Bevezetés a homotópia-típuselméletbe

Kaposi Ambrus
University of Nottingham
kaposi.ambrus@gmail.com

2014. január

1. Bevezetés

A számítógéptudományban sokféle módszer használatos a programok helyes működésének biztosítására: a program futtatása különböző bemenetekkel és a kimenetek ellenőrzése; a program futás idejű monitorozása; a program egy egyszerűsített modelljének szimulálása, és annak ellenőrzése, hogy a szimulált világ lehetséges állapotaiban megfelelő programműködést tapasztalunk -e; a program részleges specifikálása típusokkal, és a program fordítása során ennek a specifikációnak való megfelelés ellenőrzése; a program helyességének bizonyítása formális eszközökkel vagy bizonyítottan helyes program szintézise.

A fentiek közül a típusokkal való specifikálás az egyik legelterjedtebb módszer, hiszen a fordítás egyik fázisaként a fejlesztés folyamatába egyszerűen beépíthető. A fordítóprogram automatikusan elvégzi a típusellenőrzést, a nem típushelyes programot visszautasítja, és ezáltal sokféle, gyakran előforduló programozási hibát képes kiszűrni.

Ha egy típusrendszer nem elég kifejező, az az absztrakció rovására megy, pl. egy egyszerű típusrendszerrel rendelkező programozási nyelvben külön programra van szükség ahhoz, hogy egész számok listájának hosszát ill. karakterek listájának hosszát meghatározzuk, mert ezeknek különböző típusa van, `List Int` $\to$ `Int` ill. `List Char` $\to$ `Int`. Egy lehetőség

ilyenkor a típusrendszer megkerülése azáltal, hogy kiskapukat teszünk bele. Másik lehetőség, hogy kifinomultabb típusrendszert használunk, mely képes polimorf típusokat leírni. Ekkor az előbbi két típus egyesíthető az alábbi típusban: $\forall$ `a : Type . List a` $\rightarrow$ `Int`, és az ilyen típusú program működni fog mindenfajta listára. A Girard-Reynolds típusrendszer [12] megengedi az ilyen konstrukciókat, és ezzel nemcsak lehetővé teszi az absztrakciót, de a programozót rá is kényszeríti a reprezentáció-független programok írására [32], [27]. Erre a típusrendszerre épül az ML [26] és Haskell [30] programozási nyelv.

Ha az előbb említett típusrendszerben szeretnénk a véges elemszámú típusokat megvalósítani, külön-külön kell definiálnunk a `Fin1 : Type` (1-elemű), `Fin2 : Type` (2-elemű) stb. típusokat. Ez a példa azt mutatja, hogy a típusrendszer továbbra is akadályozza az absztrakciót, de most a típusok szintjén. Ha nincs szükségünk nagy biztonságra, egy `Fin` típusba egyesíthetjük az összeset, melyet pl. a természetes számok típusával definiálunk, és a programozóra bízzuk, hogy ahol `Fin2`-t vár a program, ott véletlenül se adjon egy `Fin3`-beli értéket. Ha azonban szükségünk van az elkülönítésre, és nem egy metaprogrammal szeretnénk a típus-definíciókat generálni, mert pl. az elemszámra nincs felső korlátunk, az alábbi típusnak megfelelő programra van szükségünk: `Fin : Nat` $\rightarrow$ `Type`. Ez a `Fin` típusoknak egy indexelt családja, minden egyes természetes számra egy-egy külön típus, az indexelő típus a `Nat`. `Fin 3` pl. a három-elemű típus. Azzal, hogy értékek (egy természetes szám) jelentek meg a típusban, függő típusrendszerhez jutottunk. Függő típusok használatával a Curry-Howard izomorfizmuson [19] keresztül tetszőleges, matematikailag leírható tulajdonság kifejezhető típusokkal (lásd 2.8. pont). A függő típusrendszer az elképzelhető legkifejezőbb típusrendszer, hiszen minden tulajdonság, amit valaha le szeretnénk írni egy programról, matematikai képlettel kifejezhető.

Egy ilyen típusrendszer a programozónak sokféle lehetőséget kínál: annak megfelelően, hogy mekkora biztonságot kíván a feladat, pl. az egészek listájának rendezését végző `sort` programot az alábbi típusokkal szerelheti fel, biztonságosság szerint növekvő sorrendben:

- `sort : List Int → List Int`
- `sort : (xs : List Int) → (ys : SortedList Int)`
- `sort : (xs : List Int) → (ys : List Int) × (sorted ys)`
  `× (length xs = length ys)`
- `sort : (xs : List Int) → (ys : List Int) × (sorted ys)`
  `× (ys ‘permutationOf‘ xs)`

Egy másik lehetőségként a programozó a `sort` program típusát meghagyhatja a legelsőnek, és egy külön programot írhat az alábbi típussal:

```
(xs : List Int) → let ys = sort xs in
                    (sorted ys) × (ys ‘permutationOf‘ xs)
```

Ez a program annak bizonyításának felel meg, hogy a `sort` program egy rendezett listát ad vissza, és ez a rendezett lista a bemeneti lista egy permutációja. Ilyen módon a programok bizonyításokat is tartalmazhatnak, és a programok helyességének (valamilyen szinten való) bizonyítása a programozás egy lépése lehet.

Függő típusrendszerrel rendelkező programozási nyelvek az Agda [29] és Idris [7]. Bevezető jellegű könyvek a típuselméletbe magyarul [10], angolul a logika felől közelítve [13], a programozás felől közelítve [15].

A matematika konstruktív megalapozására függő típuselméleteket régóta használnak [33]. A Curry-Howard izomorfizmuson keresztül a matematikai állítások típusoknak, az állítás bizonyításai adott típusú programoknak felelnek meg. A legnépszerűbb, Curry-Howard izomorfizmust használó számítógépes tételbizonyító rendszer (más szavakkal függő típusrendszerű programozási nyelv) a Coq [24]. Ennek alapja az intenzionális Martin-Löf típuselmélet [22]. Bár nagy és bonyolult matematikai tételeket bizonyítottak ebben a rendszerben ([14], [1]), használata mégis nehézkes, mert olyan, a matematikában (és emiatt a programokról való érvelésben) gyakran használt alapelvek, mint a pontonként egyenlő függvények egyenlősége (függvény

[1] `http://www.msr-inria.fr/news/feit-thomson-proved-in-coq`

extenzionalitás) ill. az izomorf halmazok (típusok) egyenlősége nem teljesülnek benne. Az ekvivalencia-osztályokkal való műveletek is kényelmetlenek, pl. tiszta Martin-Löf típuselméletben a valós számok nem definiálhatók [21]. Ezek a problémák mind a típuselmélet egyenlőség-fogalmával kapcsolatosak. A homotópia-típuselmélet [31] új megvilágításba helyezi az egyenlőség típust, és választ ad a fenti problémákra, egy matematikára és programozásra egyaránt kényelmesebben használható típuselmélet formájában.

Az alábbiakban bevezetjük a Martin-Löf típuselméletet, rávilágítunk az egyenlőség típus néhány tulajdonságára, bemutatjuk azokat az extenzionális alapelveket, amiket használni szeretnénk, majd megmutatjuk, hogy a típusok topologikus terekként való értelmezése hogyan teszi érthetővé az egyenlőség különleges tulajdonságait és teszi lehetővé az előbbi alapelvek használatát.

Ezen írás megértéséhez egy számítástudományi alapképzésnek megfelelő matematikai tudás elégséges, valamely funkcionális programozási nyelv ismerete előny. Néhány témakört helyhiány miatt csak nagy vonalakban tudunk érinteni, a részleteket hivatkozásokkal igyekszünk pótolni. A magyar szóhasználatban [10]-t követjük.

A típuselméletet leggyakrabban a mienkhez hasonló módon, szintaktikusan vezetik be, ami első olvasásra nagyon töménynek, esetleg rejtélyesnek tűnhet. A megadott levezetési szabályok között azonban mély szimmetriák vannak, melyeknek a kifejezésére még nem találtuk meg a legjobb formát – talán a kategóriaelmélet [3] lesz az, de mivel kevesen ismerik az alapfogalmakat, maradunk a szintaktikus bemutatásnál. A kategóriaelmélet nyelvét használó szép bevezetés a típuselméletbe pl. [17].

2. Martin-Löf típuselmélet

Ebben a pontban a Martin-Löf típuselmélet vezetjük be, aki ismerős a témával, az gyorsan átfuthatja a szabályokat, aki nem, annak ez egy gyorstalpaló bevezető lesz. A szabályok olvasásakor az elsőrendű logikában tanultakkal kapcsolatos intuícióra érdemes támaszkodni.

Martin-Löf típuselmélete [22] egy formális matematikai rendszer[2], mely következtetések levezetésére alkalmas. A formális rendszer ábécéjét, nyelvtani szabályait a levezetési szabályokon keresztül implicit módon adjuk meg.

Négyféle következtetési formát különböztetünk meg:

$\Gamma \vdash$	Γ egy érvényes környezet
$\Gamma \vdash t : A$	Γ környezetben t kifejezés típusa A
$\Gamma \equiv \Delta \vdash$	Γ és Δ környezetek definicionálisan egyenlőek
$\Gamma \vdash u \equiv v : A$	u és v, Γ környezetben A típusú kifejezések definicionálisan egyenlőek

Utóbbi esetében a környezetet és a típust gyakran elhagyjuk, és egyszerűen $u \equiv v$-t írunk.

A kifejezésekre gondolhatunk programokként, melyek valamilyen típussal rendelkeznek. A típusra matematikai állításként is gondolhatunk, a kifejezés ennek bizonyítása. Két program definicionálisan egyenlő, ha futtatásuk ugyanarra a végeredményre jut. Pl. $4 + 3 \equiv 7$. A típusok is kifejezések, melyeknek típusa van, így a típusok és kifejezések egy szintaktikus kategóriában vannak, az intuíció miatt különítjük el őket informálisan.

A környezet a kifejezésben és a típusban levő szabad változók típusait adja meg, típusok listája: $x_1 : A_1, x_2 : A_2, ..., x_n : A_n$. Ez a környezet az $A_1 ... A_n$ típusokat tartalmazza, a változónevek csak címkék, azért van rájuk szükség, hogy könnyű legyen hivatkozni a típusokra. Emiatt a pontos név nem érdekes, két környezet, mely csak az elnevezésekben különbözik, definicionálisan egyenlő; hasonlóképpen két kifezejezés, melyben a változók ugyanazokra a helyekre mutatnak, de különböző nevük van, definicionálisan egyenlő (ezt a problémakört α-konverziónak nevezik, mi nem foglalkozunk vele). A_i-ben előfordulhatnak x_j szabad változók, ahol $j < i$. A környezetekre vonatkozó levezetési szabályok az alábbiak:

$\dfrac{}{\cdot \vdash}$ üres környezet $\qquad$ $\dfrac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma, x : A \vdash}$ környezet hosszabbítás

[2]Martin-Löf típuselméletének intenzionális változatát adjuk meg implicit helyettesítéssel, Russel-féle univerzumokkal és definicionális β és η szabályokkal rendelkező Π és Σ típusokkal.

$$\frac{\Gamma, x : A, \Delta \vdash}{\Gamma, x : A, \Delta \vdash x : A} \text{ változó bevezetés}$$

A környezet hosszabbításban x-nek egy olyan változónak kell lennie,ami Γ-ban nem szerepel. U_i minden i természetes számra egy típus. Zárt kifejezéseknek nevezzük azokat, melyekben nincs szabad változó, tehát egy olyan t, melyre valamely A-ra a $\cdot \vdash t : A$ következtetés levezethető.

A definicionális egyenlőség ekvivalancia-reláció, és definicionálisan egyenlő környezetek és kifejezések bármely esetben felcserélhetőek. A következő (nem túl érdekes) levezetési szabályok ezeket fejezik ki, első olvasásra ezek nyugodtan átugorhatóak, a teljesség kedvéért tesszük őket ide:

$$\frac{\Gamma \vdash}{\Gamma \equiv \Gamma \vdash} \qquad \frac{\Gamma \equiv \Delta \vdash}{\Delta \equiv \Gamma \vdash} \qquad \frac{\Gamma \equiv \Delta \vdash \quad \Delta \equiv \Theta \vdash}{\Gamma \equiv \Theta \vdash}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv t : A} \qquad \frac{\Gamma \vdash u \equiv v : A}{\Gamma \vdash v \equiv u : A} \qquad \frac{\Gamma \vdash u \equiv v : A \quad \Gamma \vdash v \equiv w : A}{\Gamma \vdash u \equiv w : A}$$

$$\frac{\Gamma \equiv \Delta \vdash \quad \Gamma \vdash A \equiv B : \mathsf{U}_i \quad \Gamma \vdash t : A}{\Delta \vdash t : B}$$

$$\frac{\Gamma \equiv \Delta \vdash \quad \Gamma \vdash A \equiv B : \mathsf{U}_i}{\Gamma, x : A \equiv \Delta, x : B \vdash} \qquad \frac{\Gamma \equiv \Delta \vdash \quad \Gamma \vdash A \equiv B : \mathsf{U}_i \quad \Gamma \vdash u \equiv v : A}{\Delta \vdash u \equiv v : B}$$

Az utolsó előtti szabályhoz szintén hozzátartozik, hogy x változó friss legyen (ne szerepeljen Γ-ban és Δ-ban).

Ha a $\Gamma \vdash t : A$ következtetés levezethető, akkor azt mondjuk, hogy t az A típus eleme. A típusok típusát univerzumnak hívjuk, a kis típusok U_0-ban vannak, U_0-t magát nagy típusnak hívjuk, mert elemei olyan kifejezések, melyeknek vannak elemei. A legalsó szinten levő kifejezéseknek nincsenek további elemei, ezeket kifejezés-konstruktoroknak (vagy egyszerűen konstruktoroknak) hívjuk (eddig még egy ilyennel sem találkoztunk). Az univerzumokra a következő szabályaink vannak (i bármely természetes szám):

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U}_i : \mathsf{U}_{i+1}} \text{ univerzum képzés} \qquad \frac{\Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash A : \mathsf{U}_{i+1}} \text{ univerzum kumulativitás}$$

Az univerzumok megszámlálhatóan végtelen hierarchiájára azért van szükség, mert az $\mathsf{U}_0 : \mathsf{U}_0$ szabály inkonzisztenciához vezetne (Burali-Forti paradox, [11]) – az inkonzisztencia itt azt jelenti, hogy a $\cdot \vdash t : \mathsf{0}$ következtetés levezethető valamely t-re, ahol $\mathsf{0}$ az üres típus, lásd később.

A típuselméletben egy-egy adott típushoz tartozó levezetési szabályok a következő formában jelennek meg: típusképző szabály, bevezető szabály, eliminációs szabály, számítási (β) szabály, egyediség (unicitás, η) szabály, a konstruktorok definicionális egyenlőséggel való kompatibilitását kifejező szabályok. Ilyen formában adjuk meg a függvény, a függő pár (szigma), a 0-elemű típus, a 2-elemű típus, a természetes számok és az egyenlőség típus levezetesi szabályait. Végül röviden megemlítjük, hogy általános induktív típusokat milyen módon képezhetünk.

2.1. A függvény típus levezetési szabályai

A $\prod_{x:A} B$ függvény típusra gondolhatunk úgy is, mint a $\forall_{x:A}.B$ állításra, melyben az univerzális kvantor az A típus (halmaz) elemeire vonatkozik, tehát azt mondja, hogy bármely x-nek nevezett A-beli elemre teljesül B (amely tartalmazhatja x-et). Pl. $\prod_{x:\mathbb{N}}(n + 2 = 2 + n)$ azt fejezi ki, hogy bármely n természetes számra $n+2$ egyenlő $2+n$-el (a természetes számokat, összeadást és egyenlőséget később definiáljuk).

$$\frac{\Gamma \vdash A : \mathsf{U}_i \quad \Gamma, x : A \vdash B : \mathsf{U}_i}{\Gamma \vdash \prod_{x:A} B : \mathsf{U}_i}\ \Pi \text{ képzés}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \prod_{x:A} B}\ \Pi \text{ bevezetés} \qquad \frac{\Gamma \vdash f : \prod_{x:A} B \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B[a/x]}\ \Pi \text{ elimináció}$$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash a : A}{(\lambda x.t)a \equiv t[a/x] : B[a/x]}\ \Pi\ \beta \qquad \frac{\Gamma \vdash f : \prod_{x:A} B}{\Gamma \vdash f \equiv \lambda y.fy : \prod_{x:A} B}\ \Pi\ \eta$$

$t[a/x]$ egy meta-jelölés arra a kifejezésre, melyet úgy kapunk, hogy t-ben az x változó összes előfordulását a-ra cseréljük. B-t indexelt családnak (típuscsaládnak) nevezzük, a Γ környezetben $B[a/x]$ egy típus minden $a : A$-ra, A az indexelő típus.

$\prod_{x:A} B$-re további három jelölés $\prod\limits_{x:A} B$, $\Pi\ A\ B$ és $(x : A) \to B$. Π és λ változót kötnek meg, melyek csak hatáskörükben látszanak. Mindkettő hatásköre a lehető legtovább terjed, tehát pl. $\lambda x.\lambda y.t \equiv \lambda x.(\lambda y.t)$, vagyis t-ben x és y is szerepelhet. Ha x nem szerepel B-ben, $\prod_{x:A} B$ helyett $A \to B$-t írhatunk. $A \to B \to C$ jobbra zárójeleződik, tehát $A \to (B \to C)$-t jelent. $A \to B$-re gondolhatunk úgy, mint az A-ból következik B logikai állításra. λ egy konstruktor, Π típuskonstruktor, a függvényalkalmazás, melyet csak egymás mellé írással jelölünk, eliminátor.

A konstruktorok respektálják a definicionális egyenlőséget:

$$\frac{\Gamma \vdash A \equiv A' : \mathsf{U}_i \quad \Gamma, x : A \vdash B \equiv B' : \mathsf{U}_i}{\Gamma \vdash \prod_{x:A} B \equiv \prod_{x:A'} B' : \mathsf{U}_i} \qquad \frac{\Gamma, x : A \vdash t \equiv r : B}{\Gamma \vdash \lambda x.t \equiv \lambda x.r : \prod_{x:A} B}$$

2.2. A szigma típus levezetési szabályai

A $\sum_{x:A} B$ típusra úgy gondolhatunk, mint a $\exists_{x:A}.B$ állításra. Annak bizonyítása, hogy létezik egy A-beli elem, melyre B igaz, egy függő pár, mely egy x-nek nevezett A-beli elemből áll, és egy bizonyításból, hogy x-re teljesül B. Az alábbi bevezetési szabály ezt fejezi ki.

$$\frac{\Gamma \vdash A : \mathsf{U}_i \quad \Gamma, x : A \vdash B : \mathsf{U}_i}{\Gamma \vdash \sum_{x:A} B : \mathsf{U}_i}\ \Sigma \text{ képzés} \qquad \frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[u/x]}{\Gamma \vdash (u, v) : \sum_{x:A} B}\ \Sigma \text{ bevezetés}$$

$$\frac{\Gamma \vdash t : \sum_{x:A} B}{\Gamma \vdash \pi_1 t : A}\ \Sigma \text{ elimináció}_1 \qquad \frac{\Gamma \vdash t : \sum_{x:A} B}{\Gamma \vdash \pi_2 t : B[\pi_1 t/x]}\ \Sigma \text{ elimináció}_2$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[u/x]}{\Gamma \vdash \pi_1(u, v) \equiv u : A}\ \Sigma\ \beta_1 \qquad \frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B[u/x]}{\Gamma \vdash \pi_2(u, v) \equiv v : B[u/x]}\ \Sigma\ \beta_2$$

$$\frac{\Gamma \vdash t : \sum_{x:A} B}{\Gamma \vdash t \equiv (\pi_1 t, \pi_2 t) : \sum_{x:A} B}\ \Sigma\ \eta$$

A konstruktorok (Σ és $-,-$) respektálják a definicionális egyenlőséget, ezeket a levezetési szabályokat rövidített formában adjuk meg:

$$\frac{A \equiv A' \quad B \equiv B'}{\sum_{x:A} B \equiv \sum_{x:A'} B'} \qquad \frac{u \equiv u' \quad v \equiv v'}{(u, v) \equiv (u', v')}$$

A nem-függő pár (Descartes-szorzat, logikai és) a függő pár speciális esete, ahol a második elem típusa nem függ az elsőtől: $A \times B$-t egyszerűen $\sum_{x:A} B$ rövidítéseként definiáljuk abban az esetben, ha B-ben nem szerepel x.

