EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

CSÖRNYEI ZOLTÁN

FUNKCIONÁLIS PROGRAMNYELVEK IMPLEMENTÁCIÓJA

III. RÉSZ

TÍPUSELMÉLET[1]

BUDAPEST, 2003

[1] Részben az OTKA T037742 támogatásával

Tartalomjegyzék

3. FEJEZET

Bevezetés

3.1. A típus kialakulása

A típus csak egy illúzió...

Kezdetben minden egyforma, egységes volt, típus nélküli volt a világ. Ez máris egy érdekes ellentmondás, mert mit jelent az, hogy nincs semminek sem típusa? Azt, hogy mindennek azonos a típusa, azaz, hogy van egy típus, de csak egy típus van, és mindennek ez az egy típus a típusa.

Ilyen homogén rendszer például a számítógép memóriája, ahol csak bitek vannak, minden bit felveheti a 0 vagy az 1 értéket, és a bitekre csak a logikai alapműveleteket értelmezzük. Vagy például ilyen a λ-kalkulus, ahol minden egy λ-kifejezés, és csak a kifejezések átalakítására adunk meg szabályokat.

És akkor az ember egyes bitkombinációkhoz már ismert fogalmakat rendel, bizonyos bitekre azt mondja, hogy ez a nulla, másokra azt, hogy ez az 1, 2, vagy a 49 természetes szám. Egészen más bitcsoportra meg azt, hogy ez az összaadás művelete, egy másik meg a szorzásé. Ugyanakkor a 65 szám bitkombinációja lesz az „A" betű, a 66 a „B", két ilyen bitkombináció egymás mellé írva pedig egy stringet jelent. Ezután ha újabb, hosszabb, nulla bitekből álló bitsorozatot jelölünk ki, akkor ez lesz a nulla lebegőpontos szám, mások meg az 1, 2 lebegőpontos számok.

A λ-kalkulusban hasonlóan, egy kifejezéshez hozzárendelhetjük a nulla számot, egy másikhoz az 1-et, a harmadikhoz a 2-t, egy E kifejezésre pedig azt mondhatjuk, hogy ez az összeadás művelete, mert az E 0 2 applikáció éppen a 2-nek megfeleltetett kifejezést adja eredményül. Egészen más kifejezésekre is megtehetjük ezt a hozzárendelést, ekkor egy másik „számrendszert” kapunk. Vagy kiválaszthatunk három λ-kifejezést úgy, hogy az elsőre a másodikat alkalmazva a harmadikat, az elsőre a harmadikat applikálva a másodikat kapjuk eredményül, és ekkor azt mondjuk, hogy az első a *not*, függvény, a második a *true*, a harmadik a *false* érték. Ez utóbbi kettőt akár fel is cserélhetjük.

A homogén rendszerből kiválasztott elemek közül bizonyosak tehát a kijelölt műveletek szempontjából hasonlóan viselkednek. Az azonos viselkedésű elemeket egy halmazba fogjuk össze, és ez a halmaz alkotja a típust. *A típus tehát értékek halmaza.* Ezért azt mondhatjuk, hogy „egy érték egy típus egy eleme”, „egy érték egy típushoz tartozik”. A típus értékhalmazának egy megkülönböztető nevet adhatunk, és azt is mondhatjuk, hogy a halmaz elemei ilyen típusúak.

3.1.1. Példa. *(Típusok)*

A true és false értékek alkotják a *Bool* típust.

A $0, 1, 2 \ldots$ számok a *Nat* nevű típus elemei. □

Látszik tehát, hogy a típus csak illúzió, a típusnélküli világot csak mi képzeljük típusosnak. Ez a típusos világ azonban megvédi az alatta levő nem típusos világot a hibás, vagy a nem jó használattól. A típusrendszer feltörése, például egy *Integer* típusú szám *Pointer*-ként való használata, gyakran nagyon kellemetlen következményekkel jár.

3.2. A típus szerepe

A programnyelvekben típust alapvetően két célból használunk,

- bizonyos programtulajdonságok leírásának eszközeként,
- és a programok biztonságos működésének szavatolására.

Típusok használatával nem csak egyes programelemek struktúráját tudjuk leírni, például azt, hogy egy változó karaktersorozatokból álló tömb, hanem egyes programelemek működését is, például megadhatjuk azt, hogy egy függvény inputja egy egész szám, eredménye pedig logikai értékek listája.

Az ilyen tulajdonságok leírhatósága a program jobb struktúráltságán kívül a jobb megértést és a könnyebb programírást is lehetővé teszi. Ezenkívül a fordítás hatékonyságát is növeli, bár a típusok fordításban való felhasználásának a következő hatását is figyelembe kell venni. A polimorfikus típusok használata lehetővé teszi azt, hogy ugyanazt a programkódot különböző típusú adatokra használjuk, az absztrakt adattípus használata pedig a kód struktúrálását segíti. De mivel a típussal kapcsolatos vizsgálatokat fordítási időben kell végezni, a fordítóprogram csak véges időben eldönthető problémák vizsgálatával foglalkozhat, azaz egy típusrendszernek olyannak kell lennie, hogy benne a nem-terminálás ne fordulhasson elő.

A Curry-Howard izomorfizmus lehetővé teszi azt, hogy a matematikai logika módszereit és eredményeit a típuselméletben is felhasználhassuk. Ezzel a témakörrel egy későbbi fejezetben foglalkozunk.

Azokat a nyelveket, amelyekben a változóknak típusuk van, *típusos nyelveknek* nevezzük. Egy nyelv lehet típusos úgy, hogy a típus a nyelv szintaxisának része, ezeket a nyelveket *explicit típusos nyelveknek* nevezzük, de egy lehet típusos úgy is, hogy a típusfogalom a szintaxisban egyáltalán nem fordul elő. Ezek az *implicit típusos nyelvek.*

Explicit típusos nyelv például a Haskell, Clean, implicit típusos nyelvre példa az ML. Implicit típusos nyelvek esetén a típus bevezetése és a típushozzárendelés a fordítóprogram feladata.

Azok a programnyelvek, amelyekben a változók értékkészletének nem adunk meg halmazokat, azaz a változókhoz nem rendelünk típust, a *nem típusos nyelvek* közé tartoznak. Ezekben a nyelvekben egy műveletet tetszőleges argumentumra elő lehet írni, és az eredmény is tetszőleges „típusú" értéket felvehet, az eredmény lehet például egy eltérülés, vagy más run-time hiba is. A típusnélküli λ-kalkulus a nem típusos nyelvek egy olyan tipikus példája, ahol a műveletknek ez a hibára utaló eredménye nem fordul elő, itt csak egy „művelet" van, az applikáció, és ha az applikáció első tagja nem függvény, akkor azt mondjuk, hogy a kifejezés nem változik, azaz a kifejezés normál formában van.

4. FEJEZET

Formális típusrendszerek

4.1. Formális matematikai rendszer

A *formális matematikai rendszer* szimbólumok és a rájuk alkalmazható szabályok rendszere.

4.1.1. Definíció. Formális matematikai rendszer:

A formális matematikai rendszer négy komponensből áll:

1. *Az ábécé, azaz az alapszimbólumok halmaza. Az alapszimbólumokból álló véges sorozatokat formuláknak nevezzük.*
2. *Nyelvtani szabályok halmaza. Azokat a formulákat, amelyek megfelelnek a nyelvtani szabályoknak, jól formált formuláknak nevezzük.*
3. *Axiómák. A jól formált formulák egy speciális halmazáról feltesszük, hogy jelentéssel bírnak, ezek az axiómák.*

4. Levezetési szabályok. A levezetési szabályok felhasználásával az axiómáktól különböző formulákról bizonyíthatjuk be, hogy van jelentésük.

4.1.2. Példa. *(Egy G grammatika és a fomális matematikai rendszer)*
Egy $G = (T, N, S, \mathcal{P})$ grammatika által meghatározott mondatformák leírhatók egy formális matematikai rendszerrel. Az ábécé a $T \cup N$ halmaz, a nyelvtani szabályok halmaza üres, az S szimbólum az egyetlen axióma, és $\mathcal{P}$ a levezetési szabályok halmaza.

Ebben a rendszerben tehát minden szimbólumsorozat jól formált, és az S-ből a $\mathcal{P}$-beli szabályok alkalmazásával levezethető szimbólumsorozatoknak, a mondatformáknak van jelentésük. □

A következő pontban látni fogjuk, hogy a formális típusrendszer is megadható úgy, mint egy formális matematikai rendszer (4.2.10. példa).

4.2. Formális típusrendszerek

Egy formális típusrendszer megadásához először meg kell adni a kifejezések *szintaktikáját.* A szintaktika két részből áll, az egyik rész a típusok kifejezéseinek, a másik rész a λ-kifejezéseknek a szintaktikáját írja le. A szintaktika környezetfüggetlen grammatikával írható le.

A továbbiakban a *típuskifejezéseket*, vagy röviden a *típusokat* az $A, B, \ldots$ betűkkel, a λ-kifejezéseket továbbra is az $E, F, \ldots$ betűkkel jelöljük. A *típusváltozók* jelölésére az $\alpha, \beta, \ldots$ betűket, a λ-kifejezések változóinak jelölésére most is az $x, y, \ldots$ betűket használjuk.

A szintaktikusan helyes típusoknak és a szintaktikusan helyes kifejezéseknek külön nevet is adhatunk, a típusazonosságot is a $\equiv$ jel jelöli.

4.2.1. Definíció. Jól formált típus:

Egy típus jól formált, ha szintaktikusan helyes, azaz megfelel a típus szintaktiktikájának leírásában megadott szintaktikai szabályoknak.

4.2.2. Definíció. Jól formált λ-kifejezés:

Egy λ-kifejezés jól formált, ha szintaktikusan helyes, azaz megfelel a λ-kifejezések szintaktikájának leírásában megadott szintaktikai szabályoknak.

Ha egy jól formált kifejezésben csak jól formált típusok szerelnek, ez még nem garantálja azt, hogy a kifejezés a típus szempontjából helyes. A típusok és a kifejezések jól formáltságának a megkövetelése nem elég, ennek az az oka, hogy a helyes típus leírására nem elegendőek a környezetfüggetlen szabályok. Az, hogy egy kifejezés a típus szempontjából mikor helyes, nem írható le környezetfüggetlen gramatikákkal. A típus helyességét csak további vizsgálatokkal tudjuk eldönteni, erre szolgál a típuskörnyezet és a következtetési szabályok halmaza.

Egy formális típusrendszerben *következtetéseket* adhatunk meg. Egy következtetés alakja

$\Gamma \vdash \mathcal{I}$,

ahol Γ a *statikus típuskörnyezet*, és $\mathcal{I}$ a Γ-ból adódó *állítás*.

A statikus típuskörnyezet elnevezésből a „statikus" jelzőt el is hagyhatjuk.

4.2.3. Definíció. A típuskörnyezet szintaktikája:

$$
\begin{array}{lcl}
<\textit{típuskörnyezet}> & ::= & \emptyset \\
 & | & <\textit{típuskörnyezet}>\ ,\ <\textit{változó}>:<\textit{típus}>
\end{array}
$$

A típuskörnyezetekre még a következő tulajdonságokat is megköveteljük:

- *Ha x_i egy A_i típusú változó ($1 \leq i \leq n$), akkor egy típuskörnyezet az*

  $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$

  párokból felépített lista.

- *Ha $i \neq j$, akkor $x_i \neq x_j$, azaz a típuskörnyezetben egy változó csak egyszer szerepelhet, azaz egy változónak csak egy típusa lehet.*

A Γ típuskörnyezetet arra fogjuk használni, hogy λ-kifejezések szabad változóinak típusait meghatározzuk, a szabad változók típusait majd innen fogjuk „kiolvasni".

Ha a Γ típuskörnyezet egy párt sem tartalmaz, akkor a típuskörnyezetre az $\emptyset$ jelet használjuk. A Γ-ban szereplő változók halmazát $dom(\Gamma)$-val jelöljük. A Γ-t egy olyan függvénynek tekinthetjük, amelyre ha $x_i : A_i \in \Gamma$, akkor $\Gamma(x_i) = A_i$.

4.2.4. Definíció. Jól formált típuskörnyezet:

Egy típuskörnyezet jól formált, ha szintaktikusan helyes, azaz megfelel a 4.2.3. definícióban megadott szabályoknak.

Ha a Γ típuskörnyezet jól formált, akkor ezt a

$\Gamma \vdash wf$

következtetéssel jelöljük. A $\emptyset$ üres típuskörnyezet nyilvánvalóan mindig jól formált, azaz $\emptyset \vdash wf$. Ha a Γ típuskörnyezetben az A típus jól formált, akkor ezt a tulajdonságot a

$\Gamma \vdash A$

jelöli. Számunkra most a

$\Gamma \vdash E : A$

alakú *következtetések* a fontosak. A $\Gamma \vdash E : A$ következtetés azt mondja ki, hogy figyelembe véve az E szabad változóinak a Γ típuskörnyezetben levő típusait, az E kifejezés típusa A.

Megjegyezzük, hogy egy jól formált Γ típuskörnyezet statikus, azaz ha Γ-ban egy x változóra $x : A$, akkor ez a későbbiekben nem változik meg. Még akkor sem, ha ez a típus beépül egy λ-kifejezésbe, és a típus kikerül a Γ-ból. Nem lehet a későbbiekben az $x : B$ típust megadni, ha $A \neq B$. Például a $\lambda x.\lambda x.y$ kifejezésben nem lehet a külső x típusa A, a belső x típusa pedig B, azaz ha y típusa C, akkor nem lehet a kifejezés típusa $A \to (B \to C)$.

A formális típusrendszerben egy következtetés lehet *érvényes*, vagy *érvénytelen*. Egy következtetés érvényességének bizonyítására a típusrendszer *szabályai* szolgálnak.

Egy szabály

$$\frac{\Gamma_1 \vdash \mathcal{I}_1 \quad \ldots \Gamma_n \vdash \mathcal{I}_n}{\Gamma \vdash \mathcal{I}} \quad [\textsc{szabálynév}]$$

alakú, ahol a vízszintes vonal fölött a feltétel következtetései, a vonal alatt a feltételekből származtatott következmény következtetése van. Egy szabály a feltételben szereplő következtetések érvényességet nem vizsgálja, a szabály azt mondja ki, hogy ha a feltétel mindegyik következtetése érvényes, akkor a következmény következtetése is érvényes.

A szabályoknak nevet is adhatunk, és ha szükséges, a feltételek mellett jobboldalon még a szabály alkalmazásának feltételeit is megadhatjuk.

Ha egy szabályban a feltételek halmaza üres, azaz a vonal felett nincs egy állítás sem, akkor a következtetés mindig érvényes. Az ilyen következtetéseket *axiómának* nevezzük.

Mint majd a 5.1. pontban látni fogjuk, a szabályokat a jól formált típuskörnyezet leírására is használhatjuk.

4.2.5. Példa. *(Az üres típuskörnyezet mindig jól formált)*

$$\frac{}{\emptyset \vdash \mathit{wf}} \quad [\textsc{Env}\ \emptyset]$$

A szabály azt írja le, hogy az üres típuskörnyezet minden esetben jól formált. Az állítás a 4.2.3 definíció nyilvánvaló logikai következménye, de ez a szabály a majd definiálandó következtetési rendszerekben axiómaként fog szerepelni. □

A szabályokat *típuslevezetések* készítésére is használjuk, úgy, hogy például egy *következtetési fát* vagy más néven *levezetési fát* építünk fel. A szabályokat egymáshoz kapcsoljuk, az Si szabály következményének következtetése az Sj szabály feltételének egy következtetéséhez kapcsolható, ha a két következtetés megegyezik.

4.2.6. Példa. *(Levezetési fák)*

$$\dfrac{\dfrac{\dfrac{K_{11}}{K_{12}}\;\text{[S1]}}{K_{22}}\;\text{[S2]}}{K_{32}}\;\text{[S3]} \qquad\qquad \dfrac{\dfrac{\dfrac{K_{41}}{K_{42}}\;\text{[S4]}}{K_{52}}\;\text{[S5]} \qquad \dfrac{K_{61}}{K_{62}}\;\text{[S6]}}{K_{72}}\;\text{[S7]} \quad \square$$

4.2.7. Definíció. Érvényes következtetés:

Azt mondjuk, hogy a $\Gamma \ \vdash E : A$ *következtetés érvényes, ha létezik olyan típuslevezetés, ahol a következtetés a típuslevezetéshez tartozó következtetési fa gyökérpontja.*

4.2.8. Definíció. Jól típusozott kifejezés:

Ha egy típusrendszerben a $\Gamma \ \vdash E : A$ *következtetés érvényes, akkor azt mondjuk, hogy az* E *kifejezés jól típusozott.*

A $\Gamma \ \vdash E : A$ következtetés érvényessége tehát azt jelenti, hogy az E kifejezés a típus szempontjából is helyes, és a következtetés azt is megadja, hogy az E kifejezés típusa A.

Ezek után most már megadhatjuk a *formális típusrendszer* definícióját.

4.2.9. Definíció. Formális típusrendszer:

Egy $\mathcal{T}$ *formális típusrendszer egy* $(\mathcal{S}, \mathcal{J}, \mathcal{R})$ *hármas, ahol*

- $\mathcal{S}$ *a típusrendszer szintaxisa, a típusok és a kifejezések szintaktikáját adja meg,*
- $\mathcal{J}$ *a következtetések formáit írja le,*
- $\mathcal{R}$ *a következtetések érvényességének bizonyítására szolgáló szabályok halmaza.*

4.2.10. Példa. *(Egy formális típusrendszer és a fomális matematikai rendszer)*
A $(\mathcal{S}, \mathcal{I}, \mathcal{R})$ formális típusrendszer egy olyan formális matematikai rendszer, amelyben a típusokat és kifejezéseket alkotó, azaz az $\mathcal{S}$ szintaktika leírásában szereplő szimbólumok és jelek halmaza a formális rendszer ábécéje, és a típusrendszer $\mathcal{S}$ szintaxisa a formális rendszer nyelv-

tani szabályainak halmaza. Az $\mathcal{R}$ szabályhalmaz axiómái a formális rendszer axiómái, és az $\mathcal{R}$ nem-axióma szabályai lesznek a formális rendszer szabályai.

Tehát a típusrendszer jól formált kifejezései a formális matematikai rendszer jól formált formulái, és a jól típusozott kifejezések a formális matematikai rendszer jelentéssel bíró formulái. $\square$

4.3. A típus és a biztonság

A típusok alkalmazásának alapvető célja az, hogy a program futásakor keletkező végrehajtási hibákat megelőzzük. Egy végrehajtási hiba jelentkezése esetén

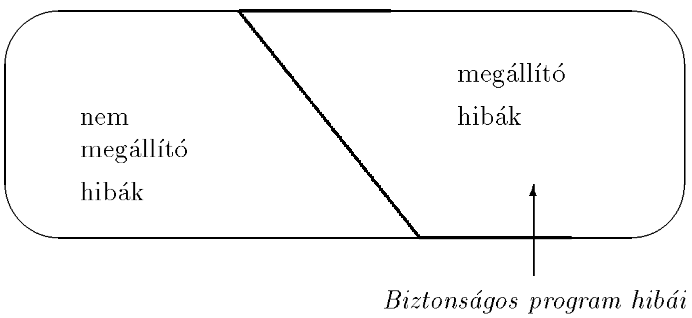
- vagy azonnal abbamarad a program végrehajtása, mert egy *kivétel* lép fel (ezeket a hibákat hívjuk *megállító hibáknak*),
- vagy a program fut tovább, és egyáltalán nem, vagy esetleg később jelentkezik valamilyen hiba (ezeket a végrehajtási hibákat hívjuk *nem megállító hibáknak*).

Megállító hiba például a nullával való osztás, vagy egy nem létező címre való hivatkozás, nem megállító hiba egy tömbelem olvasásakor az indexhatár túllépésekor a tömb előtti vagy utáni memóriaterületről történő olvasás, vagy rossz címre, például egy adatmezőbe való ugrás.

Azt mondjuk, hogy egy program *biztonságos*, ha nem okoz nem megállító hibát, azaz a program minden hiba esetén megáll. Azokat a programnyelveket, amelyekben minden program biztonságos, *biztonságos nyelveknek* nevezzük.

A típusos nyelvek úgy biztosítják a biztonságot, hogy már fordítási időben visszautasítják a nem biztonságos programokat, a nem típusos nyelveknél a biztonság elérése csak a futási időben végzett ellenőrzéssel valósítható meg.

Természetesen nem minden nyelv biztonságos, és a biztonság független attól, hogy a nyelv típusos-e. A 4.2. ábrán néhány programnyelvet osztályozunk ezen szempontok szerint.



4.1. ábra. Biztonságos program

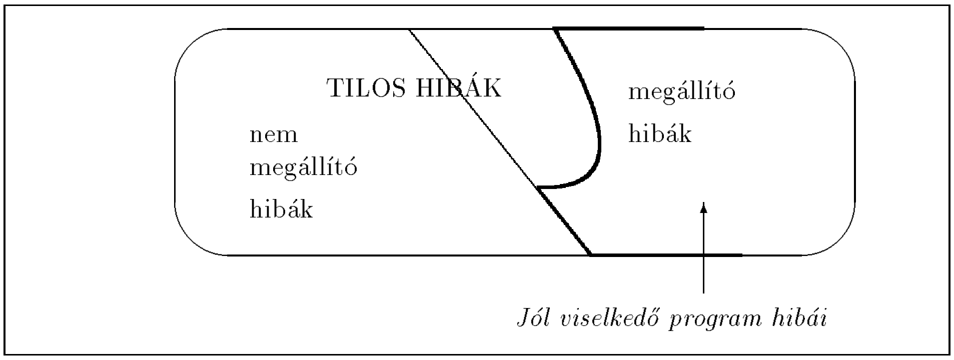
	típusos nyelv	*nem típusos nyelv*
biztonságos	ML, Haskell, Clean	LISP
nem biztonságos	C	assembly

4.2. ábra. Programnyelvek osztályozása

Egy nyelv biztonsága a programok fejlesztési és futtatási idejének arányát határozza meg. A biztonság eléréséhez szükséges vizsgáltok elhagyását legtöbbször azzal magyarázzák, hogy ezzel a hatékonyságot növelik, mivel biztonságos típusos nyelveknél a programfejlesztési idő nőhet meg, hiszen a fordítóprogram csak a biztoságos programot fogadja el, a biztonságos nem típusos nyelvek esetén pedig a run-time időben elvégzendő ellenőrzések a program futási idejét növelhetik meg jelentősen.

A biztonság ugyanakkor sok szempontból a költségek megtakarítását is jelenti, például egy biztonságos programban a hibakeresésre fordított idő minimális, vagy például lehetővé válik a hulladékgyűjtés, ami a program méretét is csökkentheti.

A programnyelvekben a nem megállító hibákat és a megállító hibák egy részét *tilos hibáknak* nevezzük. Tilos hiba például az a megállító hiba, ahol a művelet argumentuma nem megfelelő, mint például a `not 3` kifejezésben.



4.3. ábra. Jól viselkedő program

Egy programot *jól viselkedő programnak* nevezünk, ha nem okoz tilos hibát.

Nyilvánvaló, hogy minden jól viselkedő program biztonságos, de ez fordítva nem áll fenn. A típusrendszerek jól viselkedő programok előállítását tűzik ki célul, ez a biztonság vizsgálatát is magába foglalja.

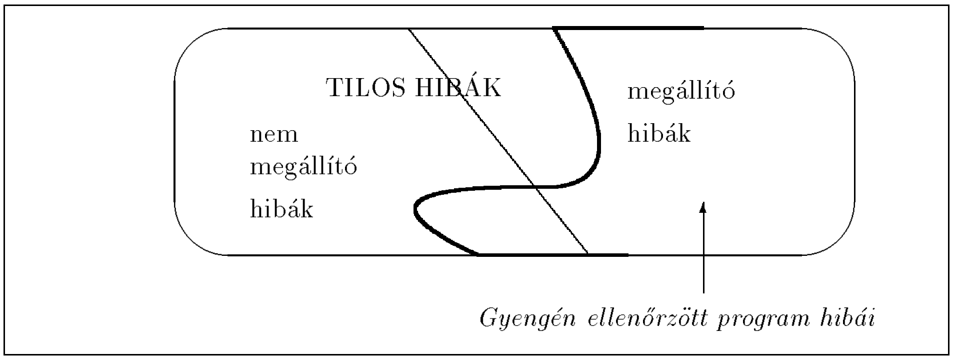
Azokat a nyelveket, amelyekben minden program jól viselkedő, *erősen ellenőrzött nyelveknek* nevezzük. Így egy erősen ellenőrzött nyelvű programban nem lehetnek nem megállító hibák, és nem lehetnek olyan megállító hibák, amelyek tilosak.

Ha egy nyelv típusos, akkor a jól viselkedés vizsgálata statikus, azaz fordítási időben végezhető. Ezt az ellenőrzést *típusellenőrzésnek*, a típusellenőrzéssel ellenőrzött programot *jól típusozott programnak* nevezzük. Ha a típusellenőrzés hibát jelez, akkor a program *rosszul típusozott program*. A rosszul típusozott program nem garantálja a jól viselkedést, futásakor tilos hibák is előfordulhatnak.

A nem típusos nyelvek is lehetnek jól viselkedőek, még akkor is, ha egyáltalán nem tartozik hozzájuk típusellenőrzés. A szükséges vizsgálatok ekkor csak futási időben végezhetők el, ezért ezek az ellenőrzések egy tilos hiba esetén, még nem megállító hiba esetén is, megállítják a program futását.

Erősen ellenőrzött, típusos nyelv az ML, és erősen ellenőrzött, de nem típusos nyelv a LISP.

Az lenne az optimális, ha a jól viselkedő programok minden megállító hibát kiszűrnének, azaz minden megállító hiba a tilos hibák közé tartozna. Sajnos ez nincs így, sőt a gyakorlatban az ellenőrzés még a nem megállító hibák egy részére sem terjed ki. Az ilyen nyelveket *gyengén ellenőrzött nyelveknek* nevezzük.



4.4. ábra. Gyengén ellenőrzött nyelv programja

Gyengén ellenőrzött nyelv például a Pascal és a C, a nem vizsgált leállító hiba a C-ben például a nagyon gyakran használt pointer-aritmetika.

4.4. Típushelyesség

Láttuk, hogy típusos rendszerekben a jól viselkedést a típusellenőrzés vizsgálta, és ha az ellenőrzés nem talált hibát, akkor a programot jól típusozott programnak neveztük. A jól típusozott program tehát nem okoz tilos hibát.

A formális típusrendszerekben is definiáltunk egy jól típusozott fogalmat, azt mondtuk, hogy egy E kifejezés jól típusozott egy Γ

típuskörnyezetben, ha létezik olyan A típus, melyre $\Gamma \vdash E : A$.

A két jól típusozott fogalom azonosságát bizonyítani kell, azaz meg kell mutatni, hogy ha egy E kifejezés jól típusozott, akkor nem okoz tilos hibát.

Vizsgáljuk R. Milnernek[3] azt a közismert állítását, hogy

(1) *egy jól típusozott program nem fut rosszul.*

Ez az állítás, pontosabban ennek az alábbiakban finomított változata a típushelyesség alapja. A típushelyesség ugyanis a következőket mondja ki. A típusrendszer helyes akkor, ha minden olyan program, amelyre a típusrendszerrel megállapítható a program jól típusozottsága, nem működik, azaz nem fut rosszul.

A jól típusozottság fogalma egyértelmű, most a „nem fut rosszul" kifejezést elemezzük.

A típusrendszer a programot fordítási időben, azaz statikusan vizsgálja. Ha egy program inputja egy természetes szám, outputja is egy természetes szám, akkor a típusrendszer megállapítja, hogy a program egy

$Num \to Num$

leképezést végez, de nem foglalkozik azzal, hogy a program például négyzetreemelést vagy éppen négyzetgyökvonást hajt végre. A típusrendszer tehát a programokat csak típusuk szerint különbözteti meg.

A típusrendszer nem a program szemantikáját vizsgálja, ezért meg kell különböztetnünk a program *run-time szemantikáját* és a program *típus szemantikáját*, és látjuk, hogy a típus szemantika nem a run-time szemantika másolata. Mivel a típustrendszer egyáltalán nem foglalkozik a run-time szemantikával, a fenti idézetet így kell módosítanunk:

(2) *egy jól típusozott program futása közben nem jelez run-time hibát.*

Run-time hibák a végrehajtó rendszer által detektált hibák, amelyek például a nem megfelelő függvény-argumentumok, vagy egy aritmetikai művelet nem megfelelő típusú operandusai esetén jelentkeznek. Nyilvánvalóan nem nevezhetjük run-time hibának azt, amikor a program run-time szemantikája nem egyezik meg a programíró szándékával.

A fenti (2) állítás csak egy irányban igaz, azaz ha egy program futása közben nem jelentkezik run-time hiba, ez még nem jelenti azt, hogy a program jól típusozott. A *Megállás Problémája* tétel szerint van olyan program, amire ez utóbbi állítás teljesül, van olyan run-time hiba jelzése nélkül futó program, aminek a jól típusozottsága nem bizonyítható.

Ezekután probáljuk megfogalmazni a típushelyességet:

$\| \; \forall P,\ ha\ \Gamma \vdash P : A,\ akkor\ P \Rightarrow E\ esetén\ \Gamma \vdash E : A,$

ahol a $P \Rightarrow E$ azt jelenti, hogy a P program futásának az eredménye E.

A (2) állítás egy kicsit mást mond, mint ez a megfogalmazás, ez a leírás azt állítja, hogy a jól típusozott programnak mindig van egy E eredménye, és az eredmény típusa azonos a program típusával. Ez a megfogalmazás erősebb, mint a (2) állítás, de célszerű a típushelyességnek ezt választani, mert ezt könnyebb bizonyítani.

A típushelyesség tehát két dologtól függ: a típusokat meghatározó típusrendszertől, és a $\Rightarrow$ levezetést meghatározó redukciós rendszertől. Tehát azt mondhatjuk, hogy egy redukciós rendszert figyelembevéve a típusrendszer helyes, ha egy program típusa megegyezik a program redukciójával kapott eredmény típusával.

A típushelyesség fenti megfogalmazása azonban még nem tökéletes. Például nyilvánvalóan

$(x : Num/y : Num) : Num,$

azaz a program jól típusozott, de mi történik, ha $y = 0$? Ez nyilván megsérti a típushelyesség állítását, hiszen a futási időben a végrehajtó rendszer run-time hibát fog jelezni.

Mi legyen a hibajelzés?

- Lehetne például egy számérték, mondjuk -1. Ez azonban nem lehet jó megoldás, hiszen ekkor a $2+(1/0)$ kifejezésnek lenne értéke, pontosan 1.

- Lehetne egy végtelen ciklusba való belépés. De ekkor hogyan különböztessük meg a run-time hibákat egymástól, és ezeket a run-time hibákat a tényleges végtelen ciklustól?

- Egy kivétel generálása tűnik a jó megoldásnak, ez egyértelműen jelezheti a hibát, és lehetőséget ad a programozónak arra, hogy a kivételhez megadja a saját kivételkezelő programját.

Ha a program redukálásának valamelyik lépésében kivételt generálhatunk, akkor ez azt jelenti, hogy a program, mint függvény, egy parciális függvény, és a „nincs értelmezve" fogalomnak itt a kivétel generálása felel meg. Például az x/y függvény egy parciális függvény, mert nincs értelmezve az $y = 0$ pontban.

Elemezve a programok végrehajtását, előre megadhatjuk a kivételeknek egy $\mathcal{X}$ halmazát, ebbe azok a kivételek tartoznak, amelyek a programfüggvények parciális tulajdonságából adódnak.

Ezzel az $\mathcal{X}$ halmazzal megadhatjuk a típushelyesség újabb megfogalmazását:

> $\forall P$, *ha* $\Gamma \vdash P : A$, *akkor*
> – $P \Rightarrow E$ esetén $\Gamma \vdash E : A$,
> – *vagy a* P *redukálása folyamán egy* $\mathcal{X}$*-beli kivétel lép fel.*

Nem foglalkoztunk még azokkal a problémákkal, hogy a P program típusának meghatározásakor végtelen ciklus léphet fel, és hogy nem feltétlenül véges a P redukálásainak száma sem.

Az első probléma megoldása egyszerű, ha a P típusa nem határozható meg véges lépésben, akkor a P nem jól típusozható. A végtelen redukciós sorozat azonban természetesnek mondható, a rekurzív függvények definiálásával, a rekurzió alkalmazásával könnyen kaphatunk olyan programokat, amelyek nem redukálhatók véges lépésben.

Ezért a típushelyesség fogalmát tovább kell finomítanunk, és ezzel megkapjuk a végső megfogalmazást.

4.4.1. Definíció. Típushelyesség

A típusrendszert típusosan helyesnek nevezzük, ha $\forall P$ *programra* $\Gamma \vdash P : A$, *és*

- *a* P *redukálásának terminálása esetén*
  - *ha* $P \Rightarrow E$, *akkor* $\Gamma \vdash E : A$,
  - *vagy a* P *redukálása folyamán egy* $\mathcal{X}$*-beli kivétel lép fel.*

Egy típusrendszer helyességét bizonyítani kell. A típusrendszer helyességének állítását z a *típushelyesség tételének* nevezzük, és ha az állítás igaz, akkor azt is mondhatjuk, hogy a típusrendszerben a típushelyesség teljesül.

Mint majd látni fogjuk, a továbbiakban vizsgálandó formális típusrendszerek mindegyike helyes típusrendszer lesz.

5. FEJEZET

Az elsőrendű típusos λ-kalkulus

Ebben a fejezetben a legegyszerűbb típusos λ-kalkulussal foglalkozunk, az *elsőrendű típusos λ-kalkulussal.* Ennek a λ-kalkulusnak szemléletükben különböző két fő tárgyalási módszere van, a módszerek elnevezése az első publikálójukra utal.

- A *Curry-típusos λ-kalkulus* 1934-ből származik [2]. A kalkulust *implicit* típusos rendszernek nevezzük, mivel nem kell a kifejezések megadásakor a típusokat megadni. Későbbi felhasználási területe a *típuslevezetés* lett. Programnyelvek implementációiban lehetővé teszi azt, hogy a forrásnyelvű programban nem kell típusokat megadni, a programnyelv fordítóprogramja határozza meg a kifejezések típusát. Az implicit típusos programnyelvre a klasszikus példa az ML.

- Church típusos λ-kalkulusról szóló cikke [1] 1940-ben jelent meg, A *Church-típusos λ-kalkulus* jellemzője az, hogy minden típusnélküli kifejezésnek egyértelmű, a kifejezés definiálásakor megadott vagy levezethető típusa van. Ezt a rendszert *explicit* típusos λ-kalkulusnak nevezzük. Az implementációban ennek a λ-kalkulusnak az alkalmazása a *típusellenőrzést* jelenti, ilyen tipikus programnyelv a Pascal vagy az Algol.

A gyakorlatban a Curry-típusos λ-kalkulust „típuslevezetéses λ-kalkulusnak" nevezik, és Church rendszere kapta „a típusos λ-kalkulus" elnevezést.

A továbbiakban mi is elsősorban a Church-típusos rendszerrel foglalkozunk, de majd tanulmányozzuk a típuslevezetés módszereit is.

5.1. Az elsőrendű Church-típusos λ-kalkulus

Az elsőrendű Church-típusos λ-kalkulust F_1 *típusrendszernek*, röviden F_1 vagy F *rendszernek* nevezzük. Ebben a rendszerben a szembetűnő változás a típusnélküli λ-kalkulushoz viszonyítva az, hogy a λ-absztrakciókban a változók típusát is jelöljük:

$\lambda x : A \,.\, E,$

azaz a λ-kifejezésekben a kötött változó típust kap, az x változó típusa ebben a λ-kifejezésben A. A programnyelvekben gyakran az E típusát is meg kell adni, az F_1 rendszerben erre nem lesz szükség.

Egy függvény típusát $A \to B$-vel jelöljük, ahol A és B típusok, $\to$ a *típuskonstruktor*. Az A a függvény argumentumának, B a függvény értékének a típusa, Ahhoz, hogy ilyen típusokat fel tudjunk építeni, szükségünk van kezdeti típusokra, azaz az *alaptípusok* halmazára. Egy ilyen alaptípushalmaz lehet például a $\{Bool,\ Nat\}$. Az alaptípusok jó megválasztása azért fontos, mert egy adott alaptípushalmazra felépített F_1 rendszerben a típusokban csak az alaptípushalmaz elemei szerepelhetnek, az alaptípusok halmazának bővítése egy másik rendszert eredményez.

5.2. Az F_1 rendszer szintaxisa

Az 1.1.1. definícióban látható, hogy a típusnélküli λ-kalkulus kifejezéseit egy egyszerű, környezetfüggetlen szintaxissal le tudtuk írni. Most megadjuk az F_1 rendszer szintaxisát.

5.2.1. Definíció. Az F_1 rendszer szintaxisa:

$$\begin{array}{lll}
<\textit{típus}> & ::= & <\textit{alaptípus}> \\
 & \mid & (<\textit{típus}> \to <\textit{típus}>) \\
<\lambda\textit{-kifejezés}> & ::= & <\textit{változó}> \\
 & \mid & (\lambda \ <\textit{változó}>:<\textit{típus}> \ . <\lambda\textit{-kifejezés}>) \\
 & \mid & (<\lambda\textit{-kifejezés}> <\lambda\textit{-kifejezés}>)
\end{array}$$

A legkülső zárójelpárt elhagyhatjuk.

Régebbi publikációkban az $A \to B$ függvénytípusra az $\mathsf{F}AB$, (AB), A^B, B^A vagy a (BA) jelölést is használták.

A fenti definícióból látható, hogy a típusnélküli λ-kalkulus leírása a típusokra vonatkozó szintaktikus szabályokkal bővült. Az F_1 rendszer szintaxisa röviden a

$$\mathbb{T} = \mathbb{U} \mid \mathbb{T} \to \mathbb{T},$$
$$\Lambda_{\mathbb{T}} = V \mid \lambda x : \mathbb{T} \,.\, \Lambda_{\mathbb{T}} \mid (\Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}})$$

alakban is írható, ahol $\mathbb{T}$ a típusokat, $\mathbb{U}$ az alaptípusokat, V a változókat, $\Lambda_{\mathbb{T}}$ a λ-kifejezéseket jelöli.

Megjegyezzük, hogy míg az E egy λ-kifejezés, az $E : A$ azt jelenti, hogy az E kifejezés típusa A, azaz az E az A típus egy eleme. Ha ezt a tartalmazást akarjuk hangsúlyozni, akkor az $E \in A$ jelölést is alkalmazhatjuk.

Az $E : A$-ban E-nek mindig nagyobb a *precedenciája*, mint az A-nak, ezt figyelembe véve $(EF) : A \equiv EF : A$, és $(\lambda x : A \,.\, E) : B \equiv \lambda x : A \,.\, E : B$.

A λ-absztrakció természetesen most is *jobbasszociatív*, az applikáció pedig *balasszociatív*.

A függvénytípus *jobbasszociatív*, ennek előnyét a következő példában mutatjuk meg.

5.2.2. Példa. *(A függvénytípus jobbasszociatív)*
A Church-számokon értelmezett $add(x, y)$ összeadás függvény a $(Nat \times Nat) \to Nat$ leképezést valósítja meg. A currying módszert alkalmazva láttuk (**??** példa), hogy

$add(x, y) = add_x(y) = (f(x))(y)$.

Az f függvény típusa $Nat \to \rho$, és az add_x függvény típusa, azaz a ρ nyilvánvalóan $Nat \to Nat$. Így az $add(x, y)$ típusa

$Nat \to (Nat \to Nat)$,
ami a jobbasszociatívitás miatt
$Nat \to Nat \to Nat$
alakban is írható. □

A típusokban és a λ-kifejezésekben a legkülső zárójelpárt és az asszociatívitások miatti redundáns zárójelpárokat elhagyhatjuk.

A 5.2.1 definíció a szintaktikus szabályokat írja le, de nem minden szintaktikusan helyes, azaz jól formált λ-kifejezés „jó" egy jól formált típus szempontjából.

5.2.3. Példa. *(A típus szempontjából hibás λ-kifejezés)*
Ha A egy alaptípus, tehát nyilvánvalóan jól formált, akkor a $\lambda x : A \,.\, xy$ kifejezés szintaktikusan helyes, de a típus szempontjából biztosan hibás, mivel az xy törzsből az következik, hogy az x-nek függvény típusúnak kell lennie, azaz az x nem lehet alaptípus. □

Tehát már az elsőrendű típusos λ-kalkulus definíciójához sem elegendő a szintaxis megadása.

5.3. Az F_1 rendszer következtetései és szabályai

Az F_1 rendszer szintaktikus szabályait már megadtuk, most adjuk meg a típusrendszer további leírását.

Az F_1 rendszerben három fajta következtetés van, az első és a második egy típuskörnyezet és egy típus jól formáltságát mondja ki, a harmadik egy jól formált kifejezés típusát adja meg.

5.3.1. Definíció. Az F_1 rendszer következtetései:

$\Gamma \vdash wf$	Γ jól formált környezet
$\Gamma \vdash A$	Γ-ban az A jól formált típus
$\Gamma \vdash E : A$	Γ-ban az E kifejezés típusa A

Az F_1 rendszerben a következtetések érvényességének bizonyítására a következő szabályokat kell használni.

5.3.2. Definíció. Az F_1 rendszer szabályai:

A környezetre vonatkozó szabályok:

$$\frac{}{\emptyset \vdash wf} \quad [\textsc{Env}\ \emptyset]$$

$$\frac{\Gamma \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash wf} \quad [\textsc{Env}\ x]$$

A típusra vonatkozó szabályok:

$$\frac{\Gamma \vdash wf \quad A \in alaptípus}{\Gamma \vdash A} \quad [\textsc{Type Const}]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B} \quad [\textsc{Type Arrow}]$$

Kifejezés típusára vonatkozó szabályok:

$$\frac{\Gamma', x : A, \Gamma'' \vdash wf}{\Gamma', x : A, \Gamma'' \vdash x : A} \quad [\textsc{Val}\ x]$$

$$\frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A \,.\, E : A \to B} \quad [\textsc{Val Fun}]$$

$$\frac{\Gamma \vdash E : A \to B \quad \Gamma \vdash F : A}{\Gamma \vdash EF : B} \quad [\textsc{Val Appl}]$$

Az F_1 rendszerben csak egy axióma van, az Env $\emptyset$.

A 5.7.3. definícióból az is látható, hogy a típusra vonatkozó szabályok (Type Const, Type Arrow) csak a szintaktikai szabályokat ismétlik meg, így, ha egy kifejezés A típusa jól formált, azaz szintaktikusan helyes, akkor a $\Gamma \vdash A$ következtetés mindig érvényes. Az ilyen következtetés érvényességének vizsgálatára tehát az F_1 rendszerben még nincs szükség. Az ilyen következtetések érvényességét majd csak a magasabb rendű típusrendszerekben kell vizsgálni. Így tehát az F_1 rendszerben az alapvető feladat a kifejezések jól típusozottságának a vizsgálata.

A Val x szabály feltételei a jól formáltságon kívül azt jelentik, hogy

$\Gamma(x) = A$, ezért a szabály egy gyengébb alakja a következő:

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$

A Val Fun szabály feltételében levő következtetés típuskörnyezetében $\Gamma, x : A$ szerepel, ami azt jelenti, hogy az x nincs benne a Γ-ban, vagyis $x \notin dom(\Gamma)$. Látható, hogy a feltétel típuskörnyezetéből az $x : A$ beépül a λ-absztrakcióba.

A Val Fun szabályt, mivel egy λ-kifejezésbe egy típust épít be, „típusbevezetésnek”, a Val Appl szabályt pedig, mivel függvénytípusból egy egyszerű típust ad eredményül, „típuseliminációnak” is nevezzük.

5.3.3. Példa. *(Az $\mathsf{id\_Nat} \equiv \lambda x : Nat \,.\, x$ kifejezés típusának meghatározása)*
Az alaptípusok halmaza legyen $K = \{Bool, Nat\}$. Ekkor

$$\dfrac{\dfrac{\dfrac{\dfrac{\emptyset \vdash wf \quad Nat \in K}{\emptyset \vdash Nat} \qquad x \notin Dom(\Gamma)}{x : Nat \vdash wf}}{x : Nat \vdash x : Nat}}{\emptyset \vdash (\lambda x : Nat \,.\, x) : Nat \to Nat}$$

□

5.3.4. Példa. *(A $\lambda x : Bool \to Nat \,.\, \lambda y : Bool \,.\, xy$ kifejezés típusának meghatározása)*
Az alaptípusok halmaza legyen $K = \{Bool, Nat\}$. Ekkor

$$\dfrac{\dfrac{\dfrac{\overline{\emptyset \vdash wf} \quad Bool \in K}{\emptyset \vdash Bool} \quad \dfrac{\overline{\emptyset \vdash wf} \quad Nat \in K}{\emptyset \vdash Nat}}{\emptyset \vdash Bool \to Nat} \qquad x \notin dom(\emptyset)}{x : Bool \to Nat \vdash wf}$$

Hasonlóan, az y-ra:

$$\frac{\dfrac{x : Bool \to Nat \ \vdash wf \qquad Bool \in K}{x : Bool \to Nat \ \vdash Bool} \qquad y \notin dom(x : Bool \to Nat)}{x : Bool \to Nat,\ y : Bool \ \vdash wf}$$

Ebből az x típusa:

$$\frac{x : Bool \to Nat,\ y : Bool \ \vdash wf}{x : Bool \to Nat,\ y : Bool \ \vdash x : Bool \to Nat}$$

Hasonlóan, az y típusa:

$$\frac{x : Bool \to Nat,\ y : Bool \ \vdash wf}{x : Bool \to Nat,\ y : Bool \ \vdash y : Bool}$$

Ezekből pedig xy-ra:

$$\frac{\begin{array}{l} x : Bool \to Nat,\ y : Bool \ \vdash \ x : Bool \to Nat \\ x : Bool \to Nat,\ y : Bool \ \vdash \ y : Bool \end{array}}{x : Bool \to Nat,\ y : Bool \ \vdash xy : Nat}$$

Most határozzuk meg az absztrakciók típusát:

$$\frac{\dfrac{x : Bool \to Nat,\ y : Bool \ \vdash xy : Nat}{x : Bool \to Nat \ \vdash (\lambda y : Bool\,.\,xy) : Bool \to Nat}}{\emptyset \ \vdash (\lambda x : Bool \to Nat\,.\,\lambda y : Bool\,.\,xy) : (Bool \to Nat) \to Bool \to Nat}$$

Mivel a

$$\emptyset \ \vdash (\lambda x : Bool \to Nat\,.\,\lambda y : Bool\,.\,xy) : (Bool \to Nat) \to Bool \to Nat$$

következtetés a levezetési fa gyökérpontja és a következtetés érvényes, a $\lambda x : Bool \to Nat\,.\,y : Bool\,.\,xy$ kifejezés jól típusozott, és a kifejezés típusa $(Bool \to Nat) \to Bool \to Nat$. □

5.3.5. Példa. *(Az Ω kifejezés típusa)*

Az F_1 rendszerben nem minden λ-kifejezésnek létezik típusa, a típusnélküli λ-kalkulusból megismert

$\Omega \equiv (\lambda x\,.\,xx)(\lambda x\,.\,xx)$

kifejezés például ilyen. Ezt a kifejezést az F_1 rendszerben felírva a következőt kapjuk:

$\Omega \equiv (\lambda x : A\,.\,xx)(\lambda x : A\,.\,xx)$

Az A típusnak függvénytípusnak kell lennie, hiszen az xx egy applikáció, és minden applikáció első tagja egy függvény, tehát A csak $B \to C$ formájú lehet, legyen tehát $A \equiv B \to C$. Azonban most az applikáció argumentuma is x, ezért az $A \equiv B$-nek is fenn kell állnia. Olyan B és C típusokat azonban nem találunk, amelyek kielégítik a $B \equiv B \to C$ egyenlőséget, azaz az Ω-hoz nem lehet típust hozzárendelni.

A típusnélküli λ-kalkulusban a $B = \ldots\ B\ \ldots$ egyenlet megoldására az Y fixpont kombinátort használtuk úgy, hogy a jobboldal B-jére egy absztrakciót alkalmaztunk, és erre a B-vel, mint argumentummal egy applikációt adtunk meg. Ez a módszer azonban itt nem alkalmazható, mivel a B-re semmilyen „típusabsztrakciót" sem tudunk alkalmazni, mivel nincs ilyen fogalmunk sem. □

5.4. Az alaptípusok halmazának bővítése

Már utaltunk arra, hogy az elsőrendű típusos λ-kalkulusokban az alaptípusok halmazának megválasztása meghatározza a definiálható típusok halmazát. A példákban eddig a $\{Bool, Nat\}$ halmazt választottuk, most ezt a halmazt bővítjük, ugyanakkor az F_1 rendszert is kiegészítjük a *Bool* és *Nat* típusra és az új alaptípusokra vonatkozó szabályokkal. Ezzel az F_1 rendszer programnyelvi kapcsolatait is megmutatjuk.

1. A *Unit* típus

Programnyelvekben gyakran van szükség annak a jelölésére, hogy egy eljárásnak nincs argumentuma, vagy egy függvénynek nincs argumentuma és értéke. Ezt most a *Unit* típussal jelöljük, a típushoz egy érték tartozik, a unit. Megjegyezzük, hogy ezt a típust gyakran *Void*-nak vagy *Null*-nak is nevezik.

A *Unit* típusra nem adunk meg semmilyen műveletet, a típus szabályai csak azt állítják, hogy a *Unit* típus egy lehetséges típus, azaz jól formált, és azt, hogy a *Unit* típushoz csak egy érték, a unit tartozik.

$$\frac{\Gamma \vdash wf}{\Gamma \vdash Unit} \quad [\textsc{Type Unit}]$$

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{unit} : Unit} \quad [\textsc{Val Unit}]$$

2. A *Bool* típus

Az **??** pontban definiáltuk a true és false λ-kifejezéseket, ez a két érték alkotja a *Bool* típust. Ugyanitt láttuk az if λ-absztrakciót is, erre az utasításra most egy külön szabályt is meg tudunk adni.

$$\frac{\Gamma \vdash wf}{\Gamma \vdash Bool} \quad [\textsc{Type Bool}]$$

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{true} : Bool} \quad [\textsc{Val True}]$$

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{false} : Bool} \quad [\textsc{Val False}]$$

$$\frac{\Gamma \vdash E : Bool \quad \Gamma \vdash F : A \quad \Gamma \vdash G : A}{\Gamma \vdash (\mathsf{if}\ E\ F\ G) : A} \quad [\textsc{Fun If}]$$

3. A *Nat* típus

A számkonstansokat a **??** pontban a c_0 konstans és a succ függvény megadásával definiáltuk. Az így meghatározott értékek alkotják a *Nat* típust. A számjegyrendszerekre megadtuk még a pred és a zero függvényt is.

A *Nat* típusra a következő szabályokat adhatjuk meg:

$$\frac{\Gamma \vdash wf}{\Gamma \vdash Nat} \quad [\textsc{Type Nat}]$$

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{c}_0 : Nat} \quad [\textsc{Val } c_0]$$

$$\frac{\Gamma \vdash E : Nat}{\Gamma \vdash \mathsf{succ}\ E : Nat} \quad [\textsc{Val Succ}]$$

$$\frac{\Gamma \vdash E : Nat}{\Gamma \vdash \mathsf{pred}\ E : Nat} \quad [\textsc{Fun Pred}]$$

$$\frac{\Gamma \vdash E : Nat}{\Gamma \vdash \mathsf{zero}\ E : Bool} \quad [\textsc{Fun Zero}]$$

4. A *Pair* típus

Adjuk meg a rendezett párra vonatkozó szabályokat. A rendezett párt adó pair függvényt és a párokra alkalmazható first és second függvényeket a ?? pontban definiáltuk. Ha a rendezett pár első eleme A_1, a második eleme A_2 típusú, akkor a párok lehetséges értékhalmazát, azaz a párok típusát jelöljük $Pair_{A_1 \times A_2}$-vel. Megjegyezzük, hogy ezt a típust gyakran *szorzat típusnak* is nevezik, és $A_1 \otimes A_2$-vel is jelölik.

A szabályok a következők:

$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash Pair_{A_1 \times A_2}} \quad [\textsc{Type Pair}_{A_1 \times A_2}]$$

$$\frac{\Gamma \vdash E_1 : A_1 \quad \Gamma \vdash E_2 : A_2}{\Gamma \vdash \mathsf{pair}\ E_1\ E_2 : Pair_{A_1 \times A_2}} \quad [\textsc{Val Pair}]$$

$$\frac{\Gamma \vdash E : Pair_{A_1 \times A_2}}{\Gamma \vdash \mathsf{first}\ E : A_1} \quad [\textsc{Fun First}]$$

$$\frac{\Gamma \vdash E : Pair_{A_1 \times A_2}}{\Gamma \vdash \mathsf{second}\ E : A_2} \quad [\textsc{Fun Second}]$$

4. A *Union* típus

A $Pair_{A_1 \times A_2}$ típus elemei olyan párok, amelyeknek első komponensei A_1, a második komponensei A_2 típusúak, azaz azt is mondhatjuk, hogy a pár típusa A_1 *és* A_2. Ha egy olyan típust definiálunk, amelynek elemei A_1 *vagy* A_2 típusúak, akkor a *Union* típust kapjuk meg, és a típust $Union_{A_1+A_2}$-vel jelöljük. Megjegyezzük, hogy ezt a típust gyakran *diszjunkt összeg típusnak* is nevezik, és $A_1 \oplus A_2$-vel is jelölik.

A *Union* típus az imperatív programnyelvekből jól ismert *case* utasítás használatára ad lehetőséget, ahol a típus egy elemének az aktuális típusa határozza meg, hogy a *case* utasítás melyik ágát kell végrehajtani.

A $Union_{A_1+A_2}$ típus konstruktora az $\mathsf{inLeft}_{\mathsf{A_2}}$ és az $\mathsf{inRight}_{\mathsf{A_1}}$ művelet. Azt, hogy egy $Union_{A_1+A_2}$ típusú kifejezés aktuális típusa A_1 vagy A_2, az isLeft és az $\mathsf{isRight}$ *Bool* értékű függvények határozzák meg, ha a típus A_1, akkor az isLeft függvény *true* értéket, a $\mathsf{isRight}$ függvény *false* értéket ad, az A_2 típusra az értékek rendre *false* és *true*. A típustól függő elágazást a case művelet valósítja meg.

A szabályok a következők:

$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash Union_{A_1+A_2}} \quad [\textsc{Type Union}_{A_1+A_2}]$$

$$\frac{\Gamma \vdash E : A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash \mathsf{inLeft}_{A_2}\ E : Union_{A_1+A_2}} \quad [\textsc{Val InLeft}]$$

$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash E : A_2}{\Gamma \vdash \mathsf{inRight}_{A_1}\ E : Union_{A_1+A_2}} \quad [\textsc{Val InRight}]$$

$$\frac{\Gamma \vdash E : Union_{A_1+A_2}}{\Gamma \vdash \mathsf{isLeft}\ E : Bool} \quad [\textsc{Fun IsLeft}]$$

$$\frac{\Gamma \vdash E : Union_{A_1+A_2}}{\Gamma \vdash \mathsf{isRight}\ E : Bool} \quad [\textsc{Fun IsRight}]$$

$$\frac{\Gamma \vdash E : Union_{A_1+A_2} \quad \Gamma \vdash F_1 : B \quad \Gamma \vdash F_2 : B}{\Gamma \vdash (\mathsf{case}\ E\ \mathsf{of}\ ((\mathsf{isLeft}\ E)\ \mathsf{then}\ F_1)\ \mathsf{or}\ ((\mathsf{isRight}\ E)\ \mathsf{then}\ F_2)) : B}$$

[Fun Case]

A *Union* típussal a *Bool* típus is kifejezhető. Legyen

$Bool \equiv Union_{Unit+Unit}$.

Mivel a *Unit* típusnak csak egy eleme van, a *Union* típuskonstruktoraival csak két elem adható meg, ezeket nevezzük a *true* és *false* értékeknek.

$\mathsf{true} \equiv \mathsf{inLeft}_{Unit}\ \mathsf{unit}$,
$\mathsf{false} \equiv \mathsf{inRight}_{Unit}\ \mathsf{unit}$.

Ekkor

$\mathsf{case\ true\ of}\ ((\mathsf{isLeft\ true})\ \mathsf{then}\ F_1)\ \mathsf{or}\ ((\mathsf{isRight\ true})\ \mathsf{then}\ F_2)$

az F_1, és hasonlóan, a false az F_2 kifejezést adja, azaz a case egy *if the else* műveletnek felel meg.

5. A *Ref* típus

A típusrendszer lehetőséget ad arra is, hogy az imperatív nyelvekből jól ismert változó tartalmú memóriacellákat is leírjuk. Ha A egy típus, akkor jelöljük *Ref* A-val az A típusú értékeket tartalmazó cellákat, azaz a *Ref* A típushalmaz elemei legyenek ezek a memóriacellák. Egy új cellát az alloc művelettel lehet allokálni, a dealloc művelettel megszüntetni, és egy cella értékét az értértékadást jelző := művelettel lehet megváltoztatni. Mivel az értékadás egy *mellékhatást* eredményez, az értékadás típusa legyen *Unit*.

$$\frac{\Gamma \vdash A}{\Gamma \vdash Ref\ A} \quad [\text{Type Ref}]$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{alloc}\ M : Ref\ A} \quad [\text{Val Alloc}]$$

$$\frac{\Gamma \vdash M : \mathit{Ref}\ A}{\Gamma \vdash \mathsf{dealloc}\ M : A} \quad [\textsc{Val Dealloc}]$$

$$\frac{\Gamma \vdash M : \mathit{Ref}\ A \quad \Gamma \vdash N : A}{\Gamma \vdash M := N : \mathit{Unit}} \quad [\textsc{Fun Assign}]$$

6. A *List* típus

A **??** pontban vizsgáltuk a láncolt lista adatszerkezeteket, és megadtuk a nil és cons konstruktorok, a head, tail és empty függvények λ-kifejezéseit.

Az A típusú elemeket tartalmazó lista típusát List_A-val jelöljük, erre a típusra a következő szabályokat adhatjuk meg:

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathit{List}_A} \quad [\textsc{Type List}_A]$$

$$\frac{\Gamma \vdash \mathit{wf}}{\Gamma \vdash \mathsf{nil} : \mathit{List}_A} \quad [\textsc{Val Nil}]$$

$$\frac{\Gamma \vdash E : A \quad \Gamma \vdash F : \mathit{List}_A}{\Gamma \vdash \mathsf{cons}\ E\ F : \mathit{List}_A} \quad [\textsc{Val Cons}]$$

$$\frac{\Gamma \vdash E : \mathit{List}_A}{\Gamma \vdash \mathsf{head}\ E : A} \quad [\textsc{Fun Head}]$$

$$\frac{\Gamma \vdash E : \mathit{List}_A}{\Gamma \vdash \mathsf{tail}\ E : \mathit{List}_A} \quad [\textsc{Fun Tail}]$$

$$\frac{\Gamma \vdash E : \mathit{List}_A}{\Gamma \vdash \mathsf{empty}\ E : \mathit{Bool}} \quad [\textsc{Fun Empty}]$$

Látható, hogy a nil konstanshoz is hozzá kell rendelni a List_A típust, azért, hogy a lista utolsó, a nil-t tartalmazó eleme is jól típusozott legyen.

8. A *Record* típus

A *Record* típus a *Pair* rendezett pár típus olyan általánosításának tekinthető, ahol a párt n-esre bővítjük, és az n-es minden eleméhez egy-egy címkét rendelünk.

Egy rekord mezőkből áll, a mezőknek neveik vannak, minden mező egy megadott típusú adatot tartalmaz.

Egy n-elemű rekord mezőnevei, azaz a mezők címkéi legyenek l_i-k ($1 \leq i \leq n$), az l_i mező adatának a típusa pedig legyen A_i. Ekkor a rekord típusa

$Record\{l_1 : A_1, \dots, l_n : A_n\}$,

vagy röviden

$\{\!| \; l_1 : A_1, \dots, l_n : A_n \; |\!\}$.

Ha az $E : \{\!| \; l_1 : A_1, \dots, l_n : A_n |\!\}$ rekord mezőit az $F_1 : A_1, \dots, F_n : A_n$ kifejezésekkel fel akarjuk tölteni, akkor ezt a $\textsf{record}\ l_1 = F_1 \ \dots \ l_n = F_n$ *konstruktorral* tehetjük meg.

Egy rekordhoz definiálhatunk $\textsf{selector}\ l_i$ ($1 \leq i \leq n$) függvényeket is, amelyek a rekord l_i nevű mezőjének adatát határozzák meg:

$\textsf{selector}\ l_i\ (\textsf{record}\ l_1 = F_1 \ \dots \ l_n = F_n) \ \rightarrow \ F_i$.

A $\textsf{selector}\ l_i\ E$ applikációt gyakran $E\,.\,l_i$-vel, vagy inkább az $E\char`^ l_i$ jellel is jelölik.

A *Record* típusra vonatkozó szabályok a következők:

$$\frac{\Gamma \vdash A_1 \ \dots \ \Gamma \vdash A_n}{\Gamma \vdash \{\!| \; l_1 : A_1, \dots, l_n : A_n \; |\!\}} \quad [\textsc{Type Record}]$$

$$\frac{\Gamma \vdash F_1 : A_1 \ \dots \ \Gamma \vdash F_n : A_n}{\Gamma \vdash \textsf{record}\ l_1 = F_1 \ \dots \ l_n = F_n : \{\!| \; l_1 : A_1, \dots, l_n : A_n \; |\!\}} \quad [\textsc{Val Record}]$$

$$\frac{\Gamma \vdash \textsf{record}\ l_1 = F_1 \ \dots \ l_n = F_n : \{\!| \; l_1 : A_1, \dots, l_n : A_n \; |\!\}}{\Gamma \vdash \textsf{selector}\ l_i\ (\textsf{record}\ l_1 = F_1 \ \dots \ l_n = F_n) : A_i} \quad [\textsc{Fun Selector}]$$

9. A *Variant* típus

Míg a *Record* típus a *Pair*, a *Variant* típus a két komponensű *Union* típus általánosítása. A *Union* típust több komponensre bővítjük, és minden komponenshez egy címkét rendelünk.

Egy n komponensű *Variant* típus komponenseinek címkéi legyenek l_i-k ($1 \leq i \leq n$), az l_i komponens típusa legyen A_i. Ekkor a *Variant* típus így írható le:

$Variant\{l_1 : A_1, \ldots, l_n : A_n\}$.

A típus konstruktorát nevezzük $\mathsf{inVariant}$-nak, és az aktuális típus vizsgálatát végezze egy $\mathsf{isVariant}$ függvény. Az erre a típusra alkalmazható case többirányú elágazást hajt végre.

A *Variant* típusra vonatkozó szabályok a következők:

$$\frac{\Gamma \vdash A_1 \ \ldots \ \Gamma \vdash A_n}{\Gamma \vdash Variant\{l_1 : A_1, \ldots, l_n : A_n\}} \quad [\textsc{Type Variant}_{A_1+A_2}]$$

$$\frac{\Gamma \vdash A_1 \ \ldots \ \Gamma \vdash A_n \quad \Gamma \vdash E : A_i}{\Gamma \vdash \mathsf{inVariant}_{Variant\{l_1:A_1,\ldots,l_n:A_n\}}\ l_i = E : Variant\{l_1 : A_1, \ldots, l_n : A_n\}}$$

[Val InVariant]

$$\frac{\Gamma \vdash E : Variant\{l_1 : A_1, \ldots, l_n : A_n\}}{\Gamma \vdash \mathsf{isVariant}\ l_i\ E : Bool} \quad [\textsc{Fun IsVariant}]$$

$$\frac{\Gamma \vdash E : Variant\{l_1 : A_1, \ldots, l_n : A_n\} \quad \Gamma \vdash F_1 : B \ \ldots \ \Gamma \vdash F_n : B}{\Gamma \vdash (\mathsf{case}\ E\ \mathsf{of}((\mathsf{isVariant}\ l_1\ E)\mathsf{then}\ F_1)\mathsf{or}\ldots\mathsf{or}((\mathsf{isVariant}\ l_n\ E)\mathsf{then}\ F_n)) : B}$$

[Fun Case]

5.5. Az alaptípusok definiálása

A 5.2.1. definícióban láttuk, hogy az F_1 rendszer típusai az *alaptípusok* halmazának elemeiből épülnek fel, az alaptípusok halmaza egy adott F_1 rendszerben nem változtatható meg.

Most egy olyan módszert mutatunk be, amellyel a predefinit alaptípusok explicit felsorolását elhagyhatjuk, és az alaptípusokat az *induktív típusdefiníciónak* nevezett művelettel definiálhatjuk, a típus elemeit pedig a definícióban levő konstruktorokkal állíthatjuk elő. Az ezzel a definícióval megadott alaptípusú F_1 rendszert F_{1ind} rendszernek nevezzük, ahol az *ind* az induktív szóra utal.

Az induktív típusdefiníciót az indtype kulcsszó vezeti be, utána található a definiálandó új típus neve, majd a with kulcsszó után a *konstruktor* : *típus* párok felsorolása.

5.5.1. Definíció. Induktív típusdefiníció:

Az F_{1ind} rendszer szintaktikája csak abban különbözik a az F_1 rendszer 5.2.1. definícióban megadott szintaktikájától, hogy a típus$_k$ alaptípus megadását most az indtype *művelettel végezzük.*

$$\begin{array}{llll} \textsf{indtype} < \textit{típus}_k > \textsf{with} & E_1 & : & T_{11} \to T_{12} \to \ldots \to T_{1m_1} \to < \textit{típus}_k > \\ & E_2 & : & T_{21} \to T_{22} \to \ldots \to T_{2m_2} \to < \textit{típus}_k > \\ & \ldots & & \\ & E_n & : & T_{n1} \to T_{n2} \to \ldots \to T_{nm_n} \to < \textit{típus}_k > \end{array}$$

ahol

- *E_i $(1 \leq i \leq n)$ a definiált típushoz tartozó konstruktor, speciálisan lehet az új típus értékhalmazának egy eleme, azaz a típus konstansa is. A kifejezések leírását a konstruktorokkal is bőviteni kell, azaz*

  $$< \lambda\textit{-kifejezés} > \; = \; \ldots \mid E_1 \mid E_2 \mid \ldots \mid E_n,$$

- *T_{ij} $(1 \leq i \leq n, 1 \leq j \leq m_i)$ az éppen ezzel a típusdefinícióval megadott $<$ típus$_k$ $>$ típusból és már korábban megadott, ismert típusokból készített jól-formált típus.*

A definícióból látható, hogy ha E_i-t applikáljuk például m_i darab, rendre $T_{i1}, T_{i2}, \ldots, T_{im_i}$ típusú argumentumra, akkor egy, az indtype utáni $<$ *típus*$_k$ $>$ típusú eredményt kapunk.

Látható az is, hogy ez a definíció konstansos λ-kifejezéseket ad, ahol a konstruktorok a λ-kalkulus δ-függvényeinek felelnek meg, a speciális konstans típusértékek pedig a λ-kalkulus konstansai lesznek.

5.5.2. Példa. *(A konstruktor nélküli típus)*
Az induktív típusdefiníció lehetőséget ad olyan típus definiálására is, amelynek egyáltalán nincs konstruktora, ami azt jelenti, hogy nincs egyetlen olyan λ-kifejezés sem, amelynek ez lenne a típusa. Nevezzük ezt a típust *Absurd*-nak, így a típus definíciója

$$\textsf{indtype}\ Absurd\ \textsf{with}\ \emptyset$$ $\square$.

5.5.3. Példa. *(A Unit típus)*
A *Unit* típus értékhalmaza egy elemet tartalmaz, ez a típus a következőképpen adható meg:

$$\textsf{indtype}\ Unit\ \textsf{with}\ \textsf{unit} : Unit$$

Nyilvánvaló, hogy a típus **unit** konstruktora konstans. $\square$

5.5.4. Példa. *(A Bool típus)*
A logikai, azaz a *Bool* típus megadása induktív típusdefinícióval a következő:

$$\textsf{indtype}\ Bool\ \textsf{with}\ \begin{array}{ll} \textsf{true} : & Bool \\ \textsf{false} : & Bool \end{array}$$

Látható, hogy a *Bool* típus mindkét konstruktora konstans. $\square$

5.5.5. Példa. *(A Nat típus induktív típusdefinícióval)*
A természetes számok típushalmaza, azaz a *Nat* típus a következő:

$$\textsf{indtype}\ Nat\ \textsf{with}\ \begin{array}{ll} 0 : & Nat \\ \textsf{succ} : & Nat \to Nat \end{array}$$ $\square$

Most megmutatjuk, hogy a fenti példában definiált *Nat* típus értékhalmaza hogyan határozható meg. Az értékhalmaz megadására egy speciális *primitív rekurzív függvényt* használunk:

5.5.6. Definíció. Az iter függvény:

$$\begin{array}{llll} \textsf{iter} & 0 & x\ f & \twoheadrightarrow\ x \\ \textsf{iter} & (\textsf{succ}\ y) & x\ f & \twoheadrightarrow\ f\ (\textsf{iter}\ y\ x\ f) \end{array}$$

A $\twoheadrightarrow$ jel a redukciót jelzi. Az iter függvény szemléletesen azt jelenti, hogy az f-t az x-re pontosan az első argumentumban megadott darabszámszor alkalmazzuk.

Az iter függvény az első argumentumban megadott tetszőleges számkonstansra biztosan termináló redukció-sorozatot ad, az iter függvény bevezetése a λ-kifejezések erős normalizálását nem változtatja meg.

Az n természetes szám a következő kifejezéssel adható meg:

$\mathsf{iter}\ n\ 0\ \mathsf{succ}$,

és ebből a Nat típusú tetszőleges n számot előállító függvény:

$\lambda x : Nat.\ \mathsf{iter}\ x\ 0\ \mathsf{succ}$.

5.5.7. Példa. *(Az iter függvény felhasználása)*

Megjegyezzük, hogy az iter függvényt más célra is fel lehet használni, például az $E_{2x+1}\ n = 2n + 1$, $E_{x+y}\ a\ b = a + b$ és az $E_{x*y}\ a\ b = a * b$ műveletek a következő kifejezésekkel adhatók meg:

$$\begin{aligned}
E_{2x+1} &\equiv \lambda x : Nat.\ \mathsf{iter}\ x\ (\mathsf{succ}\ 0)\ (\lambda z : Nat.\ \mathsf{succ}\ (\mathsf{succ}\ z)),\\
E_{x+y} &\equiv \lambda x : Nat.\ \lambda y : Nat.\ \mathsf{iter}\ x\ y\ \mathsf{succ},\\
E_{x*y} &\equiv \lambda x : Nat.\ \lambda y : Nat.\ \mathsf{iter}\ x\ 0\ (\lambda z : Nat.\ E_{z+y}\ z\ y).
\end{aligned}$$

Mindhárom kifejezés típusa $Nat \to Nat \to Nat$. □

A fenti iter függvény első argumentuma és a függvény értéke csak Nat típusú lehet. Látható, hogy az iter függvény a Nat típus elemein struktúrális indukciót hajt végre, és az indukció eredménye Nat típusú.

Általánosítsuk az iter függvényt úgy, hogy a struktúrális indukciót egy tetszőleges A típus elemeire hajtsa végre, és eredményül egy B típusú értéket adjon. Ezt a függvényt jelöljük iter_A_B-vel. Ezzel a jelöléssel az előbbi iter függvény alakja most iter_Nat_Nat lesz.

5.5.8. Definíció. Az iter_A_B függvény:

Ha az A típust megadó induktív típusdefinícióban n konstruktor van, és az i-edik $(1 \leq i \leq n)$ konstruktor

$E_i : T_{i1} \rightarrow T_{i2} \rightarrow \ldots \rightarrow T_{im_i} \rightarrow A$

alakú, akkor

$\mathsf{iter\_A\_B}\ (E_i\ G_1 G_2 \ldots G_{m_i})\ F_1 F_2 \ldots F_n \quad \rightharpoonup \quad F_i\ G'_1 G'_2 \ldots G'_{m_i},$

ahol

- *F_i egy $T'_{i1} \rightarrow T'_{i2} \rightarrow \ldots \rightarrow T'_{im_i} \rightarrow B$ típusú függvény,*
- *G'_j azt jelöli, hogy G_j-ben minden A típusú t részkifejezést ($\mathsf{iter\_A\_B}\ t\ F_1 F_2 \ldots F_n$)-nel helyettesítünk,*
- *és T'_{ik} jelöli azt, hogy T_{ik}-ban az A minden előfordulását B-re helyettesítjük.*

Ha az E_i konstruktornak nincs paramétere, akkor a fentiek szerint

$\mathsf{iter\_A\_B}\ E_i\ F_1\ F_2\ \ldots\ F_n \quad \rightharpoonup \quad F_i,$

és ha az E_i konstruktor egyik paraméterének sincs A típusú részkifejezése, akkor

$\mathsf{iter\_A\_B}\ (E_i\ a_1 a_2 \ldots\ a_{m_i})\ F_1 F_2 \ldots F_n \quad \rightharpoonup \quad F_i\ a_1 a_2 \ldots a_{m_i}.$

5.5.9. Példa. *(Az* iter_Bool_Bool *függvény)*
A fentiek szerint, ha a *Bool* leírásában **true** az első és **false** a második konstruktor, akkor

$\mathsf{iter\_Bool\_Bool\ true}\ F_1 F_2 \ \rightharpoonup\ F_1$
$\mathsf{iter\_Bool\_Bool\ false}\ F_1 F_2 \ \rightharpoonup\ F_2$ □

5.5.10. Példa. *(Az iter_Bool_Bool alkalmazása)*
Az and, or és not függvények leírására az iter_Bool_Bool függvényt

használhatjuk:

$\mathsf{and} \equiv \lambda x : Bool.\, \lambda y : Bool.\, \mathsf{iter\_Bool\_Bool}\ xy\ \mathsf{false},$
$\mathsf{or} \equiv \lambda x : Bool.\, \lambda y : Bool.\, \mathsf{iter\_Bool\_Bool}\ x\ \mathsf{true}\ y,$
$\mathsf{not} \equiv \lambda x : Bool.\, \mathsf{iter\_Bool\_Bool}\ x\ \mathsf{false\ true}.$

Például,

$\mathsf{not\ true} \twoheadrightarrow_\beta \mathsf{iter\_Bool\_Bool\ true\ false\ true} \twoheadrightarrow \mathsf{false}$ □

5.5.11. Példa. *(Az iter_Bool_Nat alkalmazása)*
Az *if x : Bool then y : Nat else z : Nat* szerkezet leírására az iter_Bool_Nat függvényt használhatjuk:

$\mathsf{if} \equiv \lambda x : Bool.\, \lambda y : Nat.\, \lambda z : Nat.\, \mathsf{iter\_Bool\_Nat}\ xyz$

Például, ha y és z 2 és 3, azaz Nat típusú, akkor

$\mathsf{if\ false}\ 2\ 3 \twoheadrightarrow_\beta \mathsf{iter\_Bool\_Nat\ false}\ 2\ 3 \twoheadrightarrow 3,$

a $Bool$ definíciójában a false a második konstruktor, ezért a iter_Bool_Nat-nak a $Bool$ típusú argumentum utáni második argumentumát kapjuk eredményül. □

5.5.12. Példa. *(A $Pair_{A_1 \times A_2}$ típus)*
Nat típusú számokból képezzük a $Pair_{Nat\times Nat}$ típust. A típus leírása induktív típusdefinícióval a következő:

$\mathsf{indtype}\ Pair_{Nat\times Nat}\ \mathsf{with\ pair} : Nat \to Nat \to Pair_{Nat\times Nat}$

Egy rendezett pár első és második elemét az $\mathsf{iter\_Pair_{Nat\times Nat}\_Nat}$ függvénnyel adhatjuk meg:

$\mathsf{first} \equiv \lambda x : Pair_{Nat\times Nat}\,.\, \mathsf{iter\_Pair_{Nat\times Nat}\_Nat}\ x\ (\lambda f : Nat.\, \lambda s : Nat.\, f)$
$\mathsf{second} \equiv \lambda x : Pair_{Nat\times Nat}\,.\, \mathsf{iter\_Pair_{Nat\times Nat}\_Nat}\ x\ (\lambda f : Nat.\, \lambda s : Nat.\, s)$

Mivel a $Pair_{Nat\times Nat}$ típusnak egy konstruktora van, a pár első vagy második elemét az $\mathsf{iter\_Pair_{Nat\times Nat}\_Nat}$ függvény második argumentumában levő kifejezés választja ki. Míg a pair konstruktor típusa $Nat \to Nat \to Pair_{Nat \times Nat}$, ennek a kifejezésnek a típusa $Nat \to Nat \to Nat$.

Például a pair uv pár első eleme:

first (pair uv) $\equiv$

$(\lambda x : Pair_{Nat\times Nat} .\, \mathsf{iter\_Pair_{Nat\times Nat}\_Nat}\ x\ (\lambda f : Nat .\, \lambda s : Nat .\, f))$ (pair uv) $\rhd_\beta$

$\mathsf{iter\_Pair_{Nat\times Nat}\_Nat}$ (pair uv) $(\lambda f : Nat .\, \lambda s : Nat .\, f)$ $\rhd$

$(\lambda f : Nat .\, \lambda s : Nat .\, f)\ uv$ $\rhd_\beta$

$(\lambda s : Nat .\, u)\ v$ $\rhd_\beta$ u □

5.5.13. Példa. *(A láncolt lista adatszerkezet List típusa)*

A Nat típusú számokat tartalmazó láncolt lista megadása a következő:

indtype $List$ with nil $: List$

cons $: Nat \to List \to List$

A lista fejelemét egy iter_List_Nat függvénnyel határozzuk meg, mivel a fejelem típusa Nat. A problémát azonban az okozza, hogy így egy Nat típusú értéket kell adnunk akkor is, ha a lista üres. Ezért válasszunk ki egy számot, ez a szám legyen az üres lista „jele", azaz a hibajelzés, és ez a szám legyen a head második paramétere. Így

head $\equiv \lambda x : List .\, \lambda d : Nat .\,$ iter_List_Nat $xd(\lambda c : Nat .\, \lambda r : List .\, c)$

Határozzuk meg a $[1 :: 2 :: 3 :: 4] \equiv$ cons 1(cons 2(cons 3(cons 4 nil))) lista fejelemét. A hibajelzés kódja legyen 999.

head (cons 1(cons 2(cons 3(cons 4 nil)))) $999 \equiv$

$\lambda x : List .\, \lambda d : Nat .\,$ iter_List_Nat xd $(\lambda c : Nat .\, \lambda r : List .\, c)$

(cons 1(cons 2(cons 3(cons 4 nil)))) 999 $\rhd_\beta^+$

iter_List_Nat (cons 1(cons 2(cons 3(cons 4 nil)))) 999 $(\lambda c : Nat .\, \lambda r : List .\, c)$ $\rhd$

$(\lambda c : Nat .\, \lambda r : List .\, c)\ 1$ $\underline{\text{(cons 2 (cons 3(cons 4 nil)))}}$

Mivel az aláhúzott kifejezés *List* típusú, az iter_List_Nat redukciójának végrehajtásához ezt a kifejezést tovább kell alakítani:

$\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))\ \twoheadrightarrow$

$\mathsf{iter\_List\_Nat}\ (\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil})))\ 999\ (\lambda c : Nat\,.\,\lambda r : List\,.\,c)\ \twoheadrightarrow$

$(\lambda c : Nat\,.\,\lambda r : List\,.\,c)\ 2\ \underline{(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))}$

Az aláhúzott kifejezés átalakítását az olvasóra bízzuk. Látható, hogy ezek az átalakítások az eredményt nem befolyásolják, hiszen a $\lambda c : Nat\,.\,\lambda r : List\,.\,c$ függvény eredményül az első argumentumot adja, és így eredményül valóban a lista $1 : Nat$ elemét kapjuk meg.

Egy lista maradékrészének megadása kissé bonyolult. A maradék részt úgy a legegyszerűbb felépíteni, hogy a lista építését jobboldalról, a lista utolsó elemétől kezdjük, és mire a lista első eleméhez érünk, az ekkor alkalmazott cons konstruktor második argumentuma éppen a meghatározandó maradék rész. Azonban ehhez nem elég csak a maradék részeket, azaz a cons konstruktor második argumentumait nyilvántartani, hiszen egy cons konstruktor második argumentumát az előző lépésben létrehozott teljes lista alkotja. Erre a problémára a szokásos megoldás az, hogy párokat hozunk létre, egy pár első eleme a lista maradék részét, a második eleme pedig a teljes listát tartalmazza. Így a lista felépítése után az eredményül kapott páros első eleme éppen a lista maradék részét tartalmazza (5.1. ábra).

Először adjuk meg a listákat tartalmazó párok típusát:

$\mathsf{indtype}\ Pair_{List\times List}\ \mathsf{with}\ \mathsf{pair} : List \to List \to Pair_{List\times List}$

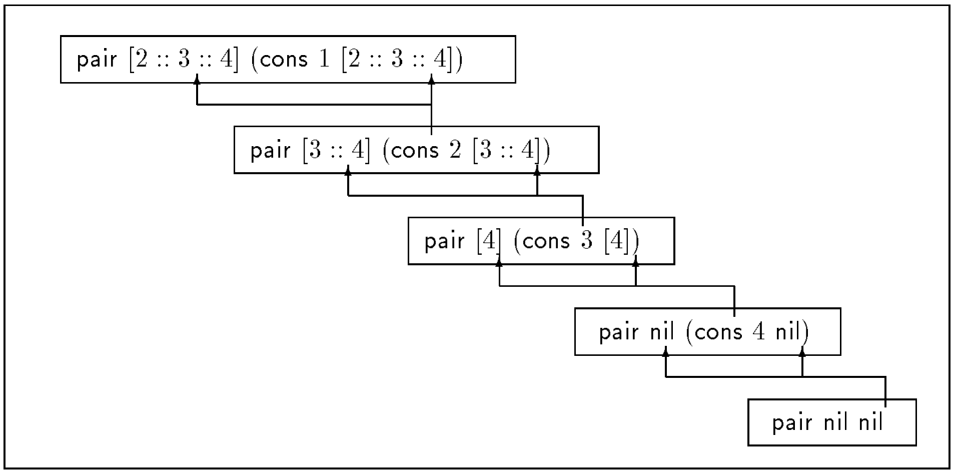
Ekkor a maradék rész a következő.

$$\begin{aligned}\mathsf{tail} \equiv \lambda l : List\,.\,&\mathsf{first}(\mathsf{iter\_List\_Pair}_{List\times List}\ l(\mathsf{pair\ nil\ nil})\\ &(\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))\end{aligned}$$

Határozzuk meg az üres lista maradék részét.

$\mathsf{tail\ nil} \equiv$

$(\lambda l : List\,.\,\mathsf{first}(\mathsf{iter\_List\_Pair}_{List\times List}\ l(\mathsf{pair\ nil\ nil})$



5.1. ábra. Az $[1 :: 2 :: 3 :: 4]$ lista

$$(\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))))\mathsf{nil}\ \twoheadrightarrow_\beta$$

$\mathsf{first}(\mathsf{iter\_List\_Pair}_{List\times List}\ \mathsf{nil}(\mathsf{pair\ nil\ nil})$
$\qquad(\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r))\ \twoheadrightarrow$
$\mathsf{first}(\mathsf{pair\ nil\ nil})\ \twoheadrightarrow_\beta^+$
nil

Határozzuk meg az $[1 :: 2 :: 3 :: 4] \equiv \mathsf{cons}\ 1(\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil})))$ lista maradék részét.

$\mathsf{tail}(\mathsf{cons}\ 1(\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil})))) \equiv$
$(\lambda l : List\,.\,\mathsf{first}(\mathsf{iter\_List\_Pair}_{List\times List}\ l(\mathsf{pair\ nil\ nil})$
$\qquad(\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))))$
$\qquad(\mathsf{cons}\ 1(\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))))\ \twoheadrightarrow_\beta$
$\mathsf{first}(\mathsf{iter\_List\_Pair}_{List\times List}\ (\mathsf{cons}\ 1(\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))))(\mathsf{pair\ nil\ nil})$
$\qquad(\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r))))\ \twoheadrightarrow$
$\mathsf{first}(\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r))$

$$1\ \underline{(\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))}$$

A fejelem meghatározásánál látottakhoz hasonlóan, mivel az az aláhúzott kifejezés $List$ típusú, az $\mathsf{iter\_List\_Pair}_{List\times List}$ redukciójának végrehajtásához ezt a kifejezést tovább kell alakítani:

$$\begin{array}{l}(\mathsf{iter\_List\_Pair}_{List\times List}\ (\mathsf{cons}\ 2(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil})))(\mathsf{pair\ nil\ nil})\\ \qquad (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))) \twoheadrightarrow\\ (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))\\ \qquad 2\ \underline{(\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))}\end{array}$$

A redukciót tovább folytatva

$$\begin{array}{l}(\mathsf{iter\_List\_Pair}_{List\times List}\ (\mathsf{cons}\ 3(\mathsf{cons}\ 4\ \mathsf{nil}))(\mathsf{pair\ nil\ nil})\\ \qquad (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))) \twoheadrightarrow\\ (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))\\ \qquad 3\ \underline{(\mathsf{cons}\ 4\ \mathsf{nil})}\end{array}$$

majd

$$\begin{array}{l}(\mathsf{iter\_List\_Pair}_{List\times List}\ (\mathsf{cons}\ 4\ \mathsf{nil})(\mathsf{pair\ nil\ nil})\\ \qquad (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))) \twoheadrightarrow\\ (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))\\ \qquad 4\ \underline{\mathsf{nil}}\end{array}$$

és végül

$$\begin{array}{l}(\mathsf{iter\_List\_Pair}_{List\times List}\ (\mathsf{nil})(\mathsf{pair\ nil\ nil})\\ \qquad (\lambda c : Nat\,.\,\lambda r : Pair_{List\times List}\,.\,\mathsf{pair}(\mathsf{second}\ r)(\mathsf{cons}\ c(\mathsf{second}\ r)))) \twoheadrightarrow\\ \mathsf{pair\ nil\ nil}\end{array}$$

azaz a 5.1. ábra utolsó sorában szereplő kifejezést kaptuk meg. Ha ezt a $\underline{\mathsf{nil}}$ kifejezés helyére írjuk és végrehajtjuk a kifejezésben megadott β-redukciókat, akkor éppen az ábra utolsó előtti sorának kifejezését kapjuk. Végrehajtva a $\mathsf{iter\_List\_Pair}_{List\times List}$ redukció minden lépését, látható, hogy a redukció éppen az ábra első sorának kifejezését határozza meg, és ennek a párosnak az első tagja a lista keresett tail része. □

5.6. Az F_1 rendszer operációs szemantikája

Ahhoz, hogy megadhassuk az F_1 rendszer konverziós szabályait, először azt kell megadnunk, hogy hogyan határozhatjuk meg a típusos λ-kifejezések szabad változóit, majd a helyettesítés műveletét kell definiálnunk.

5.6.1. Definíció. A szabad változó:

1.Az x változó szabad az x kifejezésben,

2.az x szabad az EF-ben, ha x szabad E-ben vagy szabad az F-ben,

3.az x szabad $\lambda y : A.E$-ben, ha $x \neq y$ és az x szabad E-ben.

Ha most is $FV(E)$ jelöli az E kifejezés szabad változóinak halmazát, akkor az **??**. definíció alapján

- $FV(x) = \{x\}$,
- $FV(EF) = FV(E) \cup FV(F)$,
- $FV(\lambda x : A.E) = FV(E) \setminus \{x\}$.

Látható, hogy a szabad változók halmazának képzése megegyezik a típusnélküli kifejezéseknél megadott **??** definíció módszerrel. A kötött változók, a zárt kifejezések, egy kifejezés lezárása is megegyezik a típusnélküli λ-kalkulusra megadott azonos nevű fogalmakkal.

Az $E[x := F]$ λ-kifejezéssel most is azt jelöljük, hogy az E-ben a szabad x változókat F-fel helyettesítjük. A *helyettesítést* az **??** Definícióhoz hasonlóan, most is az E szerkezete szerinti indukcióval adjuk meg.

5.6.2. Definíció. Helyettesítés:

$$1. x[y := G] \equiv \begin{cases} G, \textit{ha } x \equiv y, \\ x, \textit{ egyébként}, \end{cases}$$

$$2. (EF)[y := G] \equiv (E[y := G])(F[y := G]),$$

$$3. (\lambda x : A.E)[y := G] \equiv \begin{cases} \lambda x : A.E, & \textit{ha } x \equiv y, \\ \lambda x : A.E[y := G], & \textit{ha } x \not\equiv y \textit{ és } x \notin FV(G), \\ \lambda x : A.E, & \textit{egyébként}. \end{cases}$$

Ha a helyettesítést tetszőleges y-ra és G-re hajtjuk végre, könnyen típushibát kaphatunk. Egy helyettesítés a típus szempontjából csak akkor lesz helyes, ha a helyettesítendő változó típusa megegyezik a behelyettesített kifejezés típusával. Ezt mondja ki a következő lemma:

5.6.3. Lemma. (Helyettesítési lemma)

$$\frac{\Gamma, x : A \vdash E : B \quad \Gamma \vdash F : A}{\Gamma \vdash E[x := F] : B}$$

A lemma azt is megadja, hogy az E kifejezés típusa a helyettesítéssel nem változik meg.

Ismerve egy típusos λ-kifejezés szabad változóit és a helyettesítés szabályait, megadhatjuk az α-konverzió definícióját is:

5.6.4. Definíció. Az α-konverzió:

Ha az E-ben nincs y szabad változó, akkor $\lambda x : A.E \leftrightarrow_\alpha \lambda y : A.E[x := y]$
.

Most adjuk meg a típusos λ-kifejezésekre vonatkozó β-redukciót:

5.6.5. Definíció. A β-redukció:

Ha az $E[x := F]$-ben az F szabad változói nem válnak az E kötött változóivá, akkor

$(\lambda x : A.E)F \rightarrow_\beta E[x := F]$.

Talán meglepő, hogy a β-redukció végrehajtásakor nem vizsgáljuk az F kifejezés típusát, a redukciót bármilyen típusú F kifejezéssel végrehajthatjuk, azaz nem követeljük meg a redukálandó kifejezés jól típusozottságát.

Ha az F kifejezés típusa a fenti β-redukcióban nem A, akkor típushibát kapunk. A típushelyesség tétele majd éppen azt mondja majd ki, hogy a jól típusozott kifejezések nem okozhatnak hibát.

A redukciós sorozat és a normál forma fogalma a típusnélküli azonos nevű fogalmakkal megegyező módon definiálhatók.

A típusnélküli λ-kalkulusban láttuk, hogy egy λ-kifejezésnek lehet, hogy nincs is normál formája. Most ezt a kérdést vizsgáljuk meg a típusos λ-kalkulusban.

Az E kifejezés struktúrájára alkalmazott indukcióval bizonyítható a következő két állítás:

5.6.6. Lemma. (Típuskörnyezet kiterjesztése)

Ha $\Gamma \vdash E : A$ és $\Gamma \subseteq \Delta$, akkor $\Delta \vdash E : A$.

5.6.7. Lemma. (Típuskörnyezet szűkítése)

Ha $\Gamma \vdash E : A$ és $\Delta \equiv \{x : B \mid x : B \in \Gamma \text{ és } x \in FV(E)\}$, akkor $\Delta \vdash E : A$.

Az első állítás azt mondja ki, hogy a típuskörnyezet bővítése a kifejezések típusát nem változtatja meg, míg a második lemma szerint egy kifejezés típusa nem változik, ha a típuskörnyezetből elhagyjuk a kötött változóira vonatkozó típusállításokat.

E két lemma felhasználásával bizonyítható a következő tétel, ami azt mondja ki, hogy a β-redukciók egy jól típusozott kifejezés típusát nem változtatják meg.

5.6.8. Tétel. (A tárgyredukció tétele)

Ha $\Gamma \vdash E : A$ és $E \twoheadrightarrow^* F$, akkor $\Gamma \vdash F : A$.

A tétel elnevezése onnan származik, hogy egy $E : A$ típusállításban az E-t a típusállítás tárgyának nevezzük.

A tétel csak a jól típusozott E kifejezésekre igaz, azaz a $\Gamma \vdash E : A$ következtetésnek igaznak kell lennie, de ha E nem típusozható, akkor ettől az F kifejezésnek még lehet típusa. A tételben tehát a $\twoheadrightarrow^*$

iránya is lényeges, azaz a tétel csak β-redukciókra vonatkozik, az állítás β-absztrakciókra már nem teljesül.

5.6.9. Példa. *(A tárgyredukció tétele és a β-absztrakció)*

Nyilvánvaló, hogy $\mathsf{I} \leftarrow_\beta \mathsf{KI}\Omega$. A jobboldalon álló kifejezés azonban típushibás, hiszen Ω-hoz semmilyen típus sem rendelhető, mint ezt a **??** példában láttuk. □

Ha egy kifejezésnek létezik típusa, akkor ez a típus egyértelmű.

5.6.10. Tétel. (Típusok egyértelműsége)

Ha $\Gamma \vdash E : A$ és $\Gamma \vdash E : B$, akkor $A \equiv B$.

Az F_1 rendszernek van egy olyan tulajdonsága, ami a típusnélküli rendszereknek nincs meg, és amelyik kulcsszerepet játszik az alkalmazásokban.

5.6.11. Tétel. (Az erős normalizálás tétele)

Az F_1 típusrendszerben az $E : A$ jól típusozott kifejezésnek nincs végtelen β-redukciós sorozata.

Ebből pedig azonnal következik a következő állítás:

5.6.12. Tétel. (A normalizálás tétele)

Az F_1 rendszerben minden jól típusozott λ-kifejezésnek van normál formája.

A típusos λ-kalkulusban is érvényes a Church-Rosser tétellel megfogalmazott rombusz-tulajdonság.

5.6.13. Tétel. (Az I. Church-Rosser tétel)

Ha az $E : A$ jól típusozott kifejezésre $E : A \twoheadrightarrow^* E_1 : A$ és $E : A \twoheadrightarrow^* E_2 : A$, akkor létezik egy olyan $F : A$ kifejezés, melyre $E_1 : A \twoheadrightarrow^* F : A$ és $E_2 : A \twoheadrightarrow^* F : A$.

A normalizálás tétele és az I. Chuch-Rosser tétel garantálja, hogy minden jól típusozott λ-kifejezésnek véges lépésben meghatározható a normál formája, és ez a normál forma egyértelmű. A tétel azt is kimondja, hogy a normál formára hozás lépéseiben a típus nem változik meg.

A tárgyredukció tételének, az I. Chuch-Rosser tételnek és a típusok egyértelműsége tételének a következménye a következő állítás, ami az F_1 típusrendszer egyik sajátos tétele.

5.6.14. Tétel. (Típusok egyértelműsége és a β-konverzió)

Ha $\Gamma \vdash E : A$, $E \twoheadrightarrow^* F$ és $\Gamma \vdash F : B$, akkor $A \equiv B$.

A tétel tehát azt mondja ki, hogy ha $E \twoheadrightarrow^* F$ esetén mind az E, mind az F jól típusozott, akkor a két kifejezés típusa megegyezik.

5.7. Az altípus

A modern programnyelvek egyik általános tulajdonsága az, hogy a típusrendszer altípusok kezelését is lehetővé teszi. Mint korábban láttuk, a típus egy értékhalmaz, ezért az A típus egy eleme a típus értékhalmazát tartalmazó bővebb halmaznak is eleme. Ez a bővebb halmaz is meghatároz egy típust, és ez a halmaz-tartalmazás vezet el az *altípus* fogalmához: az A típus a bővebb típus értékhalmaz átal meghatározott B típus altípusa, ezt az $A \leq B$-vel jelöljük.

Ha az F_1 rendszert kiegészítjük az altípus fogalmával, akkor ezt a kiegészített rendszert $F^1_{\leq}$ (ejtsd: ef-egy-szub) *rendszernek* nevezzük.

Az $F^1_{\leq}$ rendszer az F_1 rendszer bővítése, ezért csak az új szintaktikai elemet, az új következtetést és az új szabályokat adjuk meg.

5.7.1. Definíció. Az $F^1_{\leq}$ rendszer szintaxisa:

Az F_1 rendszer szintaxisa a következővel bővül:

$$< \textit{típus} > \;=\; \textit{Top}$$

Az $F^1_{\leq}$ rendszerben tehát egy új alaptípus, a *Top* típus jelenik meg.

5.7.2. Definíció. $F^1_{\leq}$ rendszer következtetései:

Az F_1 rendszer következtetései a következővel bővülnek:

$\Gamