntList}(IntList)$-beli elemet. Erre

$$F_{IntList}(sum)(inl(1)) = (id_1 + (id_{Int}, sum))(inl(1)) =$$

2.5 szerint

$$= inl(id_1(1)) =$$

id_1 alkalmazása:

$$= inl(1)$$

2. Ugyanakkor egy $(x, l) \in Int \times IntList$ elemre:

$$F_{IntList}(sum)(inr(x,l)) = (id_1 + (id_{Int}, sum))(inr(x,l)) =$$

2.5 szerint

$$= inr((id_{Int}, sum)(x,l)) =$$

A direktszorzat bifunktort függvényekre komponensenként definiáltuk, így az eredmény első tagja mindenképpen x lesz. Pontosabban az eredmény nem egy pár lesz, erre még alkalmaznunk kell az inr() függvényt is. A második komponens miatt két eset lehetséges sum definíciója alapján. Tehát az eredmény:

$$= inr((x,t))$$

ahol

$$t = \begin{cases} 0, & \text{ha l=Nil} \\ a + sum(as), & \text{ha l=Cons(a, as)} \end{cases}$$

2.4 jobboldalát majdnem ki is számoltuk, már csak annyi van hátra, hogy a fenti két eset végeredményére $zero \nabla plus$-t alkalmazzuk. Az most már egyszerűen látható, hogy

$$zero \nabla plus(l) = \begin{cases} 0, & \text{ha l=inl(1)} \\ x + t, & \text{ha l=inr((x,t))} \end{cases}$$

Összességében elmondhatjuk, hogy a jobboldal egy olyan $F_{IntList} \rightarrow Int$ függvény, ami az $F_{IntList}$-beli listákhoz az elemek összegét rendeli. Annak belátását, hogy 2.4 baloldala is ugyanígy működik az olvasóra hagyjuk, a jobboldalt is csak azért írtuk ki részletesen, hogy a struktúrákkal való számolást szemléltessük.

2.4 egyenlőség jelentősége, hogy a sum függvényt a két algebra, $(IntList, Nil \nabla Cons)$ és $(Int, Zero \nabla Plus)$ egyértelműen meghatározza, ugyanis $(IntList, Nil \nabla Cons)$ egy kezdeti elem, így minden más algebrába pontosan egy olyan él vezet, ami a diagramot érvényessé teszi. Ehhez hasonlóan minden más F-algebrához, és ennek f éléhez egyértelműen létezik a fenti sum-hoz hasonló függvény, ami IntList-ről a kérdéses típusba képez. Ez indokolja, hogy bevezethetjük a $fold_{IntList}$ operátort, ami tehát egy algebra f éléhez azt az egyértelműen létező függvényt rendeli, amire teljesül, hogy

$$\begin{tikzcd} FIntList \arrow[r, "Nil \nabla Cons"] \arrow[d, "F(fold_{IntList} f)"'] & IntList \arrow[d, "fold_{IntList} f"] \\ FB \arrow[r, "f"'] & B \end{tikzcd}$$

Megjegyezzük, hogy a Haskell fejlesztői megírták a listákra vonatkozó fold operátort (illetve különböző változatait), és ezek segítségével definiálják például listák elemeinek összegét is.

Polimorf adattípusok

Az előző alfejezetben bemutatott IntList adattípusnak még mindig van egy nagy hátránya, hiszen a lista elemeinek típusa rögzített. Szeretnénk olyan típusokat is definiálni, ahol az alaptípus is paraméter lehetne, például "`a`" típusú fákat, listákat és még sorolhatnánk.

Nézzük a következő típusdefiníciót:

```
data List a      = Nil | Cons a (List a)
```

Ebben az esetben `List` egy típuskonstruktor. A polimorf adattípusok formalizálásának alapgondolata, hogy bevezetünk egy bifunktort (jelöljük $\oplus$-tel), ezután rögzítjük az egyik paraméterét, végül vesszük az így kapott funktor fixpontját. Így tetszőleges alaptípusú konstrukciókat hozhatunk létre, amit a továbbiakban így fogunk jelölni:

$$TA = DATA(A\oplus)$$

Az ehhez kapcsolódó konstruktor típusa:

$$in_{TA} : A \oplus TA \to TA$$

Például a polimorf lista típust a következőképpen adhatjuk meg:

$$List A = DATA(A\oplus)$$

ahol:

$$A \oplus\ B = 1 + (A \times B)$$

A monomorf adattípusoknál bemutatott $fold$ operátor egyszerűen általánosítható erre az esetre, hiszen a

$$TA = DATA(A\oplus)$$

utasítás minden A típushoz hozzárendel egy (TA, in_{TA}) kezdeti algebrát, amire megint teljesül, hogy minden más T-algebrába pontosan egy homomorfizmus vezet. Ezzel definiálhatjuk tetszőleges $(A \oplus B, f)$ algebrához a $fold_{TA}\ f$ élet, a továbbiakban elhagyjuk az A paramétert in_{TA}-ból és $fold_{TA}$-ból, ezzel túlterhelve a jelöléseket. A kapott éllel mindenesetre érvényes az alábbi diagram:

$$\begin{tikzcd}
A \oplus TA \arrow[r, "in_T"] \arrow[d, "id \oplus fold_T f"'] & TA \arrow[d, "fold_T f"] \\
A \oplus B \arrow[r, "f"'] & B
\end{tikzcd}$$

A $TA = DATA(A\oplus)$ típusdefiníció minden `A` típushoz rendel egy másik típust. Felmerül a kérdés, hogy nem lehetne-e valahogy úgy általánosítani, hogy C(L)-nek egy endofunktora legyen. A válasz igen, és ezzel azt kapjuk, hogy az egyparaméteres típuskonstruktorok a kategória funktoraival modellezhetők.

A kiterjesztéshez `T`-t értelmeznünk kell a kategória f éleire. Vegyük például az $f : A \to B$ élet, és legyen $Tf = fold_{TA}(in_{TB} \circ (f \oplus id))$.

Ábrával:

$$\begin{tikzcd}
A \oplus TA \arrow[rr, "in_{TA}"] \arrow[d, "id \oplus Tf"'] & & TA \arrow[d, "Tf"] \\
A \oplus TB \arrow[r, "f \oplus id"'] & B \oplus TB \arrow[r, "in_{TB}"'] & TB
\end{tikzcd} \tag{2.6}$$

Az ábrából talán jobban látható, hogy a definíció értelmes. Annak igazolására, hogy az így definiált T valóban funktor tekintsük a következő diagramot:

$$\begin{tikzcd}
A \oplus TA \arrow[rr, "in_{TA}"] \arrow[d, "id \oplus Tid"'] & & TA \arrow[d, "Tid"] \\
A \oplus TA \arrow[r, "id \oplus id"'] & A \oplus TA \arrow[r, "in_{TA}"'] & TA
\end{tikzcd}$$

A bifunktorok harmadik tulajdonsága, hogy a megőrzik az identitás éleket, azaz: $F(id_A, id_B) = id_{F(A,B)}$. Ezalapján, felhasználva az identitás élek kompozícióval kapcsolatos tulajdonságát, a bal alsó él elhagyható:

$$\begin{tikzcd}
A \oplus TA \arrow[r, "in_{TA}"] \arrow[d, "id \oplus Tid"'] & TA \arrow[d, "Tid"] \\
A \oplus TA \arrow[r, "in_{TA}"'] & TA
\end{tikzcd}$$

Mivel a homorfizmus egyértelmű, és id eleget tesz a feltételeknek $Tid = id$, azaz T az identitás éleket megőrzi.
Még a kompozíció fennállását kell igazolnunk:

$$\begin{tikzcd}
A \oplus TA \arrow[rrr, "in_{TA}"] \arrow[d, "id \oplus Tf"'] & & & TA \arrow[d, "Tf"] \\
A \oplus TB \arrow[r, "f \oplus id"'] \arrow[d, "id \oplus Tg"'] & B \oplus TB \arrow[rr, "in_{TB}"'] \arrow[d, "id \oplus Tg"] & & TB \arrow[d, "Tg"] \\
A \oplus TC \arrow[r, "f \oplus id"'] \arrow[rr, bend right, "g \circ f \oplus id"'] & B \oplus TC \arrow[r, "g \oplus id"'] & C \oplus TC \arrow[r, "in_{Tc}"'] & TC
\end{tikzcd}$$

A bifunktorok kompozícióra vonatkozó BF-2-es tulajdonsága alapján érvényesek a párhuzamos élek a bal alsó sarokból indulva, a bal alsó négyszög pedig megint az identitás élekre vonatkozó megfontolások és a kompozíció miatt. Így érvényes a következő diagram is:

$$\begin{tikzcd} A \oplus TA \arrow[rr, "in_{TA}"] \arrow[d, "id \oplus Tg \circ Tf"'] & & TA \arrow[d, "Tg \circ Tf"] \\ A \oplus TC \arrow[r, "g \circ f \oplus id"'] & A \oplus TA \arrow[r, "in_{TC}"'] & TC \end{tikzcd}$$

Ismét a homomorfizmus egyértelműségére hivatkozva azt kapjuk, hogy $T(g \circ f) = Tg \circ Tf$, amivel bebizonyítottuk, hogy T valóban endofunktor.

A funkcionális programnyelvekből ismerjük a `map` függvényt, ami elemenkénti feldolgozást hajt végre egy lista elemein. Két paramétere van: egy függvény és maga a lista. A definíciója valahogy így nézne ki a mi jelöléseinkkel:

```
map                :: (A → B) → List A → List B
map f Nil = Nil
map f (Cons a as) = Cons (f a) (f as)
```

Írjuk fel a 2.6 diagramot is listákra:

$$\begin{tikzcd} 1 + A \times ListA \arrow[rr, "Nil\nabla Cons_{ListA}"] \arrow[d, "id + id \times Listf"'] & & ListA \arrow[d, "Listf"] \\ 1 + A \times ListB \arrow[r, "1 + f \times id"'] & 1 + B \times ListB \arrow[r, "Nil\nabla Cons_{ListB}"'] & ListB \end{tikzcd}$$

Ha végigszámolnánk a diagramot, beláthatnánk, hogy listáknál ez a bizonyos $Listf$ függvény nem más, mint `map f`! Ez szemléletesen elég nyilvánvaló, ugyanis a diagram bal oldali és alsó része megfeleltethető a fenti függvénydefiníciónak, míg a felső és jobboldali út gyakorlatilag a $Listf$ függvényt írja le (a felső élre csak a másik úttal való kapcsolat miatt van szükség). A diagram érvényességéből pedig a két út egyenlősége következik.

A Haskellben van egy Functor osztály. Ennek egyetlen függvénye az `fmap`, tulajdonképpen a `map` általánosítása polimorf típusokra. A függvény implementálásakor a következő feltételeknek kell eleget tennünk:

```
fmap id         = id
fmap f . g      = fmap f . fmap g
```

Ezek pontosan a funktorok függvényekkel kapcsolatos tulajdonságainak Haskell-ben felírt megfelelői, ebből megint a kezdeti algebrából kivezető élek egyértelműsége miatt következik, hogy $fmap$ egyértelműen definiált és tetszőleges egyparaméteres típuskonstruktorból képezett funktor függvényeken ezzel egyezik meg. Röviden tehát azt mondhatjuk, hogy az egyparaméteres típuskonstruktorok a nyelvben nem feltétlenül funktorok, hiszen nem működnek függvényeken, és a Functor osztály pontosan ennek a különbségnek az eltüntetésére szolgál.

2.4. Természetes transzformációk

2.4.1. Definíció Legyen $\mathcal{G}$ egy gráf és $\mathcal{C}$ egy kategória. Vegyünk köztük két gráf homomorfizmust: $D, E : \mathcal{G} \to \mathcal{C}$. Ezt megtehetjük, hiszen $\mathcal{C}$ tekinthető gráfként is. Egy $\alpha : D \to E$ **természetes transzformáció**t kapunk, ha definiáljuk az αa élcsaládot a következőképpen:

NT-1 legyen $\alpha_a : Da \to Ea$ egy él $\mathcal{C}$-ben $\mathcal{G}$ minden a pontjára;

NT-2 tetszőleges $\mathcal{G}$-beli $s : a \to b$ élre érvényes az alábbi diagram:

$$\begin{array}{ccc} Da & \xrightarrow{\alpha_a} & Ea \\ {\scriptstyle Ds}\downarrow & & \downarrow{\scriptstyle Es} \\ Db & \xrightarrow[\alpha_b]{} & Eb \end{array}$$

Definiáljuk természetes transzformációk kompozícióját is.

2.4.2. Definíció Legyen $\mathcal{G}$ egy gráf és $\mathcal{C}$ egy kategória. Legyen $D, E, F : \mathcal{G} \to \mathcal{C}$ három gráf homomorfizmus. Legyen $\alpha : D \to E$ és $\beta : E \to F$ két természetes transzformáció. Definiáljuk a $\beta \circ \alpha : D \to F$ kompozíciót a következőképpen: minden $\mathcal{G}$-beli "a" pontra: $(\beta \circ \alpha)_a = \beta_a \circ \alpha_a$.

Egyszerűen belátható, hogy a kompozíció eredménye is természetes transzformáció, csak azt kell megmutatni, hogy az alábbi diagram külső téglalapja érvényes:

$$\begin{array}{ccccc} Da & \xrightarrow{\alpha_a} & Ea & \xrightarrow{\beta_a} & Fa \\ {\scriptstyle Ds}\downarrow & & \downarrow{\scriptstyle Es} & & \downarrow{\scriptstyle Fs} \\ Db & \xrightarrow[\alpha_b]{} & Eb & \xrightarrow[\beta_b]{} & Fb \end{array}$$

Ami nyilvánvaló a két négyzet érvényességéből (a kompozíció asszociativitása miatt).

Még könnyebb meggondolni, hogy minden $D : \mathcal{G} \to \mathcal{C}$ gráf homomorfizmusnak létezik egy identikus természetes transzformációja önmagára, ezt (id_D)-vel jelöljük, és úgy definiáljuk, hogy az "a" ponthoz a Da-hoz tartozó identitás élet rendeljük: $(id_D)_a = id_{Da}$.

$$\begin{array}{ccc} Da & \xrightarrow{(id_D)_a = id_{Da}} & Da \\ {\scriptstyle Ds}\downarrow & & \downarrow{\scriptstyle Ds} \\ Db & \xrightarrow[(id_D)_b = id_{Db}]{} & Db \end{array}$$

Ebből az is következik, hogy rögzített $\mathcal{G}$ gráf és $\mathcal{C}$ kategória esetén a gráf homomorfizmusok a természetes transzformációkkal kategóriát alkotnak, pontosabban egy olyan kategóriát, aminek objektumai a $\mathcal{G} \to \mathcal{C}$ gráf homomorfizmusok, élei pedig a köztük levő természetes transzformációk. A gráfoknak kategóriákra történő homomorf leképezését modellnek is nevezik, ez indokolja, hogy az így kapott kategóriát $Mod(\mathcal{G}, \mathcal{C})$-vel jelölik.

2.4.3. Példa Vegyünk két gráfot $\mathcal{G}$-t és $\mathcal{H}$-t. Legyen $\phi = (\phi_0, \phi_1)$ egy $\mathcal{G} \to \mathcal{H}$ gráf homomorfizmus. A gráf homomorfizmus definíciója alapján $\mathcal{G}$ minden $u : m \to n$ éléhez létezik egy $\phi_1(u) : \phi_0(m) \to \phi_0(n)$ él H-ban, ezt úgy is fogalmazhatjuk, hogy érvényesek az alábbi diagramok:

$$\begin{tikzcd} G_1 \arrow[r, "\phi_1"] \arrow[d, "kezd"'] & H_1 \arrow[d, "kezd"] \\ G_0 \arrow[r, "\phi_0"'] & H_0 \end{tikzcd} \qquad \begin{tikzcd} G_1 \arrow[r, "\phi_1"] \arrow[d, "veg"'] & H_1 \arrow[d, "veg"] \\ G_0 \arrow[r, "\phi_0"'] & H_0 \end{tikzcd}$$

Az alábbi gráfot a gráfok gráfjának nevezik abból kiindulva, hogy tetszőleges $\mathcal{G}$ gráfot definiálhatunk, ha g_1 helyébe az éleket, g_0 helyébe a csúcsokat írjuk, és definiáljuk a $kezd()$ és $veg()$ leképezéseket.

$$\begin{tikzcd} g_1 \arrow[r, shift left, "kezd"] \arrow[r, shift right, "veg"'] & g_0 \end{tikzcd}$$

Egy $\mathcal{C}$ kategóriát rögzítve igaz az, hogy egy gráf homomorfizmus tulajdonképpen a fenti gráf két modellje közti természetes transzformáció. A felső két diagram a transzformáció két példánya a gráfok gráfjában található két csúcsra. A részletek kidolgozása legyen az olvasó feladata.

2.4.4. Definíció Legyen $\mathcal{C}$ és $\mathcal{D}$ két kategória, valamint legyen $E, F : \mathcal{C} \to \mathcal{D}$ két funktor. A funktorokat gráf homomorfizmusnak tekintve a köztük értelmezett természetes transzformációkat a **funktorok természetes transzformációi**nak nevezzük.

Ha $\mathcal{C}$ és $\mathcal{D}$ két kategória, akkor $\mathbf{Fun}(\mathcal{C}, \mathcal{D})$-vel jelöljük azt a kategóriát, aminek objektumai az $E : \mathcal{C} \to \mathcal{D}$ funktorok, élei pedig a köztük értelmezett természetes transzformációk. $\mathbf{Fun}(\mathcal{C}, \mathcal{D})$ valóban kategória, ami a modellek kategóriájánál ismertetekkel analóg módon belátható.

2.4.5. Példa Természetes transzformációk a modellben
Ebben a példában $C(L)$ és a természetes transzformációk kapcsolatát vizsgáljuk. A típusokról szóló részben az egyparaméteres típuskonstruktorokat $C(L)$ endofunktoraival azonosítottuk. Vegyük az ott ismertetett `List` típuskonstruktort és definiáljuk a listák végét visszaadó függvényt:

```
tail            :: List a → List a
tail Nil = Nil
tail (Cons a as) = as
```

Ez egy úgynevezett polimorf függvény, tetszőleges alaptípusú listára működik, hiszen nem használunk ki semmi speciálisat "`a`"-ról. Ennek megfelelően a függvényhez $C(L)$-ben egyszerre több él is tartozik, pontosabban minden lista típushoz egy-egy önmagába vezető függvény, tehát `tail` gyakorlatilag függvények egy családjaként képzelhető el, hasonlóan a természetes transzformációkhoz. Keressünk `tail`-hez egy $tail : List \to List$ természetes transzformációt! Az élek családját a `tail` élcsalád definiálja. Ahhoz, hogy az így kapott $tail$ természetes legyen minden "a" és "b" $C(L)$-beli típusra és $f : a \to b$ függvényre érvényesnek kell lennie az alábbi diagramnak:

$$\begin{tikzcd} List\ a \arrow[r, "tail_a"] \arrow[d, "List\ f"'] & List\ a \arrow[d, "List\ f"] \\ List\ b \arrow[r, "tail_b"'] & List\ b \end{tikzcd}$$

Itt $tail_a$ és $tail_b$ a megfelelő típushoz tartozó $tail$ függvényt jelöli. A listáknál említettük, hogy a típuskonstruktorhoz kapcsolódó funktor függvényeken a map leképezésnek felel meg, így:

$$\begin{tikzcd} List\ a \arrow[r, "tail_a"] \arrow[d, "map\ f"'] & List\ a \arrow[d, "map\ f"] \\ List\ b \arrow[r, "tail_b"'] & List\ b \end{tikzcd}$$

Vagyis a feltétel:

$$tail_b \circ map\ f = map\ f \circ tail_a$$

Szövegesen megfogalmazva annak kell teljesülnie, hogy ha egy a típusú listára előbb elemenként végrehajtjuk az f függvényt, majd vesszük a lista végét, ugyanazt kapjuk mintha a lista végére alkalmaztuk volna elemenként f-et. Ez nyilván teljesül, így $tail$ egy természetes transzformáció.

A sikeren felbuzdulva vizsgáljuk meg általánosan a polimorf függvények és természetes transzformációk kapcsolatát a modellben. Egyik irányban a helyzet egyszerű: egy természetes transzformáció nem más mint bizonyos feltételeket kielégítő függvények egy családja. A két feltétel közül számunkra most csak az első a fontos, ugyanis ez biztosítja, hogy minden természetes transzformáció pontosan olyan éleket tartalmaz, amik egy-egy polimorf függvényt jellemeznek, tehát minden természetes transzformációhoz rendelhető egy-egy ilyen függvény.

Fordítva is sok működő példát lehet találni, az előbb bemutatott $tail$ mellett a listák megfordítását végző $reverse$ függvénynek vagy akár a fák mélységi bejárását listákba író függvénynek is megfelel egy természetes transzformáció. Ha megvizsgáljuk ezeket a függvényeket, a funkcionális nyelv szempontjából megközelítve azt mondhatjuk, hogy mivel a polimorf függvény nem használ fel semmi információt az alaptípusáról, nem tételezi fel, hogy annak vannak bizonyos műveletei, így nem csinálhat mást, mint átalakíthatja a bemenet szerkezetét, a benne foglalt adatokat elmozdíthatja, elhagyhatja, de nem módosíthatja és nem hozhat be újabb adatokat sem. Ahhoz, hogy egy ilyen függvényhez természetes transzformációt rendelhessünk csak azt kell leellenőrizni, hogy felcserélhető-e a transzformáció alkalmazása egy elemenként feldolgozó függvény alkalmazásával.

Szerencsénk van, ugyanis az elemenként feldolgozó függvények pontosan a polimorf függvények ellentétei, hiszen nem használhatnak ki semmilyen strukturális információt a bemenetről, csak az adatot transzformálhatják át. Ebből nyilvánvaló, hogy mindegy, hogy először a szerkezetet alakítjuk át és utána transzformáljuk az adatokat, vagy fordítva: az adatok transzformációja után alakítjuk át a szerkezetet. Összefoglalva: igaz az állítás, a polimorf függvényeknek is megfeleltethető egy-egy természetes transzformáció.

Sajnos a helyzet nem annyira jó mint látszik, ugyanis a polimorf függvényeket sokkal tágabb osztályon szokás definiálni. Valójában polimorf függvényeknek nevezik azokat a függvényeket is, amik különböző feltételezésekkel élnek az alaptípus műveleteiről. Például polimorf az a listákon működő függvény is, amelyik képes egy lista elemeit rendezni, ha az adott elemtípuson definiált a rendezés reláció. Az ilyen függvények azonban már a természetes transzformációk első feltételének sem tesznek eleget: nincsen él minden szükséges objektumpár között, így nem rendelhetünk hozzájuk természetes transzformációkat. Szerencsére a továbbiakban nekünk elég az állítás első fele, csak azt fogjuk kihasználni, hogy a természetes transzformációknak megfeleltethetők polimorf függvények.

A természetes transzformációk lehetőséget adnak arra, hogy definiáljuk funktorokkal vett kompozíciójukat is. Legyen $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ három kategória, $F, G : \mathcal{A} \to \mathcal{B}$ és $H, K : \mathcal{B} \to \mathcal{C}$ négy funktor, $\alpha : F \to G$ és $\beta : H \to K$ pedig két természetes transzformáció. A helyzetet jól szemlélteti az alábbi diagram:

$$\begin{tikzcd} \mathcal{A} \arrow[r, shift left=2, "F"] \arrow[r, shift right=2, "G"'] \arrow[r, phantom, "\Downarrow\alpha"] & \mathcal{B} \arrow[r, shift left=2, "H"] \arrow[r, shift right=2, "K"'] \arrow[r, phantom, "\Downarrow\beta"] & \mathcal{C} \end{tikzcd}$$

2.4.6. Definíció A $\beta F : H \circ F \to K \circ F$ természetes transzformációt definiáljuk a következőképpen: minden "A" $\mathcal{A}$-beli objektumra legyen: $(\beta F)_A = \beta_{FA}$, azaz a β természetes transzformáció FA-beli komponense.

Továbbá bevezetjük a $H\alpha : H \circ F \to H \circ G$ természetes transzformációt is úgy, hogy legyen egy "A" $\mathcal{A}$-beli objektumra $(H\alpha)_A = H(\alpha_A)$, ami nem más, mint H alkalmazása az α_A élre.

A definíciók értelmesek, hiszen $(\beta F)_A$ valóban egy $H(F(A)) \to K(F(A))$, $(H\alpha)_A$ pedig egy $H(F(A)) \to H(G(A))$ él, de meg kell mutatnunk, hogy tényleg természetes transzformációkat kaptunk.

Először lássuk (βF)-et. Be kell látnunk, hogy egy tetszőleges $f : A \to A'$ $\mathcal{A}$-beli élre az alábbi diagram érvényes:

$$\begin{tikzcd} H \circ F(A) \arrow[r, "(\beta F)_A"] \arrow[d, "H \circ F(f)"'] & K \circ F(A) \arrow[d, "K \circ F(f)"] \\ H \circ F(A') \arrow[r, "(\beta F)_{A'}"'] & K \circ F(A') \end{tikzcd}$$

Ez szerencsére nem más, mint β természetességét leíró diagram az $F(f) : F(A) \to F(A')$ élre, ami természetesen érvényes.

$H\alpha$ vizsgálatához pedig azt kell megmutatnunk, hogy:

$$\begin{tikzcd} H \circ F(A) \arrow[r, "(H\alpha)_A"] \arrow[d, "H \circ F(f)"'] & H \circ G(A) \arrow[d, "H \circ G(f)"] \\ H \circ F(A') \arrow[r, "(H\alpha)_{A'}"'] & H \circ G(A') \end{tikzcd}$$

amit úgy kapunk, hogy α diagramjára alkalmazzuk H-t, és mivel H funktor, így ez is teljesül. Ezzel megmutattuk, hogy a definíciók helyesek.

2.4.7. Példa Nézzünk meg egy-egy példát a két új kompozíció szemléltetésére. $C(L)$ minden eddig bevezetett típusa felfogható valamilyen típusnak egy funktorra vett képeként. (Az alaptípusoknál ez a funktor az identitás funktor.) Így definiálhatjuk a következő természetes transzformációt:

$$\eta : id \to List$$

Tehát η valamilyen alaptípusról egy ilyen elemekből álló listába képező polimorf függvény, ami azt jelenti, hogy minden a típushoz tartalmaz egy $\eta_a : a \to List\ a$ élet. Nézzük a konkrét definíciót, elhagyva az a típusindexet:

$$\eta(x) = Cons\ x\ Nil$$

Tehát egy elemből η egy egyelemű listát képez, könnyen belátható hogy ez valóban természetes transzformáció, nézzük meg rajta a kétféle kompozíciót.

1.. A definíció szerint $(\eta List) : id \circ List \to List \circ List$, röviden $(\eta List) : List \to List^2$. Mivel $(\eta List)$ természetes transzformáció, így tetszőleges a, b típusra és $f : a \to b$ függvényre érvényes a következő diagram:

$$\begin{tikzcd} List\ a \arrow[r, "(\eta List)_a"] \arrow[d, "List\ f"'] & List^2\ a \arrow[d, "List^2\ f"] \\ List\ b \arrow[r, "(\eta List)_b"'] & List^2\ b \end{tikzcd}$$

A definíció szerint $(\eta List)_a = \eta_{List\ a}$, azaz az η természetes transzformáció $List\ a$-hoz tartozó éle, ami nem tesz mást, mint egy a-beli elemekből álló listát "bedobozol" egy másik lista belsejébe. Például:

$$((\eta List)_{Int})([1,2,3]) = (\eta_{List\ Int})([1,2,3]) = [[1,2,3]]$$

az áttekinthetőség kedvéért nem a $Cons$-Nil alakot használtuk a lista jelölésére, hanem az ezzel ekvivalens szögletes zárójeles formát. A példából is jól látszik, hogy az $\eta List$ természetes transzformációnak a listákat becsomagoló polimorf függvény felel meg.

2.. Nézzük a fordított irányú kompozíciót, ($List\ \eta$)-t. Ha megnézzük a definíciót, megint azt kapjuk, hogy ez egy $List \to List^2$ természetes transzformáció, amire érvényes, hogy tetszőleges a, b típus és $f : a \to b$ függvény esetén:

$$\begin{tikzcd} List\ a \arrow[r, "(List\ \eta)_a"] \arrow[d, "List\ f"'] & List^2\ a \arrow[d, "List^2\ f"] \\ List\ b \arrow[r, "(List\ \eta)_b"'] & List^2\ b \end{tikzcd}$$

Továbbá egy "a" típusra $(List\ \eta)_a = List(\eta_a)$, azaz az η természetes transzformáció a-hoz tartozó élére kell a $List$ funktort alkalmazni. A korábbiakból tudjuk, ilyenkor a map függvényt kell használni, tehát: $(List\ \eta)_a = map\ (\eta_a)$. Nézzünk itt is egy konkrét példát:

$$((List\ \eta)_{Int})([1,2,3]) = (map\ (\eta_{Int}))([1,2,3]) = [[1],[2],[3]]$$

mivel ($\eta\ Int$)-et a lista elemeire egyesével kell alkalmazni, a "bedobozolás" most elemenként történik.

2.5. Monádok, a triple fogalma

Most már elég eszközünk van arra, hogy a monádokat definiáljuk.

2.5.1. Definíció Egy **monád** vagy **triple** $\mathbf{T} = (T, \eta, \mu)$ olyan hármas, ahol $T : \mathcal{A} \to \mathcal{A}$ endofunktor valamilyen $\mathcal{A}$ kategória fölött. η és μ két természetes transzformáció: $\eta : id \to T$, $\mu : T^2 \to T$ úgy, hogy az alábbi két diagram érvényes:

$$\begin{tikzcd} T \arrow[r, "\eta T"] \arrow[dr, "="'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "="] \\ & T & \end{tikzcd} \qquad \begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"'] & T \end{tikzcd}$$

A hangsúly kedvéért =-vel jelöltük az identitás természetes transzformációt. Az η transzformációt a triple **egység**ének (unit), míg μ-t **szorzás**nak nevezik. A baloldali diagram a bal- és jobboldali identitást fejezi ki, a jobboldali pedig az asszociatív tulajdonságot.

2.5.2. Példa Az alábbi példa a monoidok és monádok kapcsolatára vonatkozik, az imént bevezetett elnevezéseket a monád természetes transzformációira a monoidok műveleteivel való analógiával indokoljuk.

Legyen M egy monoid, az ehhez tartozó reprezentációs monád $\mathbf{T} = (T, \eta, \mu)$ **Set**-en a következő: $T(S) = M \times S$ valamely S halmazra, egy $f : A \to B$ függvényre pedig értelemszerűen $T(f) = (id, f)$. $\eta_S : S \to T(S) = M \times S$, ami **Set**-ben egy $S \to M \times S$ függvényt jelent, egy $s \in S$ elemre legyen

$(1,s)$, ahol 1 a monoid egységelemét jelöli. Végül legyen $\mu_S(m_1,(m_2,s)) = (m_1m_2,s)$. Így a monád egység- és szorzás transzformációit a monoid megfelelő műveleti definiálják.

Az identitás és asszociativitás tulajdonságok a monoid műveleti tulajdonságai miatt érvényesek. A definíció első diagramjának baloldali ága azt jelenti, hogy az S halmazt rögzítve valamilyen (m,s) elemből kiindulva $(\eta T)_S$-el $(1,(m,s))$-be jutunk, innen μ_S-el vissza $(1m,s) = (m,s)$-be. A másik ágon (m,s)-re az η_S függvény T szerinti képét kell alkalmazni, tehát az eredmény (m, (1,s)), innen μ_S-el kapjuk, hogy $\mu_S((m,(1,s))) = (m1,s) = (m,s)$. Figyeljük meg, hogy valóban kihasználtuk, hogy 1 kétoldali egységelem.

A másik diagram baloldali ágán $(m_1,(m_2,(m_3,s)))$-ből indulva a μ természetes transzformáció $T(S)$-beli komponensével $(m_1m_2,(m_3,s))$-be, majd μ_S-el $((m_1m_2)m_3,s)$-be jutunk. A jobboldali ágon a μ_S függvény T szerinti képével $(m_1,(m_2,(m_3,s)))$-ből $(m_1,(m_2m_3,s))$-be, majd μ_S-el $(m_1(m_2m_3),s)$-be jutunk. A két ág egyenlő, hiszen M monoid, így a szorzás asszociatív, tehát: $((m_1m_2)m_3,s) = (m_1(m_2m_3),s)$.

2.5.3. Példa Bevezettünk egy η természetes transzformációt listákra is: a bedobozoló polimorf függvényt $C(L)$-ben. Definiáljunk egy $\mu : List^2 \to List$ természetes transzformációt is (a típusindex elhagyásával):

$$\mu(Nil) = Nil$$

$$\mu(Cons\ x\ xs) \ = \ x\ ++\ \mu\ (xs)$$

Itt ++ jelöli a listák konkatenáció műveletét. μ működése nagyon egyszerű: egy lista kilapítására szolgál. Pontosabban ha paramétere egy listákból álló lista, akkor a második réteget eltünteti úgy, hogy közben elemeit összefűzi. Például:

$$\mu_{Int}([[1,2,3],[4,5],[6]]) = [1,2,3,4,5,6]$$

Belátható, hogy μ valóban természetes transzformáció, és η-val analóg módon levezethető a *List* funktorral vett kompozíciója is. $(\mu\ List)$ és $(List\ \mu)$ is kétszeresen egymásbaágyazott listákból készít (egyszeresen) egymásbaágyazottakat. Nézzünk egy-egy példát működésükre:

$$(\mu\ List)_{Int}(\ [[[1,2,3],[4,5]],[[1,2]]]\ )\ =$$

η-hoz hasonlóan most is kívülről befelé haladunk:

$$=\ [[1,2,3],[4,5],[1,2]]$$

A másik kompozíció pedig:

$$(List\ \mu)_{Int}(\ [[[1,2,3],[4,5]],[[1,2]]])\ =$$

A *map* függvényre áttérve:

$$= map\ (\mu_{Int})(\ [[[1,2,3],[4,5]],[[1,2]]]\ )\ =\ [[1,2,3,4,5],[1,2]]$$

Igaz az állítás, hogy $(List, \eta, \mu)$ egy triple. Nézzük a definícióban szereplő diagramokat *List*-re alkalmazva:

$$\begin{tikzcd}
List \arrow[r, "(\eta\ List)"] \arrow[dr, "="'] & List^2 \arrow[d, "\mu"] & List \arrow[l, "(List\ \eta)"'] \arrow[dl, "="] \\
& List &
\end{tikzcd}$$

A baloldali ág bemenete egy lista, ezt bedobozoljuk (η $List$)-el egy egyelemű listába, majd μ-vel összeolvasztjuk a listák listáját, és visszakapjuk az eredeti listát, így ez az ág érvényes. A másik ágon is egy listából indulunk, de most elemenként kell dobozolni ($List$ η)-val, végül μ ezt is összeolvasztja, aminek az eredménye a kiindulási lista, így ez az ág is érvényes.

A másik feltétel:

$$\begin{tikzcd} List^3 \arrow[r, "(List\ \mu)"] \arrow[d, "(\mu\ List)"'] & List^2 \arrow[d, "\mu"] \\ List^2 \arrow[r, "\mu"'] & List \end{tikzcd}$$

A baloldali ágon az első lépés a μ természetes transzformáció $List$-beli komponense, így a belső listákból álló listát kapjuk eredményül (a középső réteg összeolvasztásával), ezután megint a belső listákat kell összeolvasztani μ-vel, ami teljesen kilapítja a kiindulási listát. A másik ágon fordítva indulunk el: a középső réteg minden listájára alkalmazzuk a μ-t, így egyesével olvasztjuk össze a belső szintet, majd a μ-t az így kapott listákból álló listára alkalmazva, megint az eredeti lista teljes kilapítását kapjuk, így a két ág egyenlő, azaz a diagram érvényes. (A gondolatmenet megértésében segít a két számpélda is.)

2.6. Monádok a modellben, Kleisli triple

A Haskell monádjai nem közvetlenül az előző alfejezetben ismertetett fogalom, hanem a Kleisli triple segítségével definiáltak. Kezdjük egy egyszerű jelöléssel.

2.6.1. Jelölés A $\mathcal{C}$ kategóriában a kompozíció jelölésére vezessük be egy másik formát is. Legyen $f : A \to B$, $g : B \to C$ a kategória két éle, ekkor $f; g = g \circ f$

Ezután definiáljuk a Kleisli triple-t a következőképpen:

2.6.2. Definíció **Kleisli triple**-nek nevezzük azt a $\mathcal{C}$ kategória fölötti $(T, \eta, \_^*)$ hármast, ahol $T : \mathcal{C}_0 \to \mathcal{C}_0$ leképezés, $\eta : id \to T$ természetes transzformáció és minden $f : A \to TB$ $\mathcal{C}$-beli él esetén $f^* : TA \to TB$ él $\mathcal{C}$-ben, továbbá teljesülnek az alábbi feltételek:

KT-1: $(\eta A)^* = id_{TA}$;

KT-2: $(\eta A); f^* = f$ minden $f : A \to TB$ él esetén;

KT-3: $f^*; g^* = (f; g^*)^*$, ha $f : A \to TB$ és $g : B \to TC$.

Felhívjuk a figyelmet arra, hogy a definícióban szereplő T nem funktor, hiszen csak a kategória objektumain van értelmezve.

Vizsgáljuk meg a kapcsolatot a két triple-fogalom között! Azt állítjuk, hogy minden triple- höz rendelhető egy Kleisli triple, sőt ez megfordítva is igaz, azaz bijekció létesíthető közöttük. Ennek igazolása viszonylag számolásigényes, mégis részletezzük a fogalmak megértése, elmélyítése érdekében.

1. Először adjuk meg a $\mathbf{T} = (T, \eta, \mu)$ monádhoz tartozó Kleisli triple-t. Ezt jelölje $\mathbf{T} = (T, \eta, \_^*)$. Válasszuk T-nek a monádban szereplő funktor megszorítását az objektumokra. Ezzel túlterheljük a jelölést, de ez nem okozhat zavart, hiszen az objektumokon ugyanúgy működnek, ha pedig az argumentum egy él, akkor a monádbeli T-ről van szó. η-t hagyjuk változatlanul és az $f : A \to TB$ élre legyen $f^* = T(f); \mu_B$.

Először lássuk be **KT-1**-et: $\_^*$ definíciójával, majd a monád jobboldali egység tulajdonságával:

$$(\eta_A)^* = T(\eta_A); \mu_{TA} = id_{TA}$$

Nézzük **KT-2**-t. $\_^*$ definíciójával és a kompozíció asszociativitásával:

$$\eta_A; f^* = \eta_A; (T(f); \mu_B) = (\eta_A; (T(f)); \mu_B =$$

innen η természetes tulajdonságát kihasználva, az asszociativitás és a monád baloldali egység tulajdonsága alapján:

$$= (f; \eta T_B); \mu_B = f; (\eta T_B; \mu_B) = f$$

Végül **KT-3**. $\_^*$ definícióját f-re felírva és T funktor kompozícióra vonatkozó tulajdonsága miatt:

$$(f; g^*)^* = T(f; g^*); \mu_C = T(f); (T(g^*); \mu_C) =$$

Ugyanezt még egyszer g-re is elvégezve:

$$= T(f); T(T(g); \mu_C); \mu_C = T(f); T^2(g); T(\mu_C); \mu_C =$$

A *monád* asszociatív tulajdonságával az utolsó két komponens "felcserélhető":

$$= T(f); T^2(g); \mu_{TC}; \mu_C =$$

μ természetes transzformáció tulajdonságát a középső két elemre alkalmazva kapjuk:

$$= T(f); \mu_B; T(g); \mu_C =$$

csoportosítva, végül $\_^*$ definícióját kétszer visszafelé alkalmazva:

$$= (T(f); \mu_B;)(T(g); \mu_C) = f^*; g^*$$

2. Most nézzük a másik irányt. Vegyük a $\mathbf{T} = (T, \eta, \_^*)$ Kleisli triple-t, és rendeljük hozzá a $\mathbf{T} = (T, \eta, \mu)$ monádot η-t megtartva, T következő kiterjesztésével: $T(f) = (f; \eta_B)^*$, ha $f : A \to B$ él és legyen $\mu_A = id^*_{TA}$. A definíciók értelmességének ellenőrzése, valamint az alapkövetelmények ellenőrzése (T funktor, μ természetes transzformáció) legyen az olvasó feladata. Csak a monád tulajdonságaival foglalkozunk.

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[dr, "="'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "="] \\
& T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

Először nézzük a baloldali egység tulajdonságot. T definíciójával kapjuk:

$$T(\eta_A);\mu_A = (\eta_A;\eta_{TA})^*;\mu_A =$$

μ definícióját és **KT-3** tulajdonságot (fordítva) kihasználva:

$$= ((\eta_A;\eta_{TA});id^*_{TA})^* =$$

átcsoportosítva, **KT-2**-t felhasználva:

$$= (\eta_A;(\eta_{TA};id^*_{TA}))^* = (\eta_A;id_{TA})^* =$$

az identitással való kompozíció miatt és **KT-1** miatt:

$$= \eta^*_A = id_{TA}$$

A jobboldali egység tulajdonság μ definíciójából és **KT-2**-ből következik:

$$\eta_{TA};\mu_A = \eta_{TA};id^*_{TA} = id_{TA}$$

Végül a monád asszociativitása. T definícióját felhasználva:

$$T(\mu_A)\mu_A = (\mu_A;\eta_{T^2A})^*;\mu_A = \tag{2.7}$$

μ definíciójával és **KT-3**-mal:

$$= (\mu_A;\eta_{TA})^*;id^*_{TA} = ((\mu_A;\eta_{TA});id^*_{TA})^* =$$

az asszociativitással átcsoportosítva

$$= (\mu_A;(\eta_{TA};id^*_{TA}))^* =$$

végül **KT-2**-vel és az identitással való kompozícióval:

$$= (\mu_A;id_{TA})^* = \mu^*_A$$

Másrészt μ definícióját kétszer alkalmazva:

$$\mu_{TA};\mu_A = id^*_{T^2A};id^*_{TA} = \tag{2.8}$$

KT-3 alapján, majd μ definícióját fordítva alkalmazva, végül az identitással vett kompozíciót kiszámítva:

$$(id_{T^2A};id^*_{TA})^* = (id_{T^2A};\mu_A)^* = \mu^*_A$$

Ebből 2.7 és 2.8 egyenlősége következik.

Most már megadhatjuk a kapcsolatot $C(L)$ kategória és a Haskell között ([8]). A monádok a Haskellben egy osztályt alkotnak a következő definícióval:

```
class Monad m where
(>>= )        :: m a → (a → m b) → m b
return        :: a → m a
(>> )         :: m a → m b → m b
fail          :: String → m a
m >> k        = m >>= λ x → k
```

Számunkra csak az első két függvény fontos, hiszen a harmadik (>>) az első segítségével (>>=) implementálva van, `fail` pedig egy "hétköznapi" polimorf függvény. Vegyünk viszont a modellben egy $(T, \eta, \_^*)$ Kleisli triple-t. A típusok alapján a T függvény megfeleltethető a `Monad` típuskonstruktornak, az $\eta : id \to T$ természetes transzformáció pedig a `return` polimorf függvénynek!

Az egyetlen látszólagos probléma $\_^*$ függvénnyel van, hiszen ennek típusa

$$\_^* : [A \to TA] \to [TA \to TB]$$

ugyanis függvényekhez függvényeket rendel. Ugyanakkor felfogható olyan kétváltozós függvényként is, ami egy $A \to TA$ függvényhez és TA paraméterhez egy TB értéket rendel. Ha megnézzük >>= típusát, akkor látható, hogy ott is valami hasonlóról van szó, csak a két paramétert fel kell cserélni. (Ennek csak az az oka, hogy kényelmesen lehessen használni.) Ezzel a Kleisli triple elemeit leképeztük a Haskell monád osztályára.

A második fejezetben több gyakorlati példát fogunk látni a monádok alkalmazási lehetőségére. Mivel ennek a fejezetnek a célja elsősorban a fogalom matematika bevezetése volt, így inkább a struktúra néhány öröklött tulajdonságával folytatjuk, ezek a tulajdonságok megtalálhatók szinte minden a Haskell monádjaival kapcsolatos leírásban és könyvben (pl. [16], ...), de ott nem szerepel bizonyításuk, hiszen többnyire nem a kategóriaelmélet fogalmain keresztül vezetik be a monádokat. A három állítás:

```
return x >>= k = k x
m >>= return = m
m >>= (λ x → k x >>= h) = (m >>= k) >>= h
```

A fent említett leírások az állításokat monadikus törvényeknek nevezik. Az első kettő az identitás, a harmadik az asszociativitás nevet kapta. A programozónak kell figyelnie arra, hogy a monád osztály példányosításakor a törvényeket betartsa, ugyanis a rendszer ezt nem ellenőrzi. Belátjuk, hogy "monadikus törvények" ekvivalensek a Kleisli triple definíciójában szereplő feltételekkel, azaz a `Monad` osztály példányai és a Kleisli triple-k között a fordított irányú kapcsolat is létezik, közöttük bijekció létesíthető. A fordított irányú konstrukció a kérdéses példányt definiáló típuskonstruktorból, a >>= művelet segítségével definiált $\_^*$ műveletből - $f^*(x) = (x >>= f)$ - és az $\eta = return$ megfeleltetésből áll.

A bizonyítás előtt lássuk a $\_^*$ művelet egy egyszerű átírását:

$$(f(x) >>= k) = k^*(f(x)) = (f; k^*)(x) \tag{2.9}$$

1.. Az első állítás baloldala a Kleisli triple jelöléseivel átfogalmazva (x típusa legyen A):

$$\eta_A(x) >>= k \overset{2.9}{=} (\eta_A; k^*)(x)$$

Ebből az

$$(\eta_A; k^*)(x) = k(x)$$

egyenlőség ekvivalens **KT-2**-vel.

2.. A második állítás baloldalának átfogalmazása:

$$m >>= \eta_A = \eta_A^*(m)$$

ebből látható, hogy a második állítás **KT-1**-el ekvivalens.

3.. A harmadik állítás jobboldalából

$$(m >>= k) >>= h =$$

$$= k^*(m) >>= h =$$

$$= h^*(k^*(m)) =$$

$$= (k^*; h^*)(m)$$

adódik a két >>= átírásával.
A baloldal:

$$m >>= (\lambda x \rightarrow kx >>= h) =$$

a külső >>= átírásával:

$$= (\lambda x \rightarrow kx >>= h)^*(m) =$$

majd a belső >>= átírásával (a h -nak megfelelő kifejezésben `x` nem szerepel az egyenlőség baloldala miatt):

$$= (\lambda x \rightarrow kx; h^*)^*(m) =$$

a λ-kifejezést elhagyva:

$$= (k; h^*)^*(m)$$

Ebből látható, hogy a két oldal egyenlősége **KT-3**-mal ekvivalens.

Ezzel beláttuk a kérdéses ekvivalenciát, így elmondhatjuk, hogy ha a `Monad` osztályt a programozó a monadikus törvényeket tiszteletben tartva példányosítja, akkor tulajdonképpen egy Kleisli triple-t definiál. Szokás még az alábbi negyedik tulajdonságot is monadikus törvénynek nevezni:

```
xs >>= return . f = fmap f xs
```

Ez a Kleisli triple tulajdonságaiból következik, így `fmap` tulajdonságait felhasználva a fenti három monadikus törvényből is levezethető.

4.. Az állítás bizonyításához szükség lesz a Kleisli triple és a monádok ekvivalenciájának bizonyításakor bevezetett konstrukcióra. Ott $\_^*$-ot így definiáltuk: $f^* = T(f); \mu_B$, ráadásul mivel T-t függvényeken a megfelelő $fmap$-pal azonosítottuk, így $T(f)$ helyett $fmap(f)$ is írható. Most nézzük az állítás baloldalának átfogalmazását:

$$xs >>= (f; \eta_{TB}) = (f; \eta_{TB})^*(xs) =$$

$\_^*$ funktoros definíciójával folytatva:

$$= (T(f; \eta_{TB}); \mu_{TB})(xs) =$$

mivel T funktor, így a kompozícióval kapcsolatos tulajdonsággal:

$$= (T(f); T(\eta_{TB}); \mu_{TB})(xs) =$$

Ebből η baloldali egység tulajdonságával:

$$= T(f)(xs) =$$

végül $T(f) = fmap(f)$ miatt:

$$= fmap(f)(xs)$$

amit bizonyítani kellett.

2.7. Kitekintés, az adjungált fogalma

Nem kapcsolódik szorosan a témához, de fontos matematikai kapcsolat van a monádok és az adjungált fogalom között. Így bár a továbblépéshez nincs szükségünk az itt leírtakra, mindenképpen említést érdemelnek. Az alfejezetet kitekintésnek szánjuk, így több helyen csak utalunk egyes állítások bizonyítására, további példákra.

2.7.1. Definíció Legyen $\mathcal{A}$ és $\mathcal{B}$ két kategória, továbbá $F : \mathcal{A} \to \mathcal{B}$ és $U : \mathcal{B} \to \mathcal{A}$ funktorok. Akkor mondjuk, hogy F bal adjungáltja U-nak és U jobb adjungáltja F-nek, ha létezik olyan $\eta : id \to U\ F$ természetes transzformáció, amire teljesül, hogy minden "A" $\mathcal{A}$-beli és "B" $\mathcal{B}$-beli objektumra, minden $f : A \to U\ B$ élre $\mathcal{A}$-ban egyértelműen létezik olyan $g : F\ A \to B$ él $\mathcal{B}$-ben, amire teljesül, hogy: $f = U\ g \circ \eta A$.
Ábrával:

$$\begin{tikzcd}
 & A \arrow[r, Rightarrow, "F"] \arrow[dl, "\eta A"'] \arrow[d, "f"] & FA \arrow[d, "g"] \\
o \arrow[r, "Ug"'] & U\ B & B \arrow[l, Rightarrow, "U"']
\end{tikzcd}$$

A definíció annak formális megfogalmazása, hogy létezik egy út az f és a g él közti konverzióra úgy, hogy g egyértelmű megoldása az $f = U(?) \circ \eta A$ egyenletnek.

Szokás a definícióban megfogalmazott tulajdonságra az $F \vdash U$ jelölést használni. Az (F, U, η) hármast **adjungált**nak (adjoint) nevezzük, és η az adjungált **unit**ja.

A definíció látszólag aszimmetrikus F-ben és U-ban, de igaz a következő állítás:

2.7.2. Lemma Legyen $F : \mathcal{A} \to \mathcal{B}$ és $U : \mathcal{B} \to \mathcal{A}$ két funktor úgy, hogy $F \vdash U$, ekkor létezik egy $\epsilon : FU \to id_B$ természetes transzformáció, amire teljesül, hogy minden $g : FA \to B$ élhez egyértelműen létezik egy $f : A \to UB$ él úgy, hogy: $g = \epsilon B \circ Ff$. ϵ-t az adjungált **counit**jának nevezzük.

Az állítás teljes bizonyítása megtalálható [1]-ben, itt terjedelmi okok miatt nem foglalkozunk vele.

[1] sok érdekes példát hoz adjungáltakra. Például adjungáltak segítségével definiálható a szabadság általános fogalma, azaz létezik olyan definíció, ami összefogja a szabad monoidokat, a gráf által generált szabad kategóriákat és más hasonló fogalmakat. Ehhez azonban be kellene vezetnünk a felejtő (forgetful) funktorok fogalmát is. Egy másik példa lehetne a direktszorzat funktor adjungáltja, de ehhez is további definíciókra lenne szükségünk.

Persze nem maradunk példa nélkül, mert jelenlegi fogalmainkkal is be tudjuk mutatni az inverz függvényhez kapcsolódó adjungáltakat. Ehhez vegyük a $C(\mathcal{P}(S), \subseteq)$ kategóriát, aminek objektumai az S halmaz részhalmazai, két objektum közötti él pedig azt jeleni, hogy az egyik halmaz része a másiknak, azaz $f : S_0 \to S_1$ pontosan akkor él a kategóriában, ha $S_0 \subseteq S_1$. (Ezt a kategóriát már láttuk a parciális rendezések kapcsán is.)
Válasszunk egy S és egy T halmazt, és vegyük a $C(\mathcal{P}(S), \subseteq)$ és $C(\mathcal{P}(T), \subseteq)$ kategóriákat.

Azután legyen $f : S \to T$ egy függvény, és tetszőleges $T_0 \subseteq T$ esetén legyen

$$f^{-1}(T_0) = \{s \in S | f(s) \in T_0\}$$

az inverz kép függvény szokásos kiterjesztése halmazokra. Nyilvánvalóan teljesül, hogy $T_0 \subseteq T_1$ esetén $f^{-1}(T_0) \subseteq f^{-1}(T_1)$, amiből az is belátható, hogy f^{-1} valójában egy funktor a két kategória között. Felmerül a kérdés, hogy vannak-e f^{-1}-nek adjungáltjai.

A válasz igen, f^{-1}-nek mindkét oldali adjungáltja létezik. Bal adjungáltja az a funktor, ami egy halmazhoz az f szerinti képét (direkt kép) rendeli:

$$f_*(S_0) = \{f(s) | s \in S_0\}$$

Tekintsük ugyanis az alábbi diagramot:

$$\begin{tikzcd}
 & A \arrow[r, Rightarrow, "f_*"] \arrow[dl] \arrow[d] & f_*A \arrow[d] \\
f^{-1}f_*A \arrow[r] & f^{-1}\ B & B \arrow[l, Rightarrow, "f^{-1}"']
\end{tikzcd}$$

A szóbanforgó kategóriák különlegesek abban az értelemben, hogy két halmaz között legfeljebb egy él vezethet, így a kérdéses $\eta : id \to f^{-1}f_*$ természetes transzformációt csak egyféleképpen definiálhatjuk (hiszen legfeljebb egy választásunk lehet az élekre). Ehhez csak azt kell belátnunk, hogy A és $f^{-1}f_*(A)$ között tényleg van él, ami azzal ekvivalens, hogy $A \subseteq f^{-1}f_*(A)$, ez pedig a függvények definíciója miatt teljesül.

Az így kapott η valóban természetes transzformáció, hiszen ha $G \subseteq H$, akkor $f^{-1}f_*(G) \subseteq f^{-1}f_*(H)$, diagrammal:

$$\begin{tikzcd}
G \arrow[r, "\eta G"] \arrow[d] & f^{-1}f_*G \arrow[d] \\
H \arrow[r, "\eta H"'] & f^{-1}f_*H
\end{tikzcd}$$

Ez pontosan a természetesség második feltétele.

Definiáltuk η-t, már csak az kell igazolni, hogy az első diagram bal alsó és függőleges éle ugyanakkor létezik azaz, hogy $A \subseteq f^{-1}(B)$ pontosan akkor teljesül, ha $f^{-1}f_*A \subseteq f^{-1}B$. Szerencsére ez is azonnal látszik a függvények definíciójából.

$f_!$-vel jelöljük az alábbi függvényt:

$$f_!(S_0) = \{t | f^{-1}(\{t\}) \subseteq S_0\}$$

Belátható, hogy ez is funktor, és az előzőhöz hasonló meggondolással kapjuk, hogy $f^{-1} \vdash f_!$.

2.7.1. Monádok és adjungáltak kapcsolata

Végül bizonyítás nélkül közlünk néhány a monádok és adjungáltak kapcsolatára vonatkozó állítást.

2.7.3. Lemma Legyen $U : \mathcal{B} \to \mathcal{A}$ és $F : \mathcal{A} \to \mathcal{B}$ két funktor és $F \vdash U$, az $\eta : id \to UF$ unittal és az $\epsilon : FU \to id$ counittal. Ekkor az $(UF, \eta, U\epsilon F)$ egy monád $\mathcal{A}$ fölött.

A bizonyítás a Godgement-szabályokon alapszik, ezek a funktorokkal és természetes transzformációkkal kapcsolatos kompozíciók tulajdonságait írják le. A részletek megtalálhatók [1]-ben.

A lemma mondanivalója, hogy minden adjungáltpár monádot definiál az $\mathcal{A}$ kategória fölött. Ennél több is igaz, ugyanis minden monádhoz létezik olyan adjungáltpár, amire a fenti konstrukció az adott monáddal egyezik meg.

Az egyik megközelítés ennek bizonyítására az **Eilenberg-Moore algebrákkal** kapcsolatos. Legyen $\mathbf{T} = (T, \eta, \mu)$ egy monád az $\mathcal{A}$ kategória fölött. Egy (A, a) T-algebrát $\mathbf{T}$-algebrának hívunk, ha érvényesek az alábbi diagramok:

$$\begin{tikzcd} T^2A \arrow[r, "\mu A"] \arrow[d, "Ta"'] & TA \arrow[d, "a"] \\ TA \arrow[r, "a"'] & A \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, "\eta A"] \arrow[dr, "="'] & TA \arrow[d, "a"] \\ & A \end{tikzcd}$$

$\mathbf{T}$-algebrák közti homomorfizmusok (élek) a T-algebrák közötti homomorfizmusokkal egyeznek meg. Ezekkel a definíciókkal a $\mathbf{T}$-algebrák kategóriát alkotnak, amit $\mathcal{A}^{\mathcal{T}}$-vel jelölünk.

Belátható, hogy $U : \mathcal{A}^{\mathcal{T}} \to \mathcal{A}$ az $U(A, a) = A$ és $Uf = f$ definíciókkal funktor, továbbá az $F : \mathcal{A} \to \mathcal{A}^{\mathcal{T}}$, amire $FA = (TA, \mu A)$ és $Ff = Tf$ szintén funktor, sőt igaz az is, hogy $F \vdash U$ és az ez által generált monád pontosan $\mathbf{T}$.

Az eredménnyel körülbelül egyidőben Kleislinek egy másik úton sikerült belátnia ugyanezt. Vezessük be a $\mathbf{T} = (T, \eta, \mu)$ $\mathcal{C}$ kategória fölötti monádhoz a $\mathcal{K} = \mathcal{K}(T)$ kategóriát a következőképpen. $\mathcal{K}$ objektumai legyenek $\mathcal{C}$ objektumai. Egy A és TB közötti él $\mathcal{C}$-ben legyen A és B közötti él $\mathcal{K}$-ban. Az $f : A \to TB$ és $g : B \to TC$ $\mathcal{C}$-beli éleknek megfelelő $\mathcal{K}$-beli élek kompozíciója legyen a következő $\mathcal{C}$-beli kompozícióhoz rendelt él:

$$A \xrightarrow{f} TB \xrightarrow{Tg} T^2C \xrightarrow{\mu C} TC$$

Az A-hoz tartozó identitás él legyen $\eta A : A \to TA$. Belátható, hogy $\mathcal{K}$ valóban kategória, sőt definiálható két funktor: $U : \mathcal{K} \to \mathcal{C}$ és $F : \mathcal{C} \to \mathcal{K}$ úgy, hogy $UA = TA$, $Uf = \mu B \circ Tf$ (ahol $f : A \to B$ $\mathcal{K}$-ban, azaz $f : A \to TB$ $\mathcal{C}$-ben. Továbbá $FA = A$ valamint $g : A \to B$-re $Fg = Tg \circ \eta A$. Ekkor érvényes, hogy $F \vdash U$, $T = U \circ F$, és a generált monád megint $\mathbf{T}$.

Terjedelmi okok miatt ezeket az állításokat nem bizonyítjuk. Ugyanakkor mindenképpen meg kellett említenünk őket, egyrészt az eredmények fontossága miatt, másrészt mert a Haskell monádokat a Kleisli triple segítségével definiáljuk, és $\mathcal{K}$ ezen keresztül is definiálható ([8]).

3. A monádok alkalmazási lehetőségei

Ebben a fejezetben gyakorlati módszerekkel vizsgáljuk a monádok és a funkcionális programozás kapcsolatát. Az alapgondolat Euego Moggitól származik, ő volt az első, aki a monádokat különböző számítások absztrakciójaként kezdte használni. A következő alfejezetekben több példán keresztül vizsgáljuk meg, hogy az egyes számítástípusok milyen eszközöket adnak a programozó kezébe.

Először bemutatjuk a Haskell két előre definiált monádját, azután megvizsgáljuk, hogyan lehet imperatív nyelvi elemeket építeni funkcionális programokba. Az itt bemutatott állapot monád egy másik alkalmazási lehetőségét vizsgáljuk második példánkban, amivel vektorok hatékony kezelését valósíthatjuk meg funkcionális nyelvben. Ezzel a fő probléma az, hogy ugyan az olvasás művelet különösebb gond nélkül implementálható, a vektorok elemeinek értékadásával problémák vannak: gondoljunk csak arra, hogy a tisztán funkcionális programnyelveknek nincsenek is értékadás műveletei.

Mivel szükség van arra, hogy a vektor elemeinek értéket adjunk, ezt úgy szimulálhatjuk, hogy a vektort használó függvények egymásnak adogatják át a vektort, mint paramétert, és ha valamit változtatni akarunk, akkor egy másik vektort adunk tovább. Vannak olyan esetek, amikor a vektor egyszeresen hivatkozott objektum. Ilyenkor megtehetjük, hogy módosításkor az egész vektor újraépítése helyett csak a kérdéses elemet változtatjuk, amivel futási időt takaríthatunk meg. Ennek azonban az a feltétele, hogy a vektor egyszeresen hivatkozott legyen, amit nem könnyű ellenőrizni, de a monádok segítségével elegáns megoldást adunk a problémára.

A harmadik nagyobb példánk a monádok és a konkurens folyamatok kapcsolatát vizsgálja, természetesen ezt is egy számítástípus, a folytatás, segítségével.

Végül röviden bemutatjuk a Haskell IO-t is. Nem célunk a részletes elemzés, csak néhány alapfüggvényt és egyszerűbb példát mutatunk, de igyekszünk rávilágítani az implementációs kérdésekre is.

A jelölésrendszerünk a Haskell nyelv szintaxisához hasonló lesz, de használunk grafikus jeleket (például -> helyett $\rightarrow$) és görög betűket, ahol az érthetőség megkívánja. Egy másik különbség a λ betű használata a Haskell λ-függvényeiben a szokásos '\' helyett.

3.1. Beépített monádok

A fejezet többi részében a kategóriaelméleti szemléletet felváltja a Haskell funkcionális nyelv eszközrendszere. Az alfejezet egyik célja, hogy egy utolsó példát mutasson a kettő közötti kapcsolatra. Kezdjük a kategóriaelméleti eszközökkel alaposan feltérképezett listának Haskellbe transzformálásával. Emlékeztetünk a megfelelő $(List, \eta, \mu)$ triple definíciójára:

$$List A = DATA(A\oplus)$$

ahol:

$$A \oplus B = 1 + (A \times B)$$

és

$$\eta(x) = Cons\ x\ Nil$$

végül

$$\mu(Nil) = Nil$$

$$\mu(Cons\ x\ xs) = x\ ++\ \mu\ (xs)$$

Haskellben a Kleisli triple alakot használjuk, tehát végre kell hajtanunk a triple $\to$ Kleisli triple transzformációt, így azt a $(List, \eta, \_^*)$-ot kapjuk, ahol a monádban szereplő $List$-et megszorítottuk a típusokra, η-t megtartottuk és $f^* = List(f); \mu_B = map(f); \mu_B$, az $f : A \to TB$ függvényre.

A definiáló funktor megfelel a következő típusdefiníciónak:

```
data List a = Nil | Cons a (List a)
```

Mivel a Haskell beépített list típusa a tömörebb zárójeles alakot használja, így a továbbiakban áttérünk erre. Emlékeztetünk a monád osztályra:

```
class Monad m where
(>>= )        :: m a → (a → m b) → m b
return        :: a → m a
(>> )         :: m a → m b → m b
fail          :: String → m a
m >> k        = m >>= λ x → k
```

A fentiek alapján a két legfontosabb művelet példányosítása listákra:

```
instance Monad [] where
  xs >>= f = μ(map f xs)
  return x  = [x]
```

Amiből μ definíciójával adódik:

```
instance Monad [] where
  (x:xs) >>= f = (f x) ++ (map f xs)
  [] >>= f = []
  return x  = [x]
```

ez pontosan megegyezik a Haskell prelude-ben megtalálható definícióval (bár a `fail` művelettel nem foglalkoztunk). A műveletek működése nagyon egyszerű: `return` egy elemet becsomagol egy listába, >>= pedig a paraméterül kapott listában elemenként alkalmazza az `f` függvényt, `f` típusából látható, hogy ez egy listákból álló listát eredményez, amit a μ függvény egyetlen listába olvaszt össze.

3.1.1. Megjegyzés Általánosságban elmondható, hogy a >>= művelet valami hasonlót végez, ennek oka az $f^* = T(f); \mu_B$ átírásban keresendő. $T(f)$-et ugyanis $fmap(f)$-el azonosítottuk, ami úgy definiálható (és ez strukturális indukcióval igazolható), hogy a típust definiáló funktorban a típusparaméter előfordulásait f-re cseréljük, így mindig egy elemenkénti feldolgozást kapunk, aminek az eredménye valamilyen $T^2(A)$ típusba tartozó elem. A definíció első része ráadásul egyértelmű, hiszen $T(f)$-et a típust definiáló funktor meghatározza. Monádok írásakor tehát csak a μ természetes transzformáció megválasztásában van szabadságunk, ez pedig egy olyan $T^2 \to T$ polimorf függvény, ami a triple definíciójában szereplő asszociatív tulajdonságoknak eleget tesz, listáknál például célszerű a fenti összeolvasztó művelettel implementálni.

3.1.2. Megjegyzés A >>= kifejezés értéke rögzített `M` polimorf típus esetén valamilyen `M b` típusú érték. A továbbiakban a következő terminológiával élünk: egy `M b` elemet **monadikus számítás**nak nevezünk, és abban az esetben, ha a kifejezés normálformában van, **monadikus érték**ként is hivatkozunk rá. Mivel a polimorf típusok felfoghatók különböző konténerekként, a monadikus értékek valamilyen alaptípusbeli objektumokat tárolhatnak. A monadikus számítás többnyire valamilyen >>= kifejezés. Az elnevezést a $\_^*$ művelettel való kapcsolattal indokoljuk, az átírás miatt ugyanis egy olyan szekvenciális számítássorozat jön létre, amiben a kifejezésben jobbra álló >>= műveletek a baloldal kiértékelésvel kapott monadikus értéktől függenek. A típuskonstruktor különböző megválasztásával különböző monadikus számításokról beszélhetünk, néhány példa:

- az `M a = ⊥ | a` típus monadikus számításai parciális számításként foghatók fel;
- az `M a = a | e1 | e2 | e3` különböző kivételeket kiváltó számításokat jelenthet;
- az `M a = Set a` reprezentálhatja a nemdeterminisztikus számításokat;
- az `M a = S → (a, S)` mellékhatások kezelésére használható.

Ez Moggi [6] cikkének alapötlete, és a következő alfejezetekben többet is megvizsgálunk közülük.

A listákkal kapcsolatos Zermelo-Fraenkel ([9]) halmazkifejezések tulajdonképpen egy-egy monadikus számítást írnak le. Vegyük például a következő kifejezést:

```
[(x,y) | x ← [1,2,3], y ← [1,2,3], x /= y]
```

aminek (monadikus) értéke:

```
[(1,2)(1,3)(2,1)(2,3)(3,1),(3,2)]
```

A szabályok szerint a halmazkifejezésekben vesszővel elválasztott komponensek csak a megelőző komponensektől függhetnek. `x`-et és `y`-t **generátor**nak nevezzük, az `x/=y` kifejezést pedig **szűrő**nek. A fenti kifejezés működése a következő: az `x` generátor befutja az `[1,2,3]` lista elemeit, és minden elemre végrehajta az utána következő részeket. Egy rögzített `x` mellett `y` is felvesz egy elemet az `[1,2,3]` listából és ellenőrzi az `x /= y` feltételt, aminek teljesülése esetén az `(x,y)` pár bekerül az eredményt tartalmazó listába.

Nézzük, hogyan írhatnánk fel ugyanezt a >>= művelet segítségével:

```
[1,2,3]        >>= (λx →
[1,2,3]        >>= (λy →
return (x/=y)  >>= (λr →
                       case r of True → return (x,y)
                                 _    → fail "")))
```

Itt kell megjegyeznünk, hogy a listákra definiált `fail` művelet az üres listát adja eredményül. A kifejezés átgondolása legyen az olvasó feladata, az mindenesetre jól látszik a példából, hogy a halmazkifejezések automatikusan átírhatók >>= műveletekkel felírt monadikus számításokká. Mivel monádok használatakor gyakran kell λ-kifejezéseket írnunk, ami rontja a kód olvashatóságát, a Haskell fejlesztői bevezettek egy szintaktikai elemet: **do** -kifejezéseket. Az alapgondolat a következő:

```
do p ← e1; e2   = e1 >>= λp → e2
```

De `p` helyén nem csak változó, hanem tetszőleges minta állhat, ekkor előfordulhat, hogy a bemenet nem illeszkedik a mintára. Ebben az esetben a rendszer a `fail` függvényt hívja. A pontosabb definíció tehát:

```
do p ← e1; e2   = e1 >>= (λv → case v of p → e2; _ → fail "s")
```

Ahol `"s"` a hiba fellépésének helyét írhatja le.

Például a fenti halmazkifejezés átírása:

```
do x ← [1,2,3]
   y ← [1,2,3]
   True <- return (x/=y)
   return (x,y)
```

Ez valóban jobban olvasható: `x` sorban felveszi az elemeket, miközben `y` is befutja a listát és ha az `[x/=y]` lista (emlékezzünk ez `return (x/=y)` jelentése) valamelyik (egyetlen) eleme illeszkedik a `True` kifejezésre (tehát igaz), akkor az eredményhez vegyük hozzá az `(x,y)` párt.

Tehát a halmazkifejezéseket két másik módon is fel tudtuk írni. A három jelölésrendszer nem ekvivalens: a második kettővel felírhatók olyan kifejezések, ami az elsővel nem, ráadásul ezeket tetszőleg monád esetén is használhatjuk Haskellben.

Egy másik előredefiniált Haskell monád a következő:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  Just x >>= k  = k x
  Nothing >>= k = Nothing
  return        = Just
  fail s        = Nothing
```

Ez a monád speciális esete a kivételek szimulálására használtnak, amit a későbbiekben részletesen ismertetünk. A megfelelő Kleisli triple felírását az olvasóra hagyjuk.

3.2. Imperatív nyelvi elemek

Ebben az alfejezetben [5] cikke alapján bemutatjuk, hogy bizonyos esetekben hogyan könnyíthetik meg a monádok a programozók életét. Funkcionális programok írásakor időnként olyan problémákba ütközhetünk, ami miatt az addigi munkánk nagy részét teljesen át kell strukturálni. A jól megírt programot többek között a program folytonosságának alapelve (ahogy [9] nevezi) jellemzi. Ez azt jelenti, hogy a feladat kis megváltozása a programban is csak kis változtatásokat tesz szükségessé. A továbbiakban egy esettanulmányon keresztül vizsgáljuk meg a monádokat abból a szempontból, hogy mennyire képesek az elv alkalmazását elősegíteni.

[5] klasszikus példája a monádokat egy kifejezés kiértékelő programban használja fel. Tegyük fel, hogy a programot a következők szerint kell módosítanunk:

(i) vezessünk be hibakezelést a nullával való osztás kezelésére;

(ii) számoljuk meg a kiértékelés során elvégzett osztásokat;

(iii) adjuk meg a kiértékelés sorrendjét (nyomkövetés).

Egy imperatív nyelvben, vagy az olyan nem tisztán funkcionális nyelvekben, mint például az SML a fenti módosítások nem okoznának különösebb zavart a program szerkezetében, és mindegyiket egyszerűen meg is oldhatnánk kivételkezeléssel, valamilyen globális változó használatával illetve output kiírásával. Tisztán funkcionális nyelvekben azonban a helyzet egyáltalán nem ilyen egyszerű: a feladat fenti módosításai az összes rekurzív hívás átalakítását jelentenék, új paramétereket kellene bevezetni, és ezeket megfelelően kezelni, továbbadni.

A másik megoldás a monádok használata. Az alfejezet második felében megmutatjuk, hogyan strukturálhatjuk úgy a programunkat, hogy az új követelmények csak a monád újradefiniálását és néhány lokális változtatást tegyenek szükségessé. Az itt bemutatott módszerek nem csak ebben a végletekig leegyszerűsített példában állják meg a helyüket, a glasgow-i egyetem programozói is ezeket alkalmazták, mikor a Haskell fordítóprogramját írták - természetesen Haskellben.

3.2.1. Az alapeset

A kalkulátor termek értékét határozza meg. Az egyszerűség kedvéért a termek típusát definiáljuk a következőképpen:

```
data Term = Con Int | Div Term Term
```

Egy term tehát lehet egy konstans `Con a`, ahol "`a`" egy egész szám, vagy egy tört, `Div t u`, ahol `t` és `u` termek. Maga a kalkulátor is nagyon egyszerű:

```
eval           :: Term → Int
eval(Con a)    = a
eval(Div t u)  = eval t ÷ eval u
```

Az `eval` függvény tehát termekhez egész számokat rendel. Ha a term egy konstans, akkor az érték a konstans értéke, ha egy tört, akkor a számláló és a nevező hányadosát adja vissza. A $\div$ az egészosztást jelöli.

Nézzünk néhány példát:

```
answer, error::Term
answer         = (Div(Div(Con 1972)(Con 2))(Con 23))
error          = (Div(Con 1)(Con 0))
```

Ha ezt lefuttatjuk, akkor `eval answer` eredménye `((1972 ÷ 2) ÷ 23)`, azaz 42 lesz. Mivel a kalkulátor nem foglalkozik hibakezeléssel, így `eval error` eredménye nem definiált.

3.2.2. Első változat: kivételkezelés

Tegyük fel, hogy úgy kell a programot módosítanunk, hogy a kezelje a nullával való osztást, és a fenti második példában valami érthető hibaüzenettel szolgáljon a felhasználó számára. Az imperatív nyelvi eszközök közül a kivételkezelés nyújtana elegáns megoldást a problémára.

Egy tisztán funkcionális nyelvben a kivételkezelést az unió típuskonstrukció segítségével szimulálhatjuk a következőképpen:

```
data M a = Raise Exception | Return a
type Exception = String
```

Egy `M a` típusú érték vagy `Raise e` formájú, ahol `e` egy kivétel, vagy `Return a`-val egyenlő valamilyen `a`-ra. (Figyeljük meg, hogy `a`-t két különböző értelemben is használjuk: egyszer típusváltozóként `M a`-ban, és az a típus elemeit befutó változót is így jelöltük.)

Egy szó a 'data' és 'type' deklarációk közti különbségről. A 'data' deklaráció új típust vezet be, jelen esetben M-et, és új adatkonstruktorokat definiál a típusértékek létrehozásához (`Raise` és `Return`). A 'type' deklarációk csak egy új nevet adnak a már létező típusok számára, most például a `String`-et elneveztük `Exception`-nek.

A kalkulátorral elég egyszerű áttérni erre a reprezentációra:

```
eval            :: Term → M Int
eval (Con a)     = Return a
eval (Div t u)   =
     case eval t of
          Raise e → Raise e
          Return a →
                  case eval u of
                       Raise e → Raise e
                       Return b →
                              if b == 0
                              then Raise "osztás nullával"
                              else Return (a ÷ b)
```

Az eval függvény minden hívásakor ellenőrizni kell az eredményt. Ha a művelet kivételt váltott ki, akkor azt kell újra kiváltani, ha nem, akkor a kapott eredményt kell feldolgozni. Az új eval-t alkalmazva `answer`-re az eredmény `Return 42`, míg `eval error` értéke `Raise "osztás nullával"` lesz.

3.2.3. Második változat: állapot kezelés

Felejtsük el egy pillanatra a hibákat és a kivételeket, és térjünk rá a második módosításra, azaz tegyük fel, hogy meg akarjuk határozni a kifejezés kiértékeléséhez szükséges osztások számát. Ezt is egyszerűen megoldhatnánk egy imperatív nyelvben valamilyen változó bevezetésével: kezdetben az értéke legyen nulla, és növeljük eggyel minden osztás után.

A funkcionális világban a változók hiánya miatt kicsit kényelmetlenebb a helyzetünk, ugyanis a kalkulátor implementációjára hárul a szükséges állapottér átadogatása is. Vezessünk be két új típust:

```
data M a = State → (a, State)
type State = Int
```

Most az `M a` típus értékei egyparaméteres függvények. A paraméter egy `Int` típusú állapot, ezen valamilyen számítás elvégzése után az eredmény egy érték- állapot pár lesz.

A kalkulátor adaptációja az új típushoz a következő eredményre vezet:

```
eval              :: Term → M Int
eval (Con a) x    = (a, x)
eval (Div t u) x = let (a, y) = eval t x in
                        let (b, z) = eval u y in
                           (a ÷ b, z+1)
```

Bár az `M` típus definíciója elsőre kicsit meglepő, segítségével nagyon elegánsan tudtuk megoldani a problémát. Figyeljük meg, hogy az `eval` függvények két paramétert adtunk, bár típusa szerint csak egy termet kell kapnia. Ugyanakkor az `eval` eredménye egy függvény, és nekünk azt kell definiálnunk, így két paramétert adunk át, és csak azt írjuk le, hogy az eredményül kapott függvény hogyan viselkedik egy-egy állapotban.

Ha az `eval` első paramétere egy konstans, akkor nem kell változtatnunk az állapoton, így a végeredmény egy olyan pár lesz aminek első komponense a konstans értéke, a másodikban pedig visszaadjuk a kapott állapotot.

Ha az `eval` első paramétere egy osztás, akkor az állapotban tárolt osztások számát eggyel növelnünk kell, de mindenekelőtt meg kell határoznunk az osztás eredményét. Első lépésben tehát kiértékeljük `t`-t, az eredményt eltároljuk az `(a, y)` párban, ahol tehát `a` jelenti `t` értékét és `y` az ennek kiszámításához szükséges osztások számát. Aztán hasonlóan kiszámítjuk `u`-t is, így jutunk a `(b, z)` párhoz, (figyeljük meg, hogy az `eval` második hívásában már `y`-t adjuk át állapotként). Végül az eredmény a segédszámítások eredményeinek hányadosából, és az ezek meghatározásához szükséges osztások számának eggyel növelt értékéből alkotott pár lesz.

Ha kipróbáljuk a programot az `eval answer 0` hívással, az eredmény a várakozásoknak megfelelően `(42,2)` lesz, hiszen két osztást kell végrehajtani a kiértékelés során.

3.2.4. Harmadik változat: nyomkövetés, kimenet előállítása

Végül nézzük azt az esetet, amikor arra vagyunk kíváncsiak, hogy milyen lépéseket kell végrehajtani az eredmény meghatározásához. Szükségtelen említeni, hogy az imperatív nyelvekben ez is pillanatok alatt megoldható egy-egy kiíró utasítás elhelyezésével a megfelelő helyeken, és a tisztán funkcionális nyelveknél megint a teljes program átszervezéséhez jutunk.

Az előző két esethez, hasonlóan most is vezessünk be két új típust:

```
data M a = (a, Output)
type Output = String
```

Az `M a` típus értékei tehát érték-output párok lesznek, és a végrehajtás során a második komponens módosításával állítjuk elő a program futásának nyomát. A kalkulátor implementálása most sem okoz különösebb problémát:

```
eval              :: Term → M Int
eval (Con a) x    = (a, line (Con a) a)
eval (Div t u) x = let (a, x) = eval t in
                        let (b, y) = eval u in
                           (a ÷ b, x ++ y ++ line (Div t u) (a ÷ b))
```

```
line             :: Term → Int → Output
line t a = "eval (" ++ showterm t ++ ") <=" ++ showint a ++ '\n'
```

Az első eset nem szorul különösebb magyarázatra, egyszerűen visszaadjuk "a"-t, és előállítunk egy sort az output-ban.
A második esetet a két **let** alkalmazása a program második változatához nagyon hasonlóvá teszi. Most is eltároljuk az `eval` rekurzív hívásainak eredményét, végül ezek segítségével állítjuk elő az eredményt.
A `line` függvényt használjuk a kimenet soronkénti előállítására, a definíciójában szereplő `showterm` és `showint` függvények termek és egész számok szövegre konvertálását végzik. A "++" operátor a sztring konkatenációt valósítja meg, és `'\n'` jelöli az új sor karaktert. Nézzük meg mi lesz az `eval answer` eredménye:

```
eval (Con 1972) <= 1972
eval (Con 2) <= 2
eval (Div (Con 1972)(Con2)) <= 986
eval (Con 23) <= 23
eval (Div (Div (Con 1972)(Con 2))(Con 23))) <= 42
```

Az eddigiekből úgy tűnhet, hogy az imperatív nyelveken írt programokat könnyebben lehet módosítani funkcionális társaiknál. Természetesen ez nem igaz, egy jó példa a dolog ellenkezőjére, ha most azt a feladatot tűzzük ki magunknak, hogy a kiértékelés lépéseit fordított sorrendben írassuk ki, azaz valahogy így:

```
eval (Div (Div (Con 1972)(Con 2))(Con 23))) <= 42
eval (Con 23) <= 23
eval (Div (Con 1972)(Con2)) <= 986
eval (Con 2) <= 2
eval (Con 1972) <= 1972
```

Míg az imperatív esetben viszonylag sokat kellene ügyeskednünk a módosításhoz, legalábbis ha a kimenet előállításához azt az egyszerű módszert használtuk, ami a kiíró utasításokat a számítások végrehajtásakor futtatja le (tulajdonképpen a számítások mellékhatásaként), a fenti programban mindössze ezt a kifejezést:

```
x ++ y ++ line (Div t u) (a ÷ b)
```

kell kicserélnünk erre:

```
line (Div t u) (a ÷ b) ++ y ++ x
```

3.2.5. Monadikus alapeset

A kalkulátor eddig bemutatott változatainak nagyon hasonló a struktúrája. Ilyen esetekben mindig célszerű elgondolkodni azon, hogy nincs-e valamilyen magasabbrendű összefüggés, az egyébként látszólag távol álló fogalmak között.

Az eredeti kalkulátor típusa `Term` → `Int` volt, amit mindhárom esetben `Term` → `M Int`-re módosítottunk. Az `M` adattípus tulajdonképpen mindig valamilyen számítás eredményét reprezentálta. Ezek a számítások kivételeket tudtak kiváltani, állapotot módosítottak vagy kimenetet állítottak elő. Valószínűleg nem meglepő az olvasó számára az a kijelentés, hogy az `M` adattípus mindhárom esetben egy-egy monádként is felfogható lenne, és pontosan ez volt [6] javaslata is.

Ezen a kis példán ugyan nem igazán látszik, hogy az egyes változtatások mekkora terhet rónának a programozóra, ha egy komolyabb feladat megoldása során kellene hasonló módosításokat tennie, de az látható, hogy a kalkulátort mindhárom esetben teljesen újra kellett írnunk. A monadikus kalkulátor bemutatása során arra szeretnénk rámutatni, hogy a folytonosság elvének megfelelően, az új követelményekhez való adaptáció során csak lokális változtatásokat kell végrehajtanunk a rendszeren.

Először az alapesetet írjuk át monádokat használó alakra. Ehhez a legegyszerűbb monádot, az identitást fogjuk felhasználni. A definíció:

```
data M a = a

instance Monad M where
  return a = a
  a >>= k = k a
```

A függvények nem szorulnak különösebb magyarázatra. Látható, hogy `M` kategóriaelméleti megfelelője az identitás funktor, innen származik a monád neve is.

Nézzük a kalkulátor implementációját ezekkel a jelölésekkel:

```
eval            :: Term → M Int
eval (Con a)    = return a
eval (Div t u)  = eval t >>= λa → eval u >>= λb → return (a ÷ b)
```

A függvény típusából látszik, hogy a bemenete egy term, kimenete egy egész számot tartalmazó konténer. `Con a` kiértékelése az `"a"` paraméter konténerbe helyezését jelenti. `Div t u` meghatározása úgy történik, hogy előbb kiszámítjuk `t`-t, majd a konténerbe kerülő eredményt átvesszük `"a"`-ba, ezután kiszámítjuk `u`-t, az eredményt eltároljuk `"b"`-ben, végül betesszük egy konténerbe `"a"` és `"b"` hányadosát.

A definíciót a **do** -szintaxissal átírva jobban látszik, hogy mi is történik:

```
eval            :: Term → M Int
eval (Con a)    = return a
eval (Div t u)  = do a ← eval t
                     b ← eval u
                     return (a ÷ b)
```

3.2.6. Kivételek és monádok

A kivétel monádban a számítás vagy egy értéket ad vissza, vagy egy kivételt vált ki, de nézzük a definíciót:

```
data M a        = Raise Exception | Return a
type Exception  = String
```

```
instance Monad M where
  return a     = Return a
  m >>= k      = case m of
                     Raise e → Raise e
                     Return a → k a

raise          :: Exception → M a
raise e        = Raise e
```

A `return` most is egyszerűen a paraméter konténerbe helyezését jelenti, de a >>= már érdekesebb. Kérdés, mi legyen egy tetszőleges számítás eredménye, ha a paraméterül kapott monadikus érték egy kivétel. A válasz egyszerű: maga a kivétel és ebben az esetben a számításban később álló >>= műveleteket ki sem kell értékelni. Ugyanakkor ha a paraméter egy egész számot takar, erre kell alkalmazni a jobboldalt. Érdemes elgondolkodni azon, hogy ez a definíció tulajdonképpen megint a `k`-nak a típust definiáló funktorban felvett értékét adja, pontosabban az értékre végrehajtott μ függvény eredményét. A definícióban már a kettő kompozícióját írtuk fel, de megemlítjük, hogy μ-t valójában a következőképpen definiálhatnánk:

```
μ(Raise e)= Raise e
μ(Return Raise e)= Raise e
μ(Return Return a)= Return a
```

A típushoz definiáltunk még egy `raise` műveltet, ami nem más, mint egy adatkonstruktor alkalmazása az e paraméterre, mégis érdemes külön műveletet definiálni rá, arra az esetre felkészülve, hogy `M`-ből absztrakt adattípust készítünk, és elrejtjük a reprezentációt a felhasználó elől.

Ahhoz, hogy a hibakezelést a kalkulátorba építsük, csak a `return (a ÷ b)`-t kell kicserélnünk a következőre:

```
if b = 0
  then raise "osztás nullával"
  else return (a ÷ b)
```

ez nagyjából megfelel egy imperatív nyelvben szükséges változtatásnak. Végül az eredmény:

```
eval            :: Term → M Int
eval (Con a)    = return a
eval (Div t u)  = do a ← eval t
                     b ← eval u
                     if b = 0
                       then raise "osztás nullával"
                       else return (a ÷ b)
```

Érdemes megjegyezni, hogy a definíciók visszahelyettesítése után a monadikus kalkulátor megegyezik a monádok nélküli kivétel-kezeléses változattal.

3.2.7. Állapot monád

Az állapot monádban a monadikus értékek maguk is függvények: egy-egy állapothoz egy állapot-érték párt rendelnek (hasonlóan a monád nélküli kalkulátor második változatához).

```
data M a    = State → (a, State)
type State = Int

instance Monad M where
  return a       = λx →(a, x)
  m >>= k        = λx →let (a, y) = m x in
                          let (b, z) = k a y in
                             (b, z)

tick             :: M ()
tick             = λx →((), x + 1)
run              :: Term → (Int, State)
run t            = (eval t) 0
```

A `return a` most is a paraméter egyfajta becsomagolása, ezúttal egy függvénybe, ami egy állapotra (azaz az `x` paraméterre) alkalmazva mellékhatást nem okoz, az állapotot nem változtatja meg, másik komponensébe pedig a paraméter kerül.

A >>= művelet megint összetettebb: először kiszámítja a paraméterül kapott monadikus értéket, `m`-et, az `x` kezdőállapotra, ezután az eredményt eltárolja az `(a, y)` párban, végül a művelet eredményére és az új `y` állapotra végrehajtja `k`-t, így jut `(b, z)`-hez, a végeredményhez. Figyeljük meg, hogy a definíció a kifejezések kiértékelést szekvencializálja: `m x` kiszámítása meg kell előzze a `k a y` hívás feldolgozását, hiszen ez már az új `y` állapottól függ. Ennek egy kellemetlen hátránya lesz a későbbi vektor monádnál, és egy óriási előnye a Haskell IO-ban.

Ahhoz, hogy a kalkulátort használni tudjuk, bevezetünk még egy `run` műveletet is, ami egy termhez egy érték-állapot párt rendel. Itt egész egyszerűen azt kell megértenünk, hogy a `run` nélkül az `eval` függvényt egy `t` termre alkalmazva egy `State → (a, State)` függvényt kapunk, és ennek kell még egy paraméter, ami a kezdeti állapotot rögzíti, és amivel aztán a számítást tényleg el lehet végezni. Ezért a reprezentáció elrejtésének érdekében követjük azt a gyakorlatot, hogy definiáljuk a `run` függvényt, ami ezt a további inicializáló állapotkomponenst hozzáadja a függvényhíváshoz.

A függvények értékkészletének `State` komponensét mellékhatások reprezentálására fogjuk használni: a kalkulátor működése közben a `tick` függvény segítségével úgy módosíthatja ezt a komponenst, hogy közben nem is kell tudnia létezéséről. Így az alábbi `eval` definíció rekurzív hívásai a kifejezés kiértékelése közben mellékhatásokat okozhatnak az állapottéren. A kalkulátor módosítása nagyon egyszerű, csak a `return (a ÷ b)` helyett kell a következőt írni:

```
tick >>= λ_→ return (a ÷ b)
```

Így a következőhöz jutunk:

```
eval              :: Term → M Int
eval (Con a)      = return a
eval (Div t u)    = eval t >>= λa →
                    eval u >>= λb →
                    tick >>= λ_ →
```

```
                          return (a ÷ b)
```

Ami **do** -szintaxisra áttérve:

```
eval           :: Term → M Int
eval (Con a)   = return a
eval (Div t u) = do a ← eval t
                    b ← eval u
                    tick
                    return (a ÷ b)
```

Ezt azért írtuk ki részletesen, mert érdemes egy kicsit elgondolkodni rajta. Figyeljük meg ugyanis, hogy az a fenti függvénydefiníció nem tartalmazza az állapotot. A felhasználó szempontjából nagyon kényelmes használni, hiszen elég ha csak úgy gondol rá, hogy először kiszámítja `t`-t, az értéket elteszi `a`-ba, aztán ugyanígy elbánik `u`-val, aminek az eredménye b, utána eggyel növeli az osztások számát, végül az osztást elvégezve kapja a term értékét.

A bind művelet "trükkös" implementációjának köszönhetően annak ellenére, hogy a felhasználó észre sem veszi egy második komponens (az állapot) jelenlétét, az a színfalak mögött mégis átadódik, a `tick` művelettel módosítható és a végeredményben is megjelenik.

Nézzük meg egy egyszerűbb példán, mi is történik. Vegyünk a következő kifejezést:

```
eval t >>= λa' → Return (a' + 1)
```

Ha visszanézünk a bind definíciójára, és `m` helyébe `eval t`-t, `k` helyébe pedig (λa' → Return (a' + 1))-et írunk, a következőt kapjuk:

```
eval t >>= (λa' → Return (a' + 1)) =
       λx →let (a, y) = eval t x in
               let (b, z) = (λa' → Return (a' + 1)) a y in
                  (b, z)
```

Ezután a belső λ-kifejezés redukálása következhet:

```
λx →let (a, y) = eval t x in
       let (b, z) = Return (a + 1) y in
          (b, z)
```

Ebből már látható, hogy a második számítás hogyan jut hozzá az első eredményéhez, és az is, hogy annak ellenére, hogy az állapot komponens nem jelenik meg a kifejezésben, hogyan tud mégis jelen lenni az egyes számítási lépéseknél. Így a `run` művelettel kiegészítve a rendszer valóban működőképes lesz.

3.2.8. Nyomkövetés monáddal

A harmadik módosítás talán megint kicsit egyszerűbb lesz. A kimenet előállításához definiáljuk a következő monádot:

```
data M a = (a, Output)
type Output = String

instance Monad M where
  return a      = (a, "")
  m >>= k       = let (a, x) = m in
                    let (b, y) = k a in
                      (b, x ++ y)

out             :: Output → M ()
out str         = ((), str)
```

A `return` definíciója megint természetes módon adódik. Nem állítunk elő kimenetet, csak becsomagoljuk "a"-t egy rendezett párba.

A >>= definíciója ebben az esetben először a baloldali paraméterként megkapott konténert szétvágja két komponensre, aminek eredménye `(a,x)`-be kerül, aztán "a" felhasználásával `k` kiértékelhető, és a két `Output` összefűzésével az eredmény is kiszámítható.

A kalkulátorban a következő módosításokat kell végrehajtani:

i) a `Con a` estben `return a` helyett írjunk `out( line (Con a) a ) >>= λ_ →return a`-t;

ii) a másik ágon pedig `return (a ÷ b)` helyére `out( line (Div t u) (a ÷ b) ) >>= λ_ →return (a ÷ b)`-t.

Ez **do** -szintaxissal felírva:

```
eval             :: Term → M Int
eval (Con a)     = do out( line (Con a) a )
                      return a
eval (Div t u)   = do a ← eval t
                      b ← eval u
                      out( line (Div t u) (a ÷ b) )
                      return (a ÷ b)
```

Ami nem szorul további magyarázatra. Ez a változtatás is lokális és egyszerű, továbbá a definíciók behelyettesítése a monádok nélküli kalkulátor harmadik változatára vezet. Az ott tárgyalt módosítás (a futás nyomának megfordítása), most is egyszerűen elvégezhető, csak az `x ++ y`-t kell `y ++ x`-re cserélni a >>= művelet definíciójában. Ennek kategóriaelméleti hátterét már korábban említettük: gyakorlatilag a μ természetes transzformáció különböző definícióiról van szó.

3.3. Állapotkezelés

Az előző alfejezetben bemutatott állapot monád szerepe a funkcionális programozásban jóval nagyobb, mint azt első pillanatban gondolnánk. Ezen az elven alapszik ugyanis a Haskell IO kezelése is, azonban mielőtt erre rátérnénk még általánosítjuk az állapot monádot, és megmutatjuk, hogy hogyan használható vektorok kezelésére. [11] sokkal részletesebben mutatja be az itt felmerülő kérdéseket. Mi azonban az egyszerűség kedvéért [5] nyomán haladunk tovább.

Vitathatatlan, hogy a vektorok a számítástudományban kiemelkedően jelentős szerepet játszanak. A programok tele vannak vektor indexelésekkel (`x[i]`), és módosításokkal (`x[i] := v`). Funkcionális programnyelvekben azonban nem nyilvánvaló, hogyan valósítható meg hatékonyan a vektorok módosítására szolgáló művelet, hiszen egy felelőtlen update megsértené a hivatkozás átlátszóság szabályát ([9]). Azonban az előző alfejezet ismeretében sejthető, hogy a monádok ebben az esetben is hasznosnak bizonyulnak. Ez a megoldás más szemléletet tükröz, mint az előző alfejezeté. Ott a meglévő eszközeink használhatóságát növeltük, itt viszont új nyelvi elemeket fogunk definiálni. Ehhez a monádok segítségével megvalósított absztrakt adattípusok mellett további primitíveket is hozzá kell adni a nyelvhez.

3.3.1. Vektorok

Legyen `Arr` a vektorok típusa, az indexeké legyen `Ix`, a vektorban tárolt értékeké pedig `Val`. A legfontosabb típusműveletek a következők:

```
newArray    :: Val → Arr
index       :: Ix → Arr → Val
update      :: Ix → Val → Arr → Arr
```

A `newArray` függvény egy olyan vektort ad vissza, amelyikben minden indexen `v` áll; Az `index i x` hívás eredménye az `x` vektor `i` indexű eleme; végül az `update i v x` az `x` vektor `i` indexére beírja a `v` értéket, a többi pozíción pedig nem változtat. A vektorok típusműveleteit a [4] jegyzetből megismert jelölésekkel a következő axiómákkal írhatjuk le:

```
index i (newarray v) = v,
index i (update i v x) = v,
index i (update j v x) = index i x, ha i ≠ j.
```

Természetesen a gyakorlatban ezek a műveletek is bonyolultabbak lennének, például jó lenne, ha előírhatnánk a vektor indexhatárait. Ezzel kapcsolatban ismét [11]-ra utalunk, a példa egyszerűségét pedig azzal indokoljuk, hogy ezen is be tudjuk mutatni a megoldás lényegét.

Az `update` művelet hatékony megvalósítása nyilvánvalóan a vektor adott indexű pozícióján levő érték módosításával történne, de mint az már említettük a tisztán funkcionális nyelvekben ez csak akkor biztonságos, ha a vektor egyszeresen hivatkozott, azaz csak egyetlen pointer mutat rá a kifejezésekből felépített gráfban. Ezt más szóval unique tulajdonságnak is nevezik (bővebben lásd [12]-ben).

Írjunk most is egy egyszerű példaprogramot, hogy bemutathassuk rajta a különböző megoldásokat. Vegyük például egy egyszerű imperatív nyelv funkcionális környezetben megírt interpreterét, aminek a szintaxisát a következő absztrakt adattípusokkal definiáljuk:

```
data Term = Var Id | Con Int | Add Term Term
data Comm = Asgn Id Term | Seq Comm Comm | If Term Comm Comm
data Prog = Prog Comm Term
```

Az `Id` típust az azonosítók (itt nem rögzített) típusaként fogjuk kezelni. Egy term lehet változó, konstans vagy két term összege. Egy parancs (`Comm`) lehet értékadás (`Asgn`), két parancs szekvenciája (`Seq`) vagy egy `If` elágazás. Végül a programok (`Prog`) egy parancs alkalmazását jelentik valamilyen term-re.

A program futásának pillanatnyi állapotát egy vektorban fogjuk tárolni, aminek az indexei lesznek az azonosítók, a pillanatnyi értékek pedig a vektor adott indexű pozícióján tárolt értékek.

```
type State = Arr
data Ix = Id
data Val = Int
```

Végül az interpreter:

```
eval                        :: Term → State → Int
eval (Var i) x   = index i x
eval (Con a) x   = a
eval (Add t u) x = (eval t x) + (eval u x)

exec                        :: Comm → State → State
exec (Asgn i t) x           = update i (eval t x) x
exec (Seq c d) x            = exec d (exec c x)
exec (If t c d) x           = if eval t x = 0
                              then exec c x
                              else exec d x

elab                        :: Prog → Int
elab (Prog c t) = eval t (exec c (newarray 0))
```

Megjegyezzük, hogy a definíció nagyon hasonlít a denotációs szemantikára.

A termek kiértékelését végző függvény egy term és egy állapot alapján dolgozik. Egy változó értékének meghatározása az állapot indexelését jelenti. Az `exec` függvény, ami parancsok végrehajtását végzi, egy parancshoz és a kezdőállapothoz rendeli a végállapotot. Az értékadást az állapot módosításával implementáltuk. Végül az `elab` függvény a `t` termet értékeli ki, miután végrehajtotta a paraméterül kapott `c` programot egy nullákkal inicializált állapot-vektorra.

Ha figyelmesen megnézzük a fentieket, látható, hogy az állapot-vektor helyben történő módosíthatóságának csak az a feltétele, hogy az `update` függvény mohó legyen a paramétereire nézve, azaz értékelje ki a második paraméterét, mielőtt az állapot adott indexére helyezné. Ha ez nem történne meg, akkor az `x` vektor az `i`-edik pozíción tartalmazna egy önmagára való hivatkozást, tehát elvesztené a unique tulajdonságot.

Sok kutatás folyt annak érdekében, hogy automatikusan eldönthető legyen, hogy egy funkcionális program egy vektort egyszeresen hivatkozott módon használ-e. A kutatók szándéka az volt, hogy az eredményeket optimalizációs célokra használják a fordítóprogramokban. Mivel az ilyen analízis viszonylag bonyolult és számításigényes, a programozók egy csoportja úgy látja, hogy sokkal egyszerűbb lenne, ha olyan nyelvi elemeket vezetnének be, amivel a unique tulajdonságot explicit jelezni lehetne. Ezzel kapcsolatban is születtek olyan megoldások, amik a típus rendszeren keresztül adják meg ezt a lehetőséget, erről bővebben olvashatunk például [9]-ben és [12]-ben.

3.3.2. Vektor monád

A vektor monád egész egyszerűen egy olyan állapot monád, aminek az állapota egy vektor. Idézzük fel a definíciót:

```
data M a = State → (a, State)
type State = Arr
instance Monad M where
  return a      = λx →(a, x)
  m >>= k       = λx →let (a, y) = m x in
                      let (b, z) = k a y in
                         (b, z)
```

Az előző alfejezetben bemutatott változatban az állapot egy `Int` volt, és volt két további típusműveletünk `M`-hez. Most három típusműveletünk lesz, ezek a vektor létrehozásához, indexeléséhez és módosításához kapcsolódnak.

```
block          :: Val → M a → a
block v m      = let (a, x) = m (newarray v) in a

fetch          :: Ix → M Val
fetch i        = λx →(index i x, x)

assign         :: Ix → Val → M a
assign i v     = λx →((), update i v x)
```

A `block v m` hívás egy új vektort hoz létre, aminek minden pozícióját `v`-vel inicializálja. Ezután alkalmazza rá az `m` monadikus számítást, ami `Arr → (a, Arr)` függvény lévén egy "`a`" típusú eredményt és egy új állapotot szolgáltat. Végül az eredmény "`a`" lesz.

A `fetch` hívás eredménye egy monadikus érték, olyan függvény, amit a vektor egy pozíciójának lekérdezésére használhatunk. Ennek megfelelően a paraméterül kapott állapoton nem változtat, és eredmény komponensébe a vektor `i`. pozícióján álló elemet helyezi.

Végül az `assign` függvény eredménye egy állapotmódosító függvény. Ez végrehajtáskor a konténerbe eredményt nem helyez (csak az üres "`()`" elemet), de módosítja a vektort az `update` művelettel.

A három műveletből egyedül a `fetch`-ben fordulhat elő, hogy elromlik az egyszeresen hivatkozottság, de ez is kiküszöbölhető, ha `fetch` mohó az első komponensére nézve, azaz a függvény végrehajtásakor az eredményül kapott párba `index i x` redukált alakja (a kérdéses pozíción álló elem) kerül.

Az ilyen és ehhez hasonló esetekben használható az alábbi mintaillesztésen alapuló módszer, amit az unboxed típusokkal kapcsolatosan használnak ([19]). Ha ki szeretnénk értékeltetni valamit a végrehajtás egy pontján, csomagoljuk be valamilyen adattípusba. Például ebben az esetben tegyük fel, hogy Vec típusa a következő:

```
type Vec = Vec Vec#
```

Azaz egy `Vec#` típusba tarozó (dobozolatlan) értéket veszünk körül a `Vec` adatkonstruktorral. Ezután `fetch` definícióját mintaillesztéssel írjuk meg:

```
fetch i x = case index i x of Vec vector → (Vec vector, x)
```

Ezzel a rendszert rákényszerítjük arra, hogy `index`-et a szükséges mértékig redukálja.

Most alakítsuk `M`-et absztrakt adattípussá, aminek öt művelete `return` , `>>=` , `block`, `fetch` és `assign`. Ezek közül `block` szerepe azért kiemelkedő, mert ez az egyetlen, aminek a visszatérési értéke nem monadikus. Nélküle nem lehetne olyan `M`-et használó programokat írni, aminek az eredményében nem szerepel `M`.

Azzal, hogy `M` absztrakt adattípus, biztosak lehetünk benne, hogy az egyszeresen hivatkozottság megőrződik, így az `update` műveletet hatékonyan megvalósítható. Ha megengednénk, hogy a felhasználó a vektor alapműveleteit is használja, leírhatná például ezt:

```
λx → (assign i v x, assign i w x)
```

ami nyilván ütközne az egyszeresen hivatkozottsággal.

Az `M`-et használó interpreter a következő:

```
eval                          :: Term → M Int
eval (Var i) x   = fetch i
eval (Con a) x   = return a
eval (Add t u) x = eval t >>= λa → eval u >>= λb →return (a + b)

exec                          :: Comm → M ()
exec (Asgn i t) x    = eval t >>= λa → assign i a
exec (Seq c d)  x    = exec c >>= λ_ →exec d >>= λ_ → return ()
exec (If t c d) x    = eval t >>= λa →
                              if a = 0 then exec c else exec d

elab                          :: Prog → Int
elab (Prog c t) = block 0(exec c >>= λ_→ eval t>>= λa→return a)
```

A típusból látható, hogy a termek kiértékelését végző `eval` függvény eredménye egy konténerbe helyezett egész szám, aminek ennek meghatározásához használhatja az állapotot. Azt egyelőre még nem tudjuk jelezni, hogy a függvény csak olvassa az állapotot, így kénytelenek vagyunk a típusban kicsit többet feltüntetni. Hasonlóan a parancsok futtatása is kapcsolatban áll az állapottal.

Mivel a program az absztrakt adattípust használja, így nem kell attól tartanunk, hogy megsérül az egyszeres hivatkozottság.

Az újraírt interpreter kicsit hosszabb az előző változatnál, de talán könnyebb megérteni is. Jól látszik például, hogy a `Seq c d` parancs futtásához először lefuttatjuk `c`-t, aztán `d`-t, és ezt nem kell jobbról-balra függvénykompozíciós alakban felírnunk.

Az állapot monádnál már említettük, hogy ez a program "túlságosan szekvenciális" lesz, azaz nagyon megköti a fordítóprogram kezét. Például az `Add t u` kiszámítása is `t` kiszámításával kell, hogy kezdődjön, és `u`-t csak ezután lehet meghatározni. Ez nyilvánvalóan felesleges, ráadásul így elvesztjük a párhuzamos kiszámíthatóság lehetőségét is. Említettük, hogy az `eval` függvény csak olvassa az állapotot, de nem módosítja azt. Ez lehetőséget ad arra, hogy bevezessük a vektor olvasó monádot.

3.3.3. Az olvasó monád

Az állapot monádból természetes módon adódik, hogy az olvasó monád típusa a következő:

```
data M' a = State → a
```

azaz egyszerűen elhagytuk az állapot módosítására lehetőséget adó második komponenst. A megfelelő műveletek a következők:

```
return'         :: a → M' a
return' a       = λx →a

(>>=' )         :: M' a → (a → M' b) → M' b
m >>=' k        = λx →let a = m x in k a x

fetch'          :: Ix → M' Val
fetch' i        = λx →index i x
```

A definíciók nagyjából érthetőek. A `return'` a hívás eredménye egy olyan függvény, ami egyszerűen visszaadja a-t. Az `(m >>=' k) x` hívás először kiszámítja `m`-et, aztán ennek eredményét felhasználva kiszámítja `k a`-t is ugyanabban az `x` állapotban. Figyeljük meg, hogy `return'` elhagyja az állapotot, >>=' pedig megduplázza. A `fetch' i x` hívás egyszerűen visszaadja az `x` vektor `i`. pozícióján álló elemet.

Mivel az állapot olvasó számítások részhalmazai az állapotot író-olvasó számításoknak, így megadható egy függvény, ami az `M'`-beli számításokat `M`-beli számításokká alakítja.

```
coerce          :: M' a → M a
coerce m        = λx → let a = m x in (a, x)
```

A függvény nem tesz mást, mint az `M'`-beli számításhoz egy olyan `M`-belit rendel, ami az állapotot nem módosítja, értéke pedig az `M'` számítás értéke. Az egyszeres hivatkozottság megőrzéséhez a függvénynek mohónak kell lennie a közbenső `a`-ban, például a Clean-ben erre a "let-before" kifejezések (#) használatával nyílik lehetőségünk. Egy monádot kommutatívnak nevezünk, ha érvényes a következő:

```
(m >>= λa →n >>= λb →o) = (n >>= λb →m >>= λa →o)
```

A kifejezésnek természetesen csak akkor van értelme, ha "`a`"-nak nincs szabad előfordulása "`n`"-ben, "`b`"-nek pedig "`m`"-ben. Az `n` és `m` felcserélhetősége tehát azt fejezi ki, hogy kommutatív monádban az egyes számítási lépések kiértékelési sorrendje (a mondott feltételek mellett) tetszőleges.

Mivel az olvasó monád kommutatív, lehetőségünk van a >>=' műveletet párhuzamosítására:

```
m >>=' k        = λx → let a = m x in
                         let b = k a x in
                           par b (seq a b)
```

A definícióban a GRIP processzor jelöléseit használtuk. `par` jelentése, hogy a két paramétert párhuzamosan kell kiértékelni, pontosabban a végrehajtás úgy történik, hogy először elindul egy új folyamat, ami az első kifejezést (b) kezdi kiértékelni, az indító folyamat pedig a másodikkal folytatja.

`seq` hatására a folyamat előbb kiértékeli "a"-t, aztán megvárja "b" kiértékelést, és ezt visszaadja eredményként.

A műveleteket két absztrakt adattípusba csomagoljuk `M`-be és `M'`-be összesen nyolc típusművelettel (`return` , >>= , `return'` , >>=′ , `block`, `assign`, `fetch'` és `coerce`).

Ezeket felhasználva jutunk az interpreter végső verziójához:

```
eval              :: Term → M' Int
eval (Var i)    x = fetch' i
eval (Con a)    x = return' a
eval (Add t u) x = eval t >>=' λa → eval u >>=' λb→return' (a + b)

exec                         :: Comm → M ()
exec (Asgn i t) x = coerce(eval t) >>= λa → assign i a
exec (Seq c d)  x = exec c >>= λ_ →exec d >>= λ_ → return ()
exec (If t c d) x = coerce(eval t) >>= λa →
                             if a = 0 then exec c else exec d

elab                        :: Prog → Int
elab (Prog c t) = block 0(exec c >>= λ_ →
                             coerce(eval t) >>= λa →return a)
```

Ez annyiban különbözik az előzőtől, hogy `eval`-t `M'` használatával írtuk meg, és az első két `M`-et használó függvényben az `eval` hívásokat egy-egy `coerce` veszi körül. Ezzel egyrészt sikerült jeleznünk, hogy `eval` csak olvassa az állapotvektort, másrészt a párhuzamosíthatóságot is visszahoztuk a rendszerbe.

Kicsit visszakanyarodva a kategóriaelmélethez elmondhatjuk, hogy `coerce` valójában nem más, mint egy monád morfizmus, tehát egy olyan $M' \to M$ leképezés, ami a monádok struktúráját megőrzi, azaz:

```
h(return' a) = return a,
h(m >>=' λ a → n) = (h m) >>= λ a → (h n)
```

Összefoglalva elmondhatjuk, hogy a monádok használatával két olyan absztrakt típust vezettünk be, ami képes elrejteni a vektorok implementációját a felhasználó elől és az egyszeresn hivatkozottság megőrzése mellett lehetőséget ad arra, hogy néhány nyelvi primitív bevezetésével hatékony vektorkezeléssel bővítsük a Haskell eszköztárát.

3.4. Konkurens számítások modellezése

Harmadik példánkban megmutatjuk, hogy a monádok segítségével konkurens számításokat is modellezhetünk, anélkül, hogy új primitíveket kellene bevezetni funkcionális programnyelvünkbe. Ez csak egy "csináld magad" párhuzamosítás lesz, az ütemezőt is nekünk kell majd megírni hozzá, és építünk arra, hogy az egyes programszálak hajlandók a vezérlést maguktól átadni, hiszen nem lesz lehetőségünk arra, hogy kívülről szakítsuk meg a futásukat.

Az igazán szép megoldás a konkurens folyamatok modellezésére az lenne, hogy létrehozunk egy imperatív monádot valamilyen fork művelettel. Erre egyébként találhatunk is példát a konkurens

Haskellben [10], ahol további primitíveket is bevezettek. Most azonban arra keressük a választ, hogy a nyelv bővítése nélkül hogyan modellezhető a párhuzamosság.

A példában [15] dolgozatát követjük, és az állapot monádnál is érdekesebb folytatás monádot fogjuk felhasználni. Ez egyike a [6] cikkben javasolt lehetséges számítástípusoknak, ereje abban rejlik, hogy alkalmas a számítások jövőjének eltárolására.

A megoldás azért lesz igazán érdekes, mert a párhuzamos folyamatok atomi műveletei valamilyen már meglevő monád monadikus számításai lesznek.

Szükségünk lesz az író monádra, amire már láttunk példát az imperatív nyelvi elemek beépítése kapcsán. Az író monádnak a monád osztály műveleti mellett van egy írás művelete is, ezt röviden így fejezhetjük ki:

```
class Monad m ⇒ Writer m where
  write         :: String → m ()
```

A tipikus implementációt már láttuk korábban is, általában valahogy így néz ki:

```
type W a = (a, String)

instance Monad W where
  (a, s) >>= k = let (b,s') = k a in (b, s ++ s')
  return x     = (x, "")

instance Writer W where
  write s      = ((), s)
```

Korábban azt is említettük, hogy a monádokhoz többnyire tartozik valamilyen futtató művelet is, ez alól a W sem kivétel:

```
output        :: W a → String
output (a, s)  = s
```

A függvény egyszerűen visszaadja a W-beli számítás közben generált kimenetet.

Előfordul, hogy egy monád alaptípusa maga is egy monád. Erre többnyire azért van szükség, hogy egy már létező funkcionalitást (amit a paraméter monád valósít meg) kibővítsünk valamivel. Ilyenek a különböző monád kompozíciók, például, amikor kivételkezeléssel bővítjük a meglévő számításainkat. Bevezethetjük a monád transzformálók osztályát:

```
class MonadTrans t where
  lift          :: Monad m ⇒ ma → (t m) a
```

Gyakorlatilag arról van szó, hogy az `m` monád monadikus számításait felemeljük a `t m` monádba. A modellben bevezetjük a `C` monád átalakítót amivel minden felemelt monadikus számítást atomi műveletként tudunk majd kezelni.

A konkurens számítások modellezéséhez szükségünk van valamilyen eszközre, amivel a futó folyamatok között kapcsolgathatunk, ehhez azonban meg kell oldanunk a folyamatok állapotának és jövőbeli viselkedésének eltárolását.

A folyamatok felfüggesztéséhez elegánsan használható eszközt nyújtanak a **folytatások** (continuations). Egy létező függvényből úgy készítünk folytatást használót, hogy egy további paraméterben megadjuk a függvény folytatását. Ezután ahelyett, hogy a függvény visszaadja az eredményt, alkalmazza a kapott folytatásra, így ha a folytatás egy számítássorozatot takar, akkor annak lépései függhetnek a megelőző lépések eredményétől. Mivel a modellben az `Action` típus fogja a számításokat jelölni, az `"a"` eredményű folytatást használó függvény a következő típussal rendelkezik:

```
type C a = (a → Action) → Action
```

Ahol a függvénynek átadott folytatás az `a → Action` függvény. Erre az eredményét alkalmazva (ami egy `a` típusú érték), egy következő számítást (`Action`) kapunk eredményül. Esetünkben azonban az `Action` összetett, polimorf típus, típuskonstruktora egy monádot kap paraméterként. Ennek megfelelően `C` valódi típusa:

```
type C m a = (a → Action m) → Action m
```

Ezt a `C`-t fogjuk konkurens monád transzformálónak nevezni, ami lényegében azt jelenti, hogy `C m` egy monád minden `m` monád esetén:

```
instance Monad m ⇒ Monad (C m) where
  f >>= k = λc → f (λa → k a c)
  return x = λc → c x
```

Mivel a definíció elsőre nehezen olvasható, vizsgáljuk meg részletesebben. A `return` eset az eddigi példáinkhoz hasonlóan most is egyszerűbb. Ennek típusa általában:

```
return        :: a → m a
```

Azaz jelen esetben:

```
return        :: a → C m a
```

A fenti definíció azt fejezi ki, hogy valamilyen `x` paraméterhez `return x` azt a függvényt rendeli, aminek értéke konstans `x`. Ennek megfelelően az eredmény folytatást használó alakja az a függvény, ami paraméterét (a `c` folytatást) `x`-re alkalmazza.

A >>= esetben `c` az összekapcsolt monadikus számítás folytatását jelenti. Mivel `f` egy folytatást használó függvény, paramétere a végrehajtását követő folytatás kell legyen. Ez adja a lehetőséget arra, hogy `k` és `c` összekombinálásából létrehozzuk a paramétert. Már csak az a kérdés, hogy mi a definícióban szereplő `"a"` változó szerepe, ez pedig nem más, mint az `f` monadikus értékben, mint konténerben, tárolt eredmény. Így az `f` monadikus értékétől függő `k`-ból `k a`-val létrejön egy folytatást használó függvény, amire az egész kifejezés `c` folytatása alkalmazható.

A következő kérdés, hogy milyen legyen egy atomi számítási lépés, azaz egy `Action`. Nyilván szükségünk lesz vezérlő primitívekre és ténylegesen számítást végző elemekre is (ezek lesznek az `m` monadikus számításai). Vegyük a következő típusdefiníciót:

```
data Action m = Atom (m (Action m))
              | Fork (Action m) (Action m)
              | Stop
```

Az `Atom` adatkonstruktor szolgál az `m` alapmonád számítási lépéseinek becsomagolására. Ezen kívül szükség van még egy `Fork` primitívre, amivel új folyamatot indítunk, illetve egy `Stop`-ra, amivel a folyamat végét jelezzük majd az ütemező számára.

Mivel rendszer folytatást használó függvények kiértékelésével működik, mindhárom adatkonstruktorral létrehozott `Action` valamilyen folytatást használó függvény belsejébe lesz ágyazva. Speciálisan ez igaz az `Atom`-al becsomagolt `m` monadikus számításokra is, és azért kell ezeknek `Action m`-et tartalmazni eredményként, mert az lesz az `m` atomi lépést követően végrehajtandó műveletsorozat, de nézzük a megfelelő típusműveletet:

```
atom          :: Monad m ⇒ m a → C m a
atom m        = λc → Atom (do a ← m; return (c a))
```

Ebből látható, hogy egy `m` monadikus művelet kiértékelése után az eredményt a függvény folytatására alkalmazva kapjuk a következő műveletsorozatot.

Szükségünk lesz három további típusműveletre a kényelmes használat érdekében. Először egy olyan függvényt definiálunk, ami egy `C m a`-beli kifejezést `Action m`-é alakít:

```
action        :: Monad m ⇒ C m a → Action m
action m      = m (λa → Stop)
```

Ezzel az `m` számítás végrehajtása után lezárjuk a számítássorozatot.

Következő típusműveletünk a `stop` lesz, ezt akármilyen folytatásra alkalmazva az eredmény a `Stop Action`.

```
stop          :: Monad m ⇒ C m a
stop          = λc → Stop
```

Végül a `Fork` konstruktorhoz is bevezetünk egy műveletet, ami a Unix-ból ismert fork-hoz hasonlít. Egyszerűen alkalmazza az argumentumára az `action` függvényt, így egy olyan folyamatot kap, aminek a végén a folytatás a `Stop`, tehát ott az interpreter terminálni fogja az adott programszálat. Ebből és a folytatásból a függvény egy `Fork Action`-t képez:

```
fork          :: Monad m ⇒ C m a → C m ()
fork m        = λc → Fork (action m) (c ())
```

A `C` típuskonstruktorra példányosíthatjuk a MonadTrans osztályt, ugyanis az `atom` függvény pontosan a `lift` által megkövetelt típusú:

```
instance Monad m ⇒ MonadTrans C where
  lift = atom
```

Most már vannak eszközeink arra, hogy különböző `C m a`-beli elemeket hozzunk létre, de még semmit sem tudunk kezdeni velük. A konkurens számítások modellezésére bevezetünk egy kis interpretert. Ez az [18]-ban is ismertetett round Robin ütemezési eljárással fog működni. A futó folyamatok egy listában vannak felsorolva, az interpreter minden lépésben leveszi a lista elején álló folyamatot (ez valójában egy `Action` típusú elem), és elvégez egy számítási lépést vagy valamilyen vezérlőműveletre reagál:

```
round           :: Monad m ⇒ [Action m] → m ()
round []        = return ()
round (a:as)    = case a of
   Atom am      → do a′ ← am ; round (as ++ [a′])
   Fork a1 a2   → round (as ++ [a1, a2])
   Stop         → round as
```

Egy `Atom` esetén monadikusan futtatjuk az argumentumot, ennek az eredményét (ami `Action m` típusú) hozzáadjuk a folyamat lista végéhez, és folytatjuk `round` futtatását. A `Fork` hatására létrejön két további folyamat, ezeket a folyamat lista végéhez adjuk. Végül a `Stop` esetén az aktuális folyamatot elhagyjuk a listából. Mint a legtöbb monádhoz, `C m`-hez is szükségünk van run függvényre, ez először alkalmazza az `action` függvényt az argumentumára, így létrejön egy a végén terminált `Action`, amit egy listába helyezve a `round`-nak át lehet adni.

```
run             :: Monad m ⇒ C m a → m ()
run m           = round [action m]
```

Látható, hogy "a" nincs jelen a visszatérési értékben, így elveszítjük az eredeti számítás eredményét. Ugyanakkor szerencsére sokszor csak a számítás mellékhatására vagyunk kíváncsiak, és a program is módosítható úgy, hogy visszaadja ezt az eredményt is.

Nézzük meg egy példán, hogyan használható az új modellünk. Vizsgáljuk a konkurens output-ot. Próbáljuk meg az író monádot felemelni a konkurens világunkba, ehhez az összes `m` író monádhoz rendelt `C m` monádhoz példányosítani kell a Writer osztályt:

```
instance Writer m ⇒ Writer (C m) where
  write s = lift (write s)
```

Mivel az előbb úgy példányosítottuk a `MonadTrans` osztályt `C m`-re, hogy a `lift` függvényt az `atom`-mal implementáltuk, mostantól minden `write` művelet egy-egy atomi lépés lesz a konkurens modellben.

Mielőtt a példára rátérnénk, még bevezetünk egy külső függvényt - a `loop`-ot. Ez a függvény minden `Writer`-beli típusra működik. Argumentuma egy sztring, erre alkalmazza ciklikusan a kiírást a következőképpen:

```
loop            :: Writer m ⇒ String → m ()
loop s          = do write s; loop s
```

A függvényt arra használjuk, hogy két `C m a`-beli számítást hozzunk létre, amik folyamatosan generálnak egy-egy szót. Az első folyamat a diploma szót ismétli, a második pedig a munkát:

```
example         :: Writer m ⇒ C m ()
example         = do write "start:"
                     fork (loop "diploma")
                     loop "munka"
```

Az `output (run example)` kifejezés eredménye a következő - végtelen hosszú - sztring lesz:

```
"start:diplomamunkadiplomamunkadiplomamunkadiplomamunkadip...."
```

Figyeljük meg, hogy a kiíró műveletek nem zavarják össze egymást, hiszen egy-egy sztring kiírása az atomi művelet. Ahhoz, hogy változatosabb kimenetet kapjunk a következőképpen kellene példányosítani a `Write` osztályt `C m`-re:

```
instance Writer m ⇒ Writer (C m) where
  write []      = return ()
  write (c:s)   = do lift (write [c]); write s
```

Most karakterenként "emeljük fel" a kiírást, így a `output (run example)` eredménye valami hasonló:

```
"start:dmiupnlkoammaudnikpalmoumnakdaimouknok..."
```

Érdemes megjegyezni, hogy mivel a Haskell egy lusta funkcionális nyelv, így az eredmény annak ellenére kiíródik a képernyőre, hogy a folyamat sosem terminál...

3.5. Röviden a Haskell IO-ról

Utolsó példánk a Haskell IO-ról szól. Az érdeklődő olvasónak mindenképpen ajánljuk S.L. Peyton Jones és Philipp Wadler cikkét [17] az IO funkcionális megvalósítási lehetőségeiről, és konkrétan a Haskell IO implementációjáról. Itt csak a lényeget szeretnénk elmondani, és néhány példát mutatni a használatra, ebben nagy segítségünkre lesz [16].

3.5.1. Egyszerű példák

Kezdjük is mindjárt néhány alapművelettel. A koncepció lényege, hogy az IO műveletek az `IO` polimorf típusba tartozó úgynevezett **akciók**, és csak ezek az akciók alkalmasak a külvilággal való kommunikáció lebonyolítására. Más szóval mellékhatások csak az `IO` típus környezetében fordulhatnak elő, így jól elkülöníthetők a nyelv tisztán funkcionális részétől.

Minden IO akció visszaad valamilyen értéket egy `IO` konténer belsejében. Például a karakterek beolvasására szolgáló művelet típusa:

```
getChar          :: IO Char
```

Az eredményül kapott konténerbe a beolvasott karakter kerül. Vannak olyan akciók is, amik (látszólag) nem adnak vissza semmilyen eredményt. Erre példa a következő:

```
putChar          :: Char → IO ()
```

A karakterek kiírását végző akció eredménye a Haskell unit típusába tartozó üres elem. Talán nem árulunk el nagy titkot az olvasónak, ha megemlítjük, hogy az akciók összekapcsolására a >>= művelet használható. Az IO műveletek kapcsán azonban a **do** -szintaxis használata az elterjedt. Nézzünk egy egyszerű példát, ami beolvas egy karaktert, majd visszaírja a képernyőre:

```
main             :: IO ()
main = do c ← getChar
          putChar c
```

Hangsúlyozzuk, hogy mivel minden IO művelet eredménye egy `IO` konténerben van, így a programozó sem tud másképpen adatokat visszaadni egy IO végrehajtását követően. Például a következő:

```
ready          :: IO Bool
ready = do c ← getChar
           c == 'y' -- hiba!!!
```

függvény hibás, a logikai értéket be kell csomagolni a `return` művelettel:

```
ready          :: IO Bool
ready = do c ← getChar
           return (c == 'y')
```

Most már bevezethetünk egy összetettebb függvényt is, ami sorok beolvasására szolgál:

```
getLine        :: IO String
getLine = do c ← getChar
             if c == '\n'
               then return ""
               else do l ← getLine
                       return (c ++ l)
```

Az IO lehetőséget nyújt kivételkezelésre is. Ez nem pontosan egyezik meg a korábbi kivételkezeléssel, ugyanis az csak a kivételek kiváltására volt alkalmas, itt azonban el is fogjuk tudni kapni őket.

A Haskell előre definiál néhány kivételt a speciális `IOError` adattípus segítségével. Ez egy absztrakt adattípus, maga a konstruktor is rejtve van a felhasználó előtt. Használhatunk viszont néhány predikátumot a hiba típusának eldöntésére. Például az

```
isEOFError     :: IOError → Bool
```

igazat ad vissza, ha a kérdéses hiba egy fájl vége miatt következett be. A kivételek elkapására a `catch` függvény használható, típusa:

```
catch          :: IO a → (IOError → IO a) → IO a
```

Két argumentuma egy akció és egy kivételkezelő függvény. Az utóbbi akkor hívódik meg, ha az első paraméter végrehajtása során valamilyen hiba lép fel. Egy egyszerű példa lehet a `getChar` módosítása úgy, hogy hiba esetén az eredmény legyen az újsor karakter.

```
getChar'       :: IO Char
getChar' = catch getChar (λ e → return '\n')
```

Persze intelligensebb megoldást kapunk, ha a kivétel típusát is ellenőrizzük:

```
getChar'       :: IO Char
getChar' = catch getChar eofHandler where
  eofHandler e = if isEOFError e
                   then return '\n'
                   else ioError e
```

Az `ioError` függvény a kivétel újrakiváltására használható. Típusa elárulja, hogy a kivételek is action formájában utaznak felfelé az egymásbaágyazott `catch` függvények belsejében:

```
ioError         :: IOError → IO a
```

`getChar'`-t felhasználva újraírjuk a sorok beolvasását végző függvényt is, hogy bemutassuk az egymásbaágyazott kivételkezelést is:

```
getLine'        :: IO String
geLine' = catch getLine'' (λerr → return ("Hiba " ++ show err))
  where
    getLine'' = do c ← getChar
                   if c == '\n'
                     then return ""
                     else do l ← getLine'
                             return (c ++ l)
```

A beágyazott kivételkezelés lehetővé teszi, hogy `getChar'` elbánjon a fájl végével, míg minden más hibát a külső függvény kezel, aminek eredménye a "Hiba" szóval kezdődő sztring. A Haskell is definiál egy alapértelmezett kivételkezelőt a program legfelső szintjén, ami ehhez hasonlóan kiírja a kivétel szövegét, és megállítja a programot.

Végül nézzünk egy példát fájlkezelésre is. Első lépés a fájl megnyitása, ez létrehoz egy `Handle` típusú "handler"-t, amivel a későbbi IO műveletekben a fájlra hivatkozhatunk. A megnyitással és bezárással kapcsolatos legfontosabb műveletek és típusok a következők:

```
type FilePath   = String
openFile        :: FilePath → IOMode → IO Handle
hClose          :: Handle → IO ()
data IOMode     = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

A `Handle` típus nem csak fájloknál használatos. Ezzel azonosíthatjuk az összes IO csatornát. Több előre definiált csatornából is választhatunk, ilyenek az `stdin` szabványos bemeneti- és a `stdout` szabványos kimeneti csatornák. Vannak karakterenkénti IO műveletek, mint a `hGetChar` és a `hPutChar`, ezek egy-egy handler-t várnak paraméterül. A korábban bevezetett `getChar` függvényt így is definiálhatjuk:

```
getChar = hGetChar stdin
```

A Haskell az egész fájl vagy csatorna tartalmának sztringbe olvasására is tartalmaz egy műveletet:

```
getContents     :: Handle → IO String
```

Úgy tűnhet, hogy ez nagyon erőforrás-igényes. Valójában a nyelv lustaságának köszönhetően nincs szükség az egész input azonnali beolvasására, adott esetben elég karakterenként vagy valamilyen pufferbe töltéskor kisebb egységenként előállítani. Ezt kihasználva az alábbi fájlok másolását végző program is működőképes lesz:

```
main = do fromHandle ← getAndOpenFile "Bemeneti fájl:" ReadMode
          toHandle   ← getAndOpenFile "Másolás ide: " WriteMode
          contents   ← hGetContents fromHandle
```

```
                hPutStr toHandle contents
                hClose toHandle
                putStr "Rendben."

getAndOpenFile :: String → IOMode → IO Handle

getAndOpenFile prompt mode =
   do putStr prompt
      name ← getLine
      catch (openFile name mode)
            (λ_ → do putStrLn (name ++ " nem nyitható meg.\n")
                     getAndOpenFile prompt mode)
```

A `main` első két sorában megszerezzük a bemeneti és a kimeneti fájl handlerét. A harmadik sorban a `contents` (pszeudo)változóba töltjük az inputfájl tartalmát, amit a következő sor `hPutStr` hívása ír ki az outputfájlba (a lustaságnak köszönhetően ebben történik a fájl tényleges beolvasása is). Ezután a kimenetet lezárjuk (a bemenet lezárása automatikusan megtörténik), és kiírjuk a `"Rendben."` szöveget tájékoztatva a felhasználót a művelet sikeres befejezéséről.

A másik függvény szolgál a fájlok megnyitására. Először kiírja a paraméterül kapott `prompt` sztringet, majd egy fájlnevet tartalmazó sort vár a standard bemeneten. A kapott fájlt megpróbálja megnyitni, ha ez sikerül a visszatérési érték az `openFile` által adott `IO Handle`, egyébként a kivételkezelő függvényben a felhasználó tájékoztatása után rekurzívan újrahívjuk a `getAndOpenFile` függvényt.

3.5.2. A megvalósításról

A példák után vizsgáljuk meg kicsit közelebbről az IO megvalósításának hátterét. Szerencsés helyzetben vagyunk, ugyanis már minden fontosabb eszközt és módszert bemutattunk. A Haskell az IO-t (természetesen) monádokon keresztül valósítja meg. Ehhez a következő absztrakt adattípust használja:

```
data IO a = World → (a, World)
```

ami a monád osztály példányosítása után nem más mint egy olyan állapot monád, aminek állapot komponense ezúttal a `World` absztrakt adattípus. Ebbe gyakorlatilag minden beletartozik, amihez egy IO műveletnek köze lehet, ilyenek a fájlrendszer, a standard input /output streamek, vagy akár egész különleges adatfolyamok is például a különböző hálózaton keresztüli adatforgalom kezelésére szolgálók.

A `World` típusra egyetlen műveletet definiáltak a fejlesztők, `ccall`-t, amivel külső függvényhívásokat hajthatunk végre. A különböző számítógépes architektúrákon futó különböző operációs rendszereken más és más rendszerhívással hajtható végre egy-egy IO művelet. A végrehajtás egy pontján tehát mindenképpen át kell adni a vezérlést az operációs rendszernek, ez indokolja a függvény bevezetését.

Másrészt az egyetlen `ccall` művelet bevezetése elég ahhoz, hogy bármely külső rendszerhívást elérjünk, sőt akár a programozó is írhat új nyelvi primitíveket egy másik programozási nyelv segítségével, így bővítve a Haskell eszközeit.

A `ccall` művelet implementációjánál szükséges az argumentumok kiértékelése a hívás előtt, ezt a vektorok beépítésénél ismertetett mintaillesztéses módszerrel érhetjük el.

Nézzük a monád osztály példányosítását az IO típusra:

```
instance Monad IO where
  return a w    = (a, w)
  (m >>= k) w  = let (a, w') = m w in k a w'
```

A két művelet működése látszólag teljesen megegyezik a korábban bemutatott állapot monáddal. Egy IO monadikus számítás végrehajtásakor a világ kezdeti állapotára alkalmazzuk a számítás első lépésére, az eredményből kicsomagoljuk a világ új állapotát, aminek segítségével végrehajthatjuk a további lépéseket.

Ha a rendszer tényleg így működne, elviselhetetlenül lassú lenne, hiszen a háttérben minden lépésben létre kéne hozni egy új "világ" objektumot, azaz minden csatornát be kellene zárni, újra megnyitni (hálózaton ezt a különböző biztonsági előírások tovább nehezítik) vagy például ugyanazon a gépen két folyamat közti adatcserénél a memóriapointereket meg kellene őrizni, így az erőforrásigény mellett a megvalósítás is borzasztó nehéz lenne.

Szerencsére az IO egy absztrakt adattípus, így a felhasználó nem férhet hozzá közvetlenül a világ objektumhoz (mint ahogy egyik korábbi példánkban (3.2.7) is egy külön `tick` művelet végezte az állapot módosítását), ráadásul a monád fenti példányosítása a világot egyszeresen hivatkozott módon használja. Ebből következik, hogy a belső implementációnak nincs is szüksége a "világ" objektumra: minden változást a valódi világra, a fájlrendszerre, a hálózati adatfolyamokra stb. alkalmazhat.

Felmerül a kérdés, hogy nem lehetne-e valahogy teljesen a "világ" objektum nélkül megoldani a feladatot. Válaszként tekintsük a következő egyszerű példát:

```
[1/0] >>= λc → return 5
```

Kérdésünk a következő: mi lesz ennek az eredménye?

A rendszer a >>= műveletet úgy értékeli ki, hogy a baloldalán álló listára elemenként alkalmazza a jobboldali függvényt, majd az eredményeket összefűzi egyetlen listába. Mivel a jobboldal a konstans öt (pontosabban `[5]`) függvény, a lista `1/0` elemét a nyelv lustasága miatt nem kell kiértékelni, tehát a válasz: `[5]`.

Ugyanakkor vegyük a következő példát:

```
getChar >>= λc → putChar 'x'
```

A jobboldal most sem függ a baloldalon beolvasott karaktertől. Ezek szerint annak beolvasására semmi szükség nincs? Az IO folyamatok kezelésénél nem ezt a viselkedést várnánk, és a világ objektum jelenléte teszi lehetővé, hogy a jobboldal ha nem is explicit, de a >>= művelet definíciója miatt implicit függ a baloldal eredményétől, hiszen `k a`-t nem a kezdeti `w` világra, hanem a `getChar` feldolgozása után létrejövő `w'`-re alkalmazzuk. Az egyik korábbi alfejezetben (3.3) bemutatott vektor megvalósításnál hátrányos "túlságos szekvenciáltság" ezúttal tehát segítségünkre van: ez teszi lehetővé a rendszer megfelelő működését.

4. Összefoglalás

Néhány alapvető matematikai fogalom rögzítése után a kategóriaelmélet segítségével modelleztük a funkcionális nyelveket. Megvizsgáltuk, hogy az elmélet egyes fogalmai milyen konkrét nyelvi fogalmaknak feleltethetők meg. Modellünkben a **Set** kategóriához hasonló kategóriában dolgoztunk. Láttuk, milyen feltétel szükséges ahhoz, hogy a modellezett nyelvben megtalálható curring-et bevezethessük. A funktorok fogalmánál több példán keresztül bemutattuk, hogyan modellezhetők a nyelv különböző típusai segítségükkel. A természetes transzformációk bevezetésekor megállapítottuk, hogy minden természetes transzformációnak egy-egy polimorf függvény feleltethető meg a programozási nyelvben. Ezután rátértünk a triple fogalmára, és bemutattuk ekvivalenciáját a Kleisli triple struktúrával. Megvizsgáltuk a kapcsolatot a Kleisli triple és a Haskell monád osztálya között, majd tettünk egy rövid kitérőt, amiben további kapcsolatokra hívtuk fel a figyelmet a triple és az adjungáltak között.

Modellünk viszonylag kezdetleges volt, de arra mindenképpen alkalmas, hogy az elmélet és a Haskell közötti kapcsolatra rámutasson. Egyik nagy hiányossága az volt, hgy megköveteltük a függvények teljességét. Ez kiküszöbölhető ([13]), de ehhez egy másik kategóriára kellett volna felépítenünk a modellt. Ebben a kategóriában már az egyes objektumok szerkezetéről is különböző feltételezéseket kell tenni, így jutunk a teljes parciális rendezések kategóriájába (**Cpo**). Ebben a kategóriában is elmondható mindaz, amit a modellünkben bevezettünk, de a parciális függvények modellezésére is alkalmas.

A dolgozat második részében rátértünk Moggi ötletének ([6], [7]) vizsgálatára a Haskell nyelv keretein belül, itt már nem a formális gondolatmenetet követtük, hanem Wadler egyik cikke alapján ([5]) a konkrét nyelvben vizsgáltuk az eszköz használhatóságát. Több példán is illusztráltuk mennyire változatosan alkalmazhatók a monádok, és röviden arra is kitértünk, hogyan lehet az IO-t megvalósítani segítségükkel ([17]). További alkalmazások találhatók a www.haskell.org ([20]) címen, a Haskell hivatalos honlapján, például több domain specifikus nyelvet is definiáltak monádok segítségével, köztük a Haskore-t számítógépes zeneszerzéshez.

Irodalomjegyzék

[1] M. Barr and C. Wells *Category Theory for Computing Science* (Prentice-Hall International, New York, 1990.)

[2] Csörnyei Zoltán *Lambda-kalkulus* előadás jegyzet

[3] Láng Csabáné *Bevezetés a matematikába II.* (ELTE jegyzet, 1998.)

[4] Kozma László, Varga László *Adattípusok osztálya - Definíciók, elemzés, példák* - ELTE, TTK, Informatikai Tanszékcsoport, Felelős Kiadó: Dr. Kátai Imre, (c) 2001, Budapest

[5] P. Wadler *Monads for functional programming* (M. Broy, editor, Marktoberdorf Summer School on Program Design Calculi, Springer Verlag, NATO ASI Series F: Computer and systems sciences, Volume 118, August 1992.)

[6] Eugenio Moggi *An abstract view of programming languages* (Stanford course notes 1989.)

[7] Eugenio Moggi *Computational lambda-calculus and monads* (IEEE Symposium on Logic in Computer Science, Asilomar, California, June 1989)

[8] Nick Benton and John Hughes and Eugenio Moggi *Monads and Effects*

[9] Nyékyné G. Judit et al. *Programnyelvek* (megjelenés alatt)

[10] Simon Peyton Jones and Andrew Gordon and Sigbjorn Finne *Concurrent Haskell* Conference Record of POPL '96: The 23$^{\mathrm{rd}}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1996.

[11] J. Launchburry & SL Peyton Jones 1996 *State in Haskell* Lisp and symbolic computation (to appear).

[12] P. Koopman, R. Plasmeijer, M. V. Eekelen, S. Smetsers *Functional Programming in CLEAN*, 2001 szeptember, www.cs.kun.nl/$\sim$ clean

[13] Jeremy Gibbons *Calculating Functional Programs* School of Computing and Mathematical Sciences Oxford Brookes University 1997. november

[14] M. B. Smyth and G. D. Plotkin. *The category-theoretic solution of recursive domain equations.* SIAM Journal on Computing, 11(4) : 761-783, 1982 november

[15] Koen Claessen *Functional pearls: A poor man's concurrency monad*

[16] *A Gentle Introduction to Haskell* http://www.haskell.org

[17] Simon L. Peyton Jones, Philipp Wadler*Imperative funtional programming* 1993. január

[18] Tannenbaum A. S. *Operációs rendszerek*, Panem, 1999.

[19] Simon L. Peyton Jones and J. Launchbury *Unboxed values as first class citizens*, (Proceedings of the Conference on Functional Programming and Computer Architecture, Springer-Verlag LNCS523, 1991. augusztus)

[20] *Haskell honlap* http://www.haskell.org

Tartalomjegyzék