$\sum_{x:A} B$-re egy másik elterjedt jelölés $(x : A) \times B$.

2.3. Az üres típus levezetési szabályai

Az üres típus különleges, csak képzés és eliminációs szabálya van:

$$\frac{}{\Gamma \vdash \mathsf{0} : \mathsf{U}_0}\ \mathsf{0}\text{ képzés} \qquad \frac{\Gamma \vdash t : \mathsf{0} \quad \Gamma \vdash A : \mathsf{U}_i}{\Gamma \vdash \mathsf{ind_0}\, t : A}\ \mathsf{0}\text{ elimináció}$$

Az üres típusra azért van szükségünk, mert a logikai negáció vele fejezhető ki: a $\neg A$ állítást az $A \to \mathsf{0}$ típussal fejezzük ki.

2.4. A kételemű típus levezetési szabályai

A kételemű típus szabályainak olvasásakor a számítástudományi intuícióra érdemes hagyatkozni: a `Bool` típusnak felel meg, eliminációs szabálya egy (függő típusú) `if then else` alkalmazásának, β szabályai pedig egy elágazást tartalmazó program futtatásának. η szabálya nincsen.

$$\frac{}{\Gamma \vdash \mathsf{2} : \mathsf{U}_0}\ \mathsf{2}\text{ képz} \qquad \frac{}{\Gamma \vdash \mathsf{ff} : \mathsf{2}}\ \mathsf{2}\text{ bev}_1 \qquad \frac{}{\Gamma \vdash \mathsf{tt} : \mathsf{2}}\ \mathsf{2}\text{ bev}_2$$

$$\frac{\Gamma, x : \mathsf{2} \vdash A : \mathsf{U}_i \quad \Gamma \vdash u : A[\mathsf{ff}/x] \quad \Gamma \vdash v : A[\mathsf{tt}/x] \quad \Gamma \vdash t : \mathsf{2}}{\Gamma \vdash \mathsf{ind_2}\, u\, v\, t : A[t/x]}\ \mathsf{2}\text{ elim}$$

$$\frac{}{\mathsf{ind_2}\, u\, v\, \mathsf{tt} \equiv u}\ \mathsf{2}\ \beta_1 \qquad \frac{}{\mathsf{ind_2}\, u\, v\, \mathsf{ff} \equiv v}\ \mathsf{2}\ \beta_2$$

$\mathsf{ind_2}\, u\, v\, t$-re úgy lehet gondolni, mint `if` t `then` u `else` v.

A kételemű típus és a függő pár segítségével definiálhatjuk az összeg típust bármely két A és B típusra az alábbi módon:

$$\cdot \vdash \lambda A.\lambda B. \sum_{x:\mathsf{2}} (\mathsf{ind_2}\, A\, B\, x) : \mathsf{U}_i \to (\mathsf{U}_i \to \mathsf{U}_i)$$

Bevezetjük az alábbi rövidítést:

$$A + B :\equiv \sum_{x:2} (\mathsf{ind}_2\, A\, B\, x)$$

Az $A+B$ típus elemei vagy ff és egy A-beli elem, vagy tt és egy B-beli elem. A bal és jobb injekciókat az alábbi rövidítésekkel adhatjuk meg:

$$\mathsf{inl} :\equiv \lambda a.(\mathsf{ff}, a)$$
$$\mathsf{inr} :\equiv \lambda b.(\mathsf{tt}, b)$$

Szintén definiálható[3] rövidítésként az alábbi ind_{A+B} függvény, melyre az $A + B$ eliminátoraként gondolhatunk (ha minden $a : A$-ra tudjuk $C(\mathsf{inl}\, a)$-t és minden $b : B$-re tudjuk $C(\mathsf{inr}\, b)$-t, akkor minden $t : A + B$-re tudjuk $C\, t$-t):

$$A : \mathsf{U}_i, B : \mathsf{U}_i, C : A + B \to \mathsf{U}_j \vdash \mathsf{ind}_{A+B}$$
$$: \left(\prod_{a:A} C(\mathsf{inl}\, a) \right) \to \left(\prod_{b:B} C(\mathsf{inr}\, b) \right) \to \prod_{t:A+B} C\, t$$

$A + B$-re az $A \vee B$ diszjunkcióként gondolhatunk.

2.5. A természetes számok típusának levezetési szabályai

A természetes számokat Peano-számokként adjuk meg, két konstruktorral: zero és suc. Az eliminációs szabály a természetes számokon alkalmazott teljes indukciónak felel meg, ez alapján az analógia alapján hívjuk ind-nek az eliminátorokat.

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathsf{U}_0}\ \mathbb{N}\text{ képz} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathbb{N}}\ \mathbb{N}\text{ bev}_1 \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\, n : \mathbb{N}}\ \mathbb{N}\text{ bev}_2$$

[3] ind_{A+B} a következő kifejezést rövidíti: $\lambda ml.\lambda mr.\lambda t.\mathsf{ind}_2\, ml\, mr\, (\pi_1\, t)\, (\pi_2\, t)$, ahol a 2 elim szabályhoz szükséges A típus a következőképp van definiálva: $\prod_{v:\mathsf{ind}_2\, A\, B\, x} C(x, v)$. Érdemes a β-szabályokat is ellenőrizni.

$$\frac{\begin{array}{l}\Gamma, x : \mathbb{N} \vdash A : \mathsf{U}_i \\ \Gamma \vdash mz : A[\mathsf{zero}/x] \quad \Gamma, n : \mathbb{N}, p : A[n/x] \vdash ms : A[\mathsf{suc}\, n/x] \\ \Gamma \vdash t : \mathbb{N}\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}\, mz\, ms\, t : A[t/x]}\ \mathbb{N}\text{ elim}$$

$$\frac{}{\mathsf{ind}_{\mathbb{N}}\, mz\, ms\, \mathsf{zero} \equiv mz}\ \mathbb{N}\ \beta_1$$

$$\frac{}{\mathsf{ind}_{\mathbb{N}}\, mz\, ms\, (\mathsf{suc}\, m) \equiv ms[m/n, \mathsf{ind}_{\mathbb{N}}\, mz\, ms\, m/p]}\ \mathbb{N}\ \beta_2$$

A természetes számoknak nincs η-szabályuk.

Az összeadást pl. az alábbi kifejezéssel definiálhatjuk (az üres környezetben):

$$\cdot \vdash \lambda x.\lambda y.\mathsf{ind}_{\mathbb{N}}\, y\, (\mathsf{suc}\, p)\, x : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

Az $\mathsf{ind}_{\mathbb{N}}$ második argumentuma az n és p változókkal kibővített környezetben értelmezett, emiatt van értelme p-re hivatkozni a fenti $(\mathsf{suc}\, p)$ kifejezésben. Bevezetjük az alábbi rövidítést: $x + y :\equiv \mathsf{ind}_{\mathbb{N}}\, y\, (\mathsf{suc}\, p)\, x$.

Ha $x \equiv \mathsf{zero}$, akkor $x + y \equiv \mathsf{ind}_{\mathbb{N}}\, y\, (\mathsf{suc}\, p)\, \mathsf{zero}$, ami $\mathbb{N}\ \beta_1$ miatt definicionálisan egyenlő y-al. Ha $x \equiv \mathsf{suc}\, m$, akkor $x + y \equiv \mathsf{ind}_{\mathbb{N}}\, y\, (\mathsf{suc}\, p)\, (\mathsf{suc}\, m)$, mely a $\mathbb{N}\ \beta_2$ szabály miatt definicionálisan egyenlő $\mathsf{suc}\, (\mathsf{ind}_{\mathbb{N}}\, y\, (\mathsf{suc}\, p)\, m)$-el stb. Így gondolhatunk a program végrehajtására (további részletek a 2.9. pontban).

2.6. Egyenlőség típus

Két kifejezés egyenlőségének leírására szolgál a definicionális egyenlőség következtetés-típus, pl. $u \equiv v$. Ez az egyenlőség azonban nem szerepelhet magukban a kifejezésekben (programokban, típusokban, matematikai állításokban, bizonyításokban). A definicionális egyenlőség internalizálására vezetjük be a (propozicionális) egyenlőséget[4]. Ez egy típus, mely azt fejezi ki, hogy két kifejezés egyenlő.

Levezetési szabályai:

$$\frac{\Gamma \vdash A : \mathsf{U}_i \quad \Gamma \vdash u : A \quad \Gamma \vdash v : A}{\Gamma \vdash u =_A v : \mathsf{U}_i} =\ \text{képz} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}\, a : a =_A a} =\ \text{bev}$$

[4] A „propozicionális” jelzőt gyakran elhagyjuk.

$$\frac{\begin{array}{l}\Gamma \vdash A : \mathsf{U}_i \\ \Gamma, x : A, y : A, p : x =_A y \vdash C : \mathsf{U}_j \\ \Gamma, a : A \vdash m : C[a/x, a/y, \mathsf{refl}\, a/p] \\ \Gamma \vdash u : A \quad \Gamma \vdash v : A \quad \Gamma \vdash r : u =_A v\end{array}}{\Gamma \vdash \mathsf{J}\, C\, t\, r : C[u/x, v/y, r/p]} = \text{ elim}$$

$$\frac{}{\mathsf{J}\, C\, t\, (\mathsf{refl}\, u) \equiv t[u/a]} = \beta$$

A reflexivitás (refl) az egyetlen módja az egyenlőség bevezetésének. Az eliminációs szabályban C egy olyan típus, amely függ két A-beli elemtől (x és y), és ezek egyenlőségének bizonyításától (p). Az eliminációs szabály pedig azt fejezi ki, hogy ha C-t be tudjuk látni bármely A-beli elemre és refl-re, akkor bármilyen egyenlőségre (r-re) is be tudjuk látni. Néha $u =_A v$ helyett egyszerűen csak $u = v$-t írunk.

A definiconális egyenlőséggel való kompatibilitást az alábbi szabályok biztosítják:

$$\frac{A \equiv A' \quad u \equiv u' : A \quad v \equiv v' : A}{(u =_A v) \equiv (u' =_{A'} v')} \qquad \frac{a \equiv a'}{\mathsf{refl}\, a \equiv \mathsf{refl}\, a'}$$

A bal oldali szabályból következik, hogy ha két kifejezés definicionálisan egyenlő, akkor propozicionálisan is egyenlők (internalizálás). Fordítva ez nem igaz: pl. a fenti definícióval az $u : \mathbb{N}, v : \mathbb{N}, w : \mathbb{N}$ környezetbeli $(u + v) + w$ és $u + (v + w)$ kifejezések nem definicionálisan egyenlőek, de propozicionálisan igen (lásd lejjebb). Ha viszont tetszőleges környezetben egy egyenlőség bizonyítása egyszerűen $\mathsf{refl}\, t$ valamely t-re, akkor az egyenlőség két oldalán szereplő kifejezések definicionálisan is egyenlők. Az üres környezetben egyenlő kifejezések mind definicionálisan is egyenlők.

A $\forall_{f:A\to B}.(u = v \to fu = fv)$ tételt az alábbi módon tudjuk pl. bebizonyítani:

$$A : \mathsf{U}_i, B : \mathsf{U}_i, u : A, v : A \vdash \lambda f.\mathsf{J}\,(fx =_B fy)\,(\mathsf{refl}\,(fa)) : \prod_{f:A\to B} (u =_A v \to fu =_B fv)$$

A fenti kifejezésre bevezetjük az ap rövidítést.

Az ap segítségével bebizonyíthatjuk, hogy a természetes számok fentebb definiált összeadása asszociatív:

$$\cdot \vdash \lambda u.\lambda v.\lambda w.\mathsf{ind}_{\mathbb{N}}\,(\mathsf{refl}\,(v+w))\,(\mathsf{ap}\,\mathsf{suc}\,p)\,x$$
$$: \prod_{u:\mathbb{N}} \prod_{v:\mathbb{N}} \prod_{w:\mathbb{N}} (u+v)+w =_{\mathbb{N}} u+(v+w)$$

A bizonyítást így írhatjuk le: teljes indukcióval bizonyítunk x szerint. Az $\mathbb{N}$ elim szabályban szereplő x-től függő A típusnak $(x+v)+w =_{\mathbb{N}} x+(v+w)$-t választunk. Ha x értéke 0 ($x \equiv \mathsf{zero}$), akkor az $\mathbb{N}$ elim szabályban szereplő $mz : (\mathsf{zero}+v)+w =_{\mathbb{N}} \mathsf{zero}+(v+w)$, ennek típusa pedig a $+$ rövidítés behelyettesítésével és a $\mathbb{N}$ β_1 szabály alapján definicionálisan egyenlő $v+w =_{\mathbb{N}} v+w$-vel, amit $\mathsf{refl}\,(v+w)$ bizonyít. Ha $x \equiv \mathsf{suc}\,n$, akkor az ms környezetében levő p típusa $(n+v)+w =_{\mathbb{N}} n+(v+w)$, ez az indukciós hipotézisünk, ha erre alkalmazzuk az $\mathsf{ap}\,\mathsf{suc}$ függvényt, $\mathsf{suc}\,((n+v)+w) =_{\mathbb{N}} \mathsf{suc}\,(n+(v+w))$-ot kapunk, ami viszont a $+$ rövidítés behelyettesítésével és $\mathbb{N}$ β_2 alkalmazásával definicionálisan egyenlő $(\mathsf{suc}\,n+v)+w =_{\mathbb{N}} \mathsf{suc}\,n+(v+w)$-el, és ez az, amit az indukciós lépésben bizonyítani szeretnénk (ms típusa ez).

Egy további érdekes tétel, melyet transport-al rövidítünk, szintén egyszerűen bizonyítható J segítségével:

$$A : \mathsf{U}_i, u : A, v : A \vdash \lambda P.\lambda r.\mathsf{J}\,(\lambda x.\lambda y.\lambda p.P\,x \to P\,y)\,(\lambda a.\lambda s.s)\,r$$
$$: \prod_{P:A\to\mathsf{U}_j} u =_A v \to P\,u \to P\,v$$

transport a P tulajdonság A-ban egyenlő elemek közötti transzportálását fejezi ki: ha $u = v$, és u rendelkezik a P tulajdonsággal, akkor v is.

Szintén a J eliminátor segítségével bizonyítható, hogy egy adott A típusra $\_ =_A \_ : A \to A \to \mathsf{U}_i$ ekvivalencia-reláció, refl egységelem, és

a tranzitivitás asszociatív:

$\mathsf{refl}\, a : a = a$	reflexivitás
$\_^{-1} : a = b \to b = a$	szimmetria
$\_ \centerdot \_ : a = b \to b = c \to a = c$	tranzitivitás
$p \centerdot \mathsf{refl}\, b = p$	jobb oldali egységelem
$\mathsf{refl}\, a \centerdot p = p$	bal oldali egységelem
$(p \centerdot q) \centerdot r = p \centerdot (q \centerdot r)$	asszociativitás

adott $a, b, c, d : A$, $p : a = b, q : b = c, r : c = d$ esetén.

2.7. Általános induktív típusok

A természetes számok, bináris fák, listák stb. mind induktív típusok. Ezeket funkcionális programozási nyelvekben konstruktoraik listázásával adjuk meg, és eliminálásukra mintaillesztéssel adunk meg függvényeket. Pl. a feljebb definiált természetes számokat Haskell-ben így adjuk meg: `data Nat = Zero | Suc Nat`, Agda-ban ekképp:

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Az összes induktív típus megadható egy konténerből számított W-típusként [1]. Egy konténert az $S : \mathsf{U}_i$ alakjával és a $P : S \to \mathsf{U}_i$ pozícióival adunk meg. Az alak reprezentálja a konstruktorok halmazát a nem-rekurzív paraméterekkel együtt, míg a pozíciók a rekurzív előfordulásokat adják meg. A természetes számok, A-beli elemek listája és leveleknél L-beli elemet, elágazásoknál N-beli elemet tartalmazó bináris fák konstruktorai ill. a nekik megfelelő konténerek (1 az egyelemű típus):

$\mathsf{zero} : \mathbb{N}$	$S :\equiv 2$
$\mathsf{suc} : \mathbb{N} \to \mathbb{N}$	$P :\equiv \lambda s.\mathsf{ind}_2\, 0\, 1\, s$
$\mathsf{nil} : \mathsf{List}_A$	$S :\equiv 1 + A$
$\mathsf{cons} : A \to \mathsf{List}_A \to \mathsf{List}_A$	$P :\equiv \lambda s.\mathsf{ind}_{1+A}\, (\lambda x.0)\, (\lambda x.1)\, s$
$\mathsf{leaf} : L \to \mathsf{Tree}_{L,N}$	$S :\equiv L + N$
$\mathsf{node} : N \to \mathsf{Tree}_{L,N} \to \mathsf{Tree}_{L,N} \to \mathsf{Tree}_{L,N}$	$P :\equiv \lambda s.\mathsf{ind}_{L+N}\, (\lambda x.0)\, (\lambda x.2)\, s$

Tehát a természetes számoknál $s \equiv \mathsf{ff}$ jelképezi a zero konstruktort, $s \equiv \mathsf{tt}$ a suc konstruktort, és $P\,\mathsf{ff} \equiv 0$, tehát a zero konstruktornak nincs rekurzív paramétere, míg $P\,\mathsf{tt} \equiv 1$, tehát a suc konstruktornak egy rekurzív ($\mathbb{N}$ típusú) paramétere van. Listáknál a nil konstruktornak nincs rekurzív paramétere ($s \equiv \mathsf{inl}\,*$ jelzi, ahol $*$ az egyelemű típus eleme), míg a cons konstruktor igazából A-nyi különböző konstruktor, annak megfelelően, hogy a cons első paramétere micsoda, és függetlenül ettől az értéktől mindig 1 db rekurzív (List_A típusú) paramétere van.

Ha adott egy S, P konténer, a hozzá tartozó $\mathsf{W}\,S\,P$ típus az alábbi bevezető szabállyal rendelkezik:

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash f : P\,s \to \mathsf{W}\,S\,P}{\Gamma \vdash \mathsf{sup}\,s\,f : \mathsf{W}\,S\,P}\ \mathsf{W}\text{ bev}$$

Tehát ha egy $\mathsf{W}\,S\,P$ típusú értéket szeretnénk konstruálni, egy formára és egy függvényre van szükség, utóbbi a formának megfelelő összes pozícióra ad egy újabb $\mathsf{W}\,S\,P$ típusú értéket. Az eliminációs és egyéb szabályok is megadhatók, részletek a W-típusokat bevezető [23]-ban. A konténerek kategória-elméleti megfelelői a szigorúan pozitív funktorok [1], a W-típusok ezek iniciális algebrái (legkisebb fixpontjai). Terminális koalgebráikat M-típusoknak nevezik [8], ezek maximális fixpontoknak felelnek meg, végtelen listák pl. így írhatók le. A konténerek kiterjesztései az indexelt konténerek [28], illetve az ezekhez tartozó indexelt W-típusok, melyekkel indexelt típusok leírhatók, pl. az n-hosszú vektorok típusa, $\mathsf{Vec}_A\,n$, ahol $n : \mathbb{N}$.

2.8. Curry-Howard izomorfizmus

Az előbb megadott típuselméletetet a következőképp használhatjuk logikai érvelésre: a propozíciók halmazának U_0-t vesszük, egyéb halmazokat típusokkal értelmezünk, pl. a természetes számok halmazát az $\mathbb{N}$ induktív típussal értelmezzük. Az elsőrendű logikai összekötőket így értelmezzük:

$$
\begin{array}{rll}
\mathsf{Prop}^{\mathsf{c}} :\equiv \mathsf{U}_0 & & \text{(propozíciók halmaza)} \\
A \to \mathsf{Prop}^{\mathsf{c}} :\equiv A \to \mathsf{U}_0 & & A\text{-n értelmezett állítások} \\
 & & \text{halmaza} \\
\bot :\equiv \mathsf{0} & & \text{logikai hamis} \\
\top :\equiv \mathsf{1} & & \text{logikai igaz} \\
\neg P :\equiv P \to \mathsf{0} & & \text{negáció} \\
P \wedge Q :\equiv P \times Q \equiv \sum_{p:P} Q & & (Q\text{-ban nem szerepel } p) \\
P \vee^{\mathsf{c}} Q :\equiv P + Q \equiv \sum_{x:2} (\mathsf{ind}_2\, A\, B\, x) & & \\
P \Rightarrow Q :\equiv P \to Q \equiv \prod_{p:P} Q & & (Q\text{-ban nem szerepel } p) \\
\forall_{x \in A} P_x :\equiv \prod_{x:A} P & & \\
\exists^{\mathsf{c}}_{x \in A} P_x :\equiv \sum_{x:A} P & & \\
a = b :\equiv a =_A b & & (a \text{ és } b \text{ az } A \text{ típus elemei})
\end{array}
$$

Ez a logikai rendszer néhány fontos különbséget mutat a klasszikus matematikához képest:

- Halmazok helyett típusokat használunk, emiatt nincs értelmezve az unió és metszet művelet, hiszen ezek tetszőleges típusok között sem értelmesek. Minden kifejezésnek van típusa, nincs olyan helyzet, hogy egy elem többféle halmazba (típusba) is tartozhat, a típus valamely

kifejezésnek el nem idegeníthető és meg nem változtatható tulajdonsága (ami egyébként elősegíti nemcsak a biztonságos programozást, de a biztonságos bizonyítást és absztrakciót is).

- $\mathsf{Prop}^{\mathsf{c}} \equiv \mathsf{U}_0$ bizonyítás-relevanciával (proof-relevance) rendelkezik: ugyanannak az állításnak több különböző bizonyítása is lehet, és ezek nem kell, hogy egyenlők legyenek – a bizonyítások információt hordoznak (ennek kiküszöbölésére a típuselméletben gyakran bevezetnek egy külön $\mathsf{Prop}^{\mathsf{c}}$ univerzumot, melynek elemei definicionálisan egyenlők). Heyting nyomán egy állításra gondolhatunk úgy, mint bizonyításainak halmazára (típusára).

- A $\exists^{\mathsf{c}}$ kvantor erős (vagy konstruktív, emiatt jelöltük c-vel): ha van egy bizonyításunk arra, hogy létezik A-nak valamely P tulajdonsággal rendelkező eleme, akkor ebből a bizonyításból ezt az elemet mindig képesek vagyunk kinyerni (nem csak a puszta létezésről tudunk). Hasonlóképpen $P \vee^{\mathsf{c}} Q$ bizonyításából extrahálható, hogy P vagy Q az igaz. A klasszikus változataik megadásához homotópia-típuselmélet szükséges, lásd a 7. pont.

- Ehhez szorosan kapcsolódik, hogy a kizárt harmadik elve nincs validálva, vagyis nem igaz, hogy minden állítás eldönthető. Ha indirekten bizonyítjuk P-t, akkor valójában $\neg\neg P$-t bizonyítottuk. Mivel a dupla negáció művelet monádként [3] viselkedik, hasonlóan ahhoz, ahogy a Haskell programozási nyelvben `IO`-t kell írni a mellékhatásokat használó függvények típusa elé, itt $\neg\neg$-et kell írni a kizárt harmadik elvét használó bizonyítások típusa elé.

2.9. Metaelméleti tulajdonságok

Az ún. helyettesítési és gyengítési szabályok [10] a levezetési fákon végzett indukcióval bizonyíthatók. Hasonlóképp bizonyítható, hogy ha $a \equiv b : A$, akkor mind a-nak, mind b-nek a típusa A (tehát a definicionális egyenlőség megőrzi a típust).

A fent bevezetett formális rendszer azért használható jól programozásra, mert a típusellenőrzés eldönthető. Tehát, ha adott egy Γ környezet, t kifejezés és A típus, akkor létezik egy olyan program, mely eldönti, hogy a $\Gamma \vdash t : A$ következtetés levezethető -e[5]. Úgy is fogalmazhatunk, hogy ha van egy matematikai állításunk, és egy jelöltünk ennek bizonyítására, akkor a számítógép felismeri, hogy a bizonyítás helyes -e.

Továbbá a rendszer konzisztens, amin azt értjük, hogy nem létezik olyan t kifejezés, melyre $\cdot \vdash t : \mathsf{0}$ levezethető lenne. Ezt természetesen csak egy másik, erősebb rendszerhez képest relatíve tudjuk kijelenteni, ilyen rendszer pl. a ZFC megfelelő nagyságú kardinálisokkal.

A konzisztencia bizonyítása normalizáláson keresztül történik. Ha a definicionális egyenlőség szerint ekvivalencia-osztályokat képzünk, minden ilyen osztályból kiválasztható egy reprezentáns, az ún. normálforma, és létezik egy algoritmus, mely bármely kifejezést átalakít az ekvivalencia-osztályának reprezentására. Ezt a folyamatot normalizálásnak hívjuk. A normalizálás történhet kis lépésekben [13], nagy lépésekben [9], vagy a normalizálás kiértékeléssel technikával [5]. A normálformák úgy vannak meghatározva, hogy az üres környezetben mindig konstruktorral kezdődnek, emiatt, mivel az üres típusnak nincs konstruktora, az üres környezetben nincs $\mathsf{0}$ típusú kifejezés. Pl. természetes számok normálformái az üres környezetben $\mathsf{suc}^n\,\mathsf{zero}$, tehát a suc konstruktor n-szer alkalmazva a zero konstruktorra, ahol n egy természetes szám.

A fent megadott típusrendszerre (kivéve általános induktív típusok és az egyelemű típus) a normálformákat v adja meg (a normálformák típusozottak, ettől most eltekintünk):

[5]Ehhez az egyes kifejezéseknek több információt kell hordozniuk, mint amennyit informálisan leírunk, pl. a λ kifejezéseket fel kell díszíteni a változó típusával stb. A kényelmes, tömör írásmódból teljes információval rendelkező kifejezéseket létrehozó folyamatot elaborációnak hívjuk, további információ pl. az Agda programozási nyelv működését leíró PhD-dolgozatban [29] vagy az Epigram elaborációját leíró cikkben [25] található.

$$v ::= \mathsf{U}_i \mid \prod_{x:v} v \mid \lambda x.v \mid \sum_{x:v} v \mid (v,v) \mid 0 \mid 2 \mid \mathsf{ff} \mid \mathsf{tt} \mid \mathbb{N} \mid \mathsf{zero} \mid \mathsf{suc}\, v \mid v =_v v \mid \mathsf{refl}\, v$$

$$n ::= x \mid n\, v \mid \pi_1\, n \mid \pi_2\, n \mid \mathsf{ind}_2\, v\, v\, n \mid \mathsf{ind}_{\mathbb{N}}\, v\, v\, n \mid \mathsf{J}\, v\, v\, n$$

n az ún. neutrális kifejezéseket jelöli, x egy változó. A neutrális kifejezések bár nem konstruktorral kezdődnek, mégsem egyszerűsíthetők, mert β szabály nem alkalmazható rájuk. Zárt kifejezés nem lehet neutrális, mert nincs benne szabad változó.

3. Extenzionalitás

A fent megadott típuselméletben az egyenlőség túlságosan finom: olyan kifejezéseket is megkülönböztet, melyeket a matematikában nem szeretnénk megkülönböztetni. Ilyenek a pontonként egyenlő, de definíció szerint különböző függvények, pl. a $\lambda x.x : \mathbb{N} \to \mathbb{N}$ és a $\lambda x.x + \mathsf{zero}$ fügvények. Az extenzionalitás elve azt mondja, hogy két objektum nem lehet egyszerre (a meglevő egyenlőség használata nélkül) megkülönbözethetetlen és nem-egyenlő. Mivel függvényeket csak úgy lehet használni, hogy alkalmazzuk őket, ha tetszőleges kifejezésre alkalmazva őket, egyenlő eredményt adnak, akkor megkülönböztethetetlenek. Ennek megfelelően szeretnénk, hogy az alábbi szabály része legyen a típuselméletünknek:

$$\frac{f : \prod_{x:A} B \quad g : \prod_{x:A} B \quad t : \prod_{a:A} f\, a =_{B[a/x]} g\, a}{\mathsf{funext}\, t : f =_{\prod_{x:A} B} g}$$

Ha azonban ezt, mint új levezetési szabályt hozzáadjuk a rendszerhez, elveszítjük a normalizálást: mivel az egyenlőség β szabálya csak olyankor működik, ha az egyenlőség a refl konstruktorral lett létrehozva, ha J-t egy funext által létrehozott egyenlőségre alkalmazzuk, elakad a normálformára hozás algoritmusa. Az egyenlőség más fajta definíciójával, és egy újabb levezetési szabály, a K (lásd később) hozzáadásával megadható egy olyan típuselméletet, ami normalizáló és a függvény extenzionalitást is validálja

[2]. Ez az elmélet azonban inkonzisztens a következő extenzionalitási alapelvvel, az izomorf típusok egyenlőségével.

A és B típusok izomorfak, ha léteznek $f : A \to B$ és $g : B \to A$ függvények, melyekre igaz, hogy $\prod_{a:A} g\,(f\,a) =_A a$ és $\prod_{b:B} f\,(g\,b) =_B b$. Ha két típus izomorf, akkor ha egyikben kapunk egy elemet, áttehetjük a másikba, és tudjuk, hogy ez a lépés nem változtatja meg lényegesen az elemet, mert mindig visszakapjuk az eredetit, ha a másik irányú függvényt alkalmazzuk. Ha $A \cong B$ jelöli az A és B közötti izomorfizmusok típusát, akkor szeretnénk egy $\mathsf{isotoid} : A \cong B \to A = B$ függvényt. Ha van egy ilyenünk, ez azzal az előnnyel jár, hogy bármely műveletet vagy tulajdonságot, amivel az egyik típus rendelkezik, transzportálni tudunk a másikra. Pl. ha van egy $m : A \to A \to A$ műveletünk és egy $i : A \cong B$ izomorfizmusunk, akkor az ennek megfelelő $m' : B \to B \to B$ műveletet megkaphatjuk a 2.6 pontban megadott $\mathsf{transport}$ függvénnyel: $m' :\equiv \mathsf{transport}\,(\lambda X.X \to X \to X)\,(\mathsf{isotoid}\,i)\,m$.

Ha egy szótárt szeretnénk implementálni, megtehetjük ezt nem-hatékony módon, egy listával `(List, _::_, ...)`, vagy hatékony módon piros-fekete keresőfákkal `(RBT, insert, ...)` (mindkettő a következő egzisztenciális típus eleme: $\sum_{T:U_i}(A \to T \to T) \times ...$). A hatékony implementációról szeretnénk bizonyítani bizonyos tulajdonságokat, pl. hogy kétszer egymás után ugyanazt az `insert` parancsot végrehajtva ugyanazt kapjuk, mint ha csak egyszer tettük volna meg. A hatékony implementációról azonban nehéz formálisan érvelni. Ha viszont bizonyítunk egy izomorfizmust a hatékony és a nem-hatékony implementáció között, onnantól elég a tulajdonságokat az egyszerű implementációról belátni, és $\mathsf{transport}$ segítségével ezek mind igazak lesznek a bonyolultra is.

Ha $\mathsf{isotoid}$-t egyszerűen levezetési szabályként hozzáadjuk az elméletünkhöz, ugyanabba a problémába ütközünk, mint az előbb, elveszítjük a normalizálást. A típuselméletünk nem válik inkonzisztenssé, mert Voevodsky szimpliciális halmaz modellje [20] validálja ezeket a levezetési szabályokat.

Az extenzionális tulajdonságok intenzionális típuselméletbe való beillesztéséről szól [16] (az izomorfizmusokra nem tér ki).

4. Típusok mint topologikus terek

Hagyományosan a típusokra halmazokként gondolunk, és a Curry-Howard izomorfizmus (2.8. pont) sem változtat ezen, hiszen az egy típust a neki megfelelő állítás bizonyításainak halmazaként értelmezi.

Az egyenlőség típus nem úgy viselkedik, ahogyan azt egy halmaztól elvárnánk. A természetesnek tűnő szabály, miszerint ha van két bizonyításunk a és b egyenlőségére, akkor ez a két bizonyítás (propozicionálisan) egyenlő, a Martin-Löf típuselméletben nem bizonyítható. Ezt Hofmann és Streicher mutatták meg groupoid modelljükkel [18] (a groupoid egy olyan kategória, melyben minden morfizmus izomorfizmus [3]). Ez a szabály ekvivalens az egyenlőség típus ún. K eliminációs szabályával, mely újabb levezetési szabályként hozzávehető a típuselmélethez, és mellé tehető egy β szabály is, amellyel a típuselmélet normalizáló marad. Hogy az induktív típusként megadható egyenlőség normális eliminációs szabálya (J) miért nem elégséges az egyenlőség leírására, sokáig a típuselmélet egy titokzatos tulajdonsága volt.

A groupoid modellt Awodey [4] és tőle függetlenül Voevodsky [34] fejlesztette tovább a típusokat topologikus terekként értelmező modellé. Ha egy típust topologikus térként értelmezünk, a típus elemeit pedig annak pontjaiként, akkor egy $a = b$ típusú kifejezés egy a és b közötti útnak felel meg. Az, hogy $p, q : a = b$ esetén nem feltétlenül igaz, hogy $p = q$, annak felel meg, hogy két út nem feltétlenül egyenlő, vagyis nem feltétlenül igaz, hogy létezik egy homotópia p és q között.

A topologikus tér és topologikus terek közötti folytonos függvény definícióját nem adjuk meg, bármely topológia könyvben megtalálhatóak ezek a fogalmak. Egy térre intuitíve gondolhatunk úgy, mint pl. egy háromdimenziós objektum belsejére (vagy felszínére stb.), folytonos függvényre pedig mint egy vonalra, melyet ceruzánk felemelése nélkül megrajzolhatunk.[6] Ezek általánosításai a topológiai alapfogalmak. Igazából nem is érdekes, hogy pontosan mik a definíciók, mert a típuselméletnek egyfajta „szintetikus” topológia felel meg, ahol az alapfogalmak definíciója absztrakt. Emiatt

[6] Főként $[0, 1]^n$ értelmezési tartományú függvényekkel fogunk foglalkozni, emiatt az analógia elég jól használható.

minden definiálható függvény folytonos, és nem tudjuk a terek topológiáját megváltoztatni.

Definíció 4.1 (Homotópia (topológiában)) *X, Y topologikus terek, az $f, g : X \to Y$ folytonos függvények közötti homotópia egy folytonos $h : X \times [0,1] \to Y$ függvény, melyre minden $x : X$-re $h\,(x, 0) = f\,x$ és $h\,(x, 1) = g\,x$. Jelölés: $h : f \sim g$.*

Egy homotópiára úgy gondolhatunk, mint f képének folytonos deformálására g képébe.

Egy X topologikus tér két pontja megadható mint $a : \mathsf{1} \to X$ és $b : \mathsf{1} \to X$ (folytonos) függvények. Egy a és b közötti homotópia így egy $f : \mathsf{1} \times [0,1] \to X$ folytonos függvény lesz, melynek típusa izomorf[7] $[0,1] \to X$-el, és így $f\,0 = a$ és $f\,1 = b$ (ahol pongyolán a-t írunk $a\,*$ helyett, hiszen $(\mathsf{1} \to X) \cong X$). Típuselméletben f megfelelője egy $f : a = b$ kifejezés. Ezzel adja magát a következő definíció, mely megfelel a topológiai definíciónak.

Definíció 4.2 (Homotópia (típuselméletben)) *$f, g : \prod_{x:A} B$ függvények közötti homotópiát*

$$f \sim g :\equiv \prod_{x:A} (f\,x = g\,x)$$

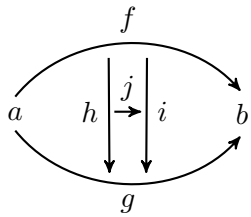
definiálja.

Ha adott két folytonos függvény, $f, g : [0,1] \to X$, melyek 0-ban a-t, 1-ben b-t vesznek föl, akkor az f és g közötti homotópia egy $h : [0,1]^2 \to X$ folytonos függvény, melyre minden $z : [0,1]$-re $h\,(z, 0) = f\,z$ és $h\,(z, 1) = g\,z$. h típuselméleti megfelelője egy $h : (a, b, f) =_{\sum_{x,y:X} x=y} (a, b, g)$, ami nem kifejezetten érdekes (J kétszeri alkalmazásával bármely két ilyen hármas triviálisan egyenlő lesz). Mi lesz tehát egy típuselméletbeli magasabb egyenlőség, $h' : f = g$ topológiai megfelelője?

[7] A matematikában, mint ebben az érvelésben is, gyakran használjuk az „izomorf típusok egyenlőek” alapelvet. Típuselméleti megfelelőjéért lásd a a 6. pontot.

Definíció 4.3 (Relatív homotópia (topológiában)) *X, Y topologikus terek, az $f, g : X \to Y$ folytonos függvények közötti, $A \subseteq X$ halmazhoz relatív homotópia egy olyan $h : f \sim g$, hogy minden $a : A$-ra a $h\,(a, t)$ érték független t-től.*

Természetesen f és g között relatív homotópia csak akkor adható meg, ha $f\,a = g\,a$ minden $a : A$-ra. Egy relatív homotópiára úgy gondolhatunk, mint f képének folytonos deformálására g képébe úgy, hogy közben az A-beli pontok fixen maradnak. Az előbbi f és g közötti $h' : f\ g$, $\{0, 1\}$-re relatív homotópia típuselméletben egy $h' : f = g$ kifejezésnek felel meg.



1. ábra. Egyenlőségek közötti egyenlőségek.

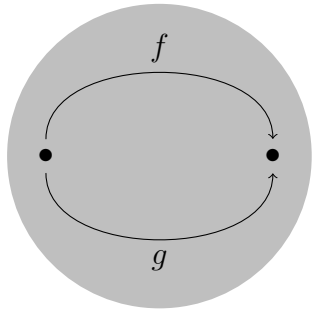
A 1. ábrán látható elrendezésben levő egyenlőségeket típuselméletben ill. topológiában így jelölhetjük:

$a, b : X$	$a : 1 \to X, b : 1 \to X$
$f, g : a =_X b$	$f, g : a \sim b$
$h, i : f =_{a=b} g$	$h, i : f \sim g$, melyek $\{0, 1\}$-re relatívok
$j : h =_{f=g} i$	$j : h \sim i$, mely $\{(t, 0) \mid t \in [0, 1]\} \cup \{(t, 1) \mid t \in [0, 1]\}$-re relatív

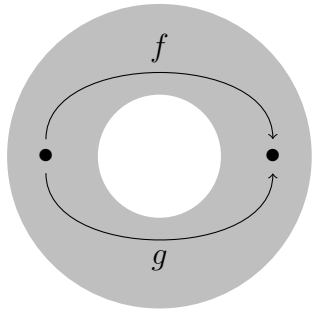
A 2. ábra egy olyan teret (korong) ábrázol, melyben minden pont között van út (minden pont egyenlő), és bármely két út között van $[0, 1]$-re relatív homotópia (bármely két egyenlőség, pl. az ábrán f-el és g-vel jelölt is, egyenlő).

A 3. ábra egy olyan teret (gyűrű) ábrázol, melyben bár minden pont között van út (minden pont egyenlő), de az ábrán jelölt f, g utak között

nincs $[0,1]$-re relatív homotópia ($f = g$-nek nincs eleme).



2. ábra. Korong.



3. ábra. Gyűrű.

A $\mathsf{refl}\, a : a =_X a$ kifejezésnek a konstans út ($\lambda t.a : [0,1] \to X$) felel meg; a $p : a = b$-ből kapott $p^{-1} : b = a$ egyenlőségnek a p-nek megfelelő homotópia és a $\lambda t.1-t : [0,1] \to [0,1]$ függvény kompozíciója; a tranzitivitás az alábbi homotópiával adható meg (ha adott f, g, melyekre $f\,1 = g\,0$):

$$(f \centerdot g)(z) :\equiv \begin{cases} f(z * 2) & z < \frac{1}{2} \\ g\left(\left(z - \frac{1}{2}\right) * 2\right) & \text{egyébként} \end{cases}$$

5. Homotópia-szintek

Léteznek olyan típusok, melyekre teljesül, hogy bizonyos szinttől kezdve az egyenlőségeik triviálisak. Pl. a fentebb mutatott korong típus bármely két pontja között van egy egyenlőség, és ezek mind egyenlők. A gyűrű típus bármely két pontja között van egyenlőség, de azok nem feltétlenül egyenlők, de ha igen, akkor csak egyféleképpen lehetnek azok. Azok a típusok, melyeket halmaznak tekinthetünk, azzal a tulajdonsággal rendelkeznek, hogy két pont között vagy van egyenlőség, vagy nincs, de két nem-egyenlő egyenlőség nem fordulhat elő. A fentieket általánosítják a homotópia-szintek.

Definíció 5.1 (Kontraktibilitás) *Egy X típus kontraktibilis, ha az*

$$\mathsf{isContr}\, X :\equiv \sum_{x:X} \left(\prod_{x':X} x = x' \right)$$

típusnak van eleme.

Definíció 5.2 (Homotópia-szint) *Ha $n \geq -2$, a homotópia szinteket a következőképp definiáljuk:*

$$\begin{aligned}
&\mathsf{is}\text{-}n\text{-}\mathsf{type} : \mathsf{U}_i \to \mathsf{U}_i \\
&\mathsf{is}\text{-}(-2)\text{-}\mathsf{type} :\equiv \mathsf{isContr} \\
&\mathsf{is}\text{-}(n+1)\text{-}\mathsf{type} :\equiv \lambda X. \prod_{x,x':X} \mathsf{is}\text{-}n\text{-}\mathsf{type}\,(x = x')
\end{aligned}$$

Tehát a -2. homotópia szinten vannak a kontraktibilis típusok. A -1. szinten levő típusokat propozícióknak nevezzük: 0 vagy 1 elemük van. Az ilyen típusok csak annyi információt hordoznak, hogy van -e elemük vagy nincsen (bebizonyíthatóak -e vagy sem); ők felelnek meg a matematikai logikában szokványos (bizonyítás-irreleváns) állításoknak. Ha a Curry-Howard izomorfizmust ilyen állításokra szorítjuk meg, a klasszikus logikához közelebb álló logikát kapunk, mint a 2.8. pontban megadott

logika. Ehhez szükséges a logikai összekötők propozicionális csonkítása, lásd a 7. pontban. Egyébként maga $\mathsf{is}\text{-}n\text{-}\mathsf{type}\, X$ minden X típusra propozicionális. A homotópia-szintek kumulatívak, tehát ha egy X típusra $\mathsf{is}\text{-}n\text{-}\mathsf{type}\, X$ igaz, akkor $\mathsf{is}\text{-}(n+1)\text{-}\mathsf{type}\, X$ is.

A 0. szinten levő típusokat halmazoknak nevezzük. Az 1. szinten levőket groupoidoknak, a 2. szinten levőket 2-groupoidoknak stb.

Ezen szintek rendszere csak egy új fajta csoportosítása a Martin-Löf típuselmélet típusainak, egyelőre nem vezettünk be semmilyen új levezetési szabályt.

Az előbbi példák közül a korong típus kontraktibilis, -2. szinten levő típus. A gyűrű groupoid, az egyenlőségeinek egyenlőségei propozíciók. A legtöbb, programozásban használt adattípus (pl. 2, $\mathbb{N}$, természetes számok listája) halmaz, melynek egyenlőségei propozicionálisak. Bármely, konténerből W-val képzett típus is halmaz. Bizonyítható, hogy Π és Σ megőrzik a homotópia-szinteket: ha $\mathsf{is}\text{-}n\text{-}\mathsf{type}\, X$ és $\prod_{x:X} \mathsf{is}\text{-}n\text{-}\mathsf{type}\, Y$, akkor $\mathsf{is}\text{-}n\text{-}\mathsf{type}\, (\sum_{x:X} Y)$; amennyiben $\prod_{x:X} \mathsf{is}\text{-}n\text{-}\mathsf{type}\, Y$, akkor $\mathsf{is}\text{-}n\text{-}\mathsf{type}\, (\prod_{x:X} Y)$ (nem kell, hogy X is n-szintű legyen). Ezzel belátható, hogy az összes, U_0-beli típus halmaz (ha magasabb induktív típusokat nem engedünk meg, lásd 7. pont).

A homotópia-szinteket nem szabad összetéveszteni az univerzum-szintekkel. Ez két, majdnem teljesen független dimenzió: a homotópia-szintek a magasabb egyenlőségekkel kapcsolatosak, míg az univerzumok a Russel-paradoxon és változatainak elkerülésére lettek bevezetve.

Az eddig bemutatott összes típus (a gyűrűn kívül, melyet még formálisan nem definiáltunk) halmaz. Magasabb egyenlőségekhez meg kell változtatunk az eddig használt Martin-Löf típuselméletet: a homotópia-típuselmélet új levezetési szabályokkal bővíti a típuselméletet, az ekvivalenciákra vonatkozó unvalenciával és a magasabb induktív típusokra vonatkozó szabályokkal. A homotópia-típuselmélet matematikusoknak szóló, összefoglaló tankönyve, mely az összes, ebben az írásban taglalt szempontra kitér, [31].

6. Ekvivalencia

Ahogy a 3. pontban írtuk, szeretnénk, ha izomorf típusok egyenlőek lennének. A két típus közötti izomorfizmusok típusa azonban nem propozicionális, emiatt egy további koherencia feltétellel egészítjük ki azt, s így jutunk az ekvivalencia definíciójához, mely a homotópia-ekvivalencia definíciójára hajaz:

Definíció 6.1 (Ekvivalencia) *Adott $A, B : \mathsf{U}_i$. Azt mondjuk, hogy egy $f : A \to B$ függvény ekvivalencia, ha*

$$\mathsf{isEquiv}\, f :\equiv \sum_{g:B\to A} \sum_{\alpha:g\circ f\sim id_A} \sum_{\beta:f\circ g\sim id_B} \mathsf{ap}\, f\,(\alpha\, x) = \beta\,(f\, x)$$

ahol id_X az X típuson értelmezett $\lambda x.x$ identikus függvény, $\_ \circ \_$ pedig a szokásos függvény-kompozíció.

Az alábbi rövidítést használjuk:

$$A \simeq B :\equiv \sum_{f:A\to B} \mathsf{isEquiv}\, f$$

$A \simeq B$ bármely A, B-re propozicionális típus. Ha van egy izomorfizmusunk, mindig képezhetünk belőle egy ekvivalenciát, tehát a fenti ötös ötödik tagját megkaphatjuk az első négyből.

Minden $A, B : \mathsf{U}_i$-re definiálható egy $\mathsf{idtoeqv} : A = B \to A \simeq B$ függvény. Az ekvivalencia $f : A \to B$ tagja $\mathsf{transport}\,(\lambda x.\mathsf{U}_i)$-ként adható meg, az egyenlőségek J-vel bizonyíthatók.

Definíció 6.2 (Univalence) *Azt mondjuk, hogy egy U_i univerzum univalens, ha*

$$\prod_{A,B:\mathsf{U}_i} \prod_{p:A=B} \mathsf{isEquiv}\,(\mathsf{idtoeqv}\, p)$$

Más szavakkal: minden $A, B : \mathsf{U}_i$-re $(A = B) \simeq (A \simeq B)$.

Ha a Martin-Löf típuselmélethez axiómaként hozzávesszük, hogy minden U_i univerzum univalens, nem csak az izomorf típusok egyenlőségét, hanem a függvény extenzionalitást is validáljuk [31] a 3. pontban megadott extenzionalitással kapcsolatos tulajdonságok közül.

Ahogyan induktív típusokra a refl konstruálja az egyenlőségeket, az univerzumokra a univalence teszi ugyanezt. Ezzel új egyenlőségeket vezet be a meglévők mellé az univerzumokban: a $2 \simeq 2$ típusnak két eleme van, az egyikhez $f = id_2$, a másikhoz $f = \lambda x.not\, x$ tartozik, ahol a *not* a boolean negáció. Ez a két izomorfizmus nem egyenlő, ha egyenlő lenne (ahogy az K-ből következne[8]), ellentmondásra jutnánk.

A univalence axióma az alábbi, naív kizárt harmadik elvével is inkompatibilis:

$$\mathsf{LEM}_\infty :\equiv \prod_{A:\mathsf{U}_i} A + \neg A$$

Azonban, ha az eldönthetőséget megszorítjuk a propozíciókra, ahogyan a klasszikus matematikában teszik, kompatibilis axiómát kapunk, ami lehetővé teszi a klasszikus érvelést:

$$\mathsf{LEM} :\equiv \prod_{A:\mathsf{U}_i} \mathsf{is\text{-}}(-1)\text{-}\mathsf{type}\, A \to A + \neg A$$

7. Magasabb induktív típusok

Az induktív típusokat (2.7. pont) konstruktoraikkal adjuk meg; ha terekként értelmezzük őket, akkor a konstruktorok pont-konstruktoroknak felelnek meg, és természetesen adódik az általánosítás, hogy lehessen út-konstruktorokat (egyenlőség-konstruktorokat) is megadni. Az $\mathbb{S}^1$ típust pl. az alábbi konstruktorok adják meg:

$\mathsf{base} : \mathbb{S}^1$

$\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$

[8] Emiatt K inkompatibilis a univalence axiómával.

Ez a korábban bemutatott gyűrű (vagy kör) típus, melynek egy pontja van (base), és egy nemtriviális (nem-refl) hurok base-ből base-be.

Magasabb induktív típusoknak azokat az induktív típusokat nevezzük, melyeknek (magasabb) egyenlőség-konstruktoraik is vannak. A konténerek elmélete egyelőre még nincs kiterjesztve a magasabb induktív típusokra, és általánosságban nem adható meg egy ilyen típus eliminátora, de $\mathbb{S}^1$-nek pl. a következő az eliminátora:

$$\frac{\begin{array}{l}\Gamma, x : \mathbb{S}^1 \vdash P : \mathsf{U}_i \\ \Gamma \vdash b : P[\mathsf{base}/x] \\ \Gamma \vdash l : \mathsf{transport}\, P\, \mathsf{loop}\, b =_{P[base]} b \\ \Gamma \vdash t : \mathbb{S}^1\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}\, b\, l\, t : P[t/x]}$$

Tehát, ha adott a P család egy b eleme base-nél, és egy l egyenlőség b és b között, mely P-ben loop fölött helyezkedik el, akkor bármely $t : \mathbb{S}^1$-re kapunk egy $P[t/x]$ elemet.

A β szabályok $ind_{\mathbb{S}^1}\, b\, l\, \mathsf{base} \equiv b$ és $\mathsf{apd}\, (\lambda x.\mathsf{ind}_{\mathbb{S}^1}\, b\, l\, x)\, \mathsf{loop} = l$. A második szabály egy propozicionális számítási szabály, mely az ap függvény függő típusú függvényekkel is működő változatával van megadva.

A magasabb induktív típusok arra is használhatók, hogy egy meglevő típust valamilyen homotópia-szintre csonkítsunk. Pl. a propozicionális csonkítás propozíciót képez valamely típusból, tehát azon kívül, hogy a típusnak van -e eleme, minden információt elfelejt, melyet a típus eredetileg hordozott.

Definíció 7.1 (Propozicionális csonkítás) *Adott A típusra az $||A||$ típus egy magasabb induktív típus az alábbi konstruktorokkal:*

$$|\_| : A \to ||A||$$
$$\mathsf{prop\text{-}eq} : \prod_{x,y:||A||} x =_{||A||} y$$

Ebből a típusból akkor tudunk eliminálni (akkor tudunk információt kinyerni belőle), ha biztosítjuk, hogy a g függvény, melyet erre használunk,

nem tesz különbséget a típus különböző elemei között:

$$\frac{\begin{array}{l}\Gamma, x : ||A|| \vdash P : \mathsf{U}_i \\ \Gamma \vdash g : \prod_{x:A} P \\ \Gamma, a, a' : ||A||, u : P[a/x], u' : P[a'/x] \vdash q : \mathsf{transport}\, P\, (\mathsf{prop\text{-}eq}\, a\, a')\, u = v \\ \Gamma \vdash t : ||A||\end{array}}{\Gamma \vdash \mathsf{ind}_{||A||}\, g\, q\, t : P[t/x]}$$

q azt fejezi ki, hogy a P indexelt család tagjai mind egyenlőek: tetszőleges $u : P[a/x]$-re és $u' : P[a'/x]$-ra u egyenlő u'-vel, de mivel a típusuk különbözik, transzportálnunk kell közöttük.

A propozicionális csonkítással kifejezhető a klasszikus diszjunkció és a klasszikus egzisztenciális kvantor, így a 2.8. pontban megadott logikai összekötők kiegészíthetők az alábbiakkal:

$$\begin{aligned}\mathsf{Prop} &:\equiv \sum_{P:\mathsf{U}_i} (\mathsf{is\text{-}}(-1)\mathsf{\text{-}type}\, P) \\ P \vee Q &:\equiv ||P + Q|| \\ \exists_{x \in A} P_x &:\equiv ||\sum_{x:A} P||\end{aligned}$$

Mivel a függvény típusnál elég, ha az értékkészlet propozíció, a függvény típus maga is propozíció lesz, azt nem szükséges csonkítani ahhoz, hogy jól működjön a homotópia-szinttel megadott propozíciókkal.

A magasabb induktív típusok egy további alkalmazása a típus valamely ekvivalencia-reláció szerinti osztályfelbontása; ennek az extrém használata a propozicionális csonkítás is, ahol mindent egy osztályba sorolunk. A természetes számok használatával az egész számok pl. az alábbi konstruktorokkal jellemzett magasabb induktív típusként adhatók meg:

$$\begin{aligned}&\mathsf{minus} : \mathbb{N} \to \mathbb{N} \to \mathbb{Z} \\ &\mathsf{quot} : \prod_{a,b,c,d:\mathbb{N}} a + d = c + b \to \mathsf{minus}\, a\, b =_{\mathbb{Z}} \mathsf{minus}\, c\, d \\ &\mathsf{set} : \prod_{(x,y:\mathbb{Z})} \prod_{(p,q:x=_{\mathbb{Z}}y)} p =_{x=_{\mathbb{Z}}y} q\end{aligned}$$

Az utolsó konstruktor azt biztosítja, hogy nincsenek magasabb egyenlőségeink, tehát $\mathbb{Z}$ egy halmaz.

8. Homotópia-típuselmélet

A homotópia-típuselmélet az intenzionális Martin-Löf típuselmélet 2. pontban ismertetett szabályai mellé hozzáveszi a univalence axiómát (6. pont) és a magasabb induktív típusok bevezetését lehetővé tevő szabályokat (melyek még nincsenek formalizálva, néhány példa a 7. pontban). Ezzel a típuselméleten belül használható (propozicionális) egyenlőség kényelmesebbé válik: pontonként egyenlő függvények egyenlőek, ahogyan izomorf típusok is azok. A propozíciók, bizonyítás-releváns állítások vagy halmazok külön homotópia-szintekbe különülnek, és elkülöníthető az információt nem hordozó, klasszikus $\exists$ kvantor a konstruktív Σ-tól; tehát probléma nélkül használható a klasszikus érvelés: ha azt bizonyítjuk, hogy két egész számnak létezik legnagyobb közös osztója, akkor a konstruktív Σ-t használjuk, ha analízist formalizálunk, a klasszikus (propozicionálisan csonkított) $\exists$-et.

A homotópia-típuselmélet szimpliciális halmaz modellje [20] által tudjuk, hogy ezek az új levezetési szabályok nem teszik inkonzisztenssé a típuselméletet. Azonban a 3. pontban leírtakhoz hasonlóan a típuselmélet elveszíti normalizáló tulajdonságát. Konstruktív modellek használatával [6] a normalizálás visszanyerhető, és reményeink szerint a közeljövőben a típuselméletet közvetlenül is ki tudjuk olyan szabályokkal egészíteni, melyek normalizálóvá teszik azt. Ehhez az egyenlőség 2.6. pontbeli definíciója helyett valószínűleg egy rekurzív definícióra van szükség, mely típuskonstruktoroktól függően határozza meg, hogy az adott típus egyenlősége hogyan van megadva: pl. függvények egyenlősége pontbeli egyenlőség, párok egyenlősége egyenlőségek párja, univerzumok egyenlősége ekvivalencia stb.

További részletek a homotópia-típuselmélet összefoglaló tankönyvében [31] és a `http://homotopytypetheory.org` weboldalon találhatók.

Hivatkozások

[1] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[2] Thorsten Altenkirch, Conor Mcbride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–58. ACM, 2007.

[3] S. Awodey. *Category Theory*. Oxford Logic Guides. OUP Oxford, 2010.

[4] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. September 2007.

[5] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[6] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. Unpublished, 2013.

[7] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[8] Venanzio Capretta. Coalgebras in functional programming and type theory. *Theoretical Computer Science*, 412(38):5006–5024, 2011. CMCS Tenth Anniversary Meeting.

[9] James Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, January 2009.

[10] Zoltán Csörnyei. *Bevezetés a típusrendszerek elméletébe*. ELTE Eötvös Kiadó, 2012.

[11] J. Y. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. 63:63–92, 1971.

[12] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theor. Comput. Sci.*, 45(2):159–192, 1986.

[13] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

[14] Georges Gonthier. A computer-checked proof of the Four Colour Theorem. 2005.

[15] Robert Harper. *Practical Foundations for Programming Languages*. December 2009.

[16] M. Hofmann. *Extensional Concepts in Intensional Type Theory*. Thesis. University of Edinburgh, Department of Computer Science, 1995.

[17] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[18] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[19] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[20] Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. The simplicial model of univalent foundations, 2012. arXiv:1211.2851.

[21] Nicolai Kraus. Non-normalizability of Cauchy sequences. Unpublished, 2014.

[22] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[23] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

[24] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[25] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.

[26] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[27] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988.

[28] Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

[29] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[30] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas J ohnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

[31] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[32] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, September 19-23, 1983*, pages 513–523. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1983.

[33] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction*. Studies in logic and the foundations of mathematics. North-Holland, 1988.

[34] Vladimir Voevodsky. A very short note on the homotopy λ-calculus. `http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf`, 2006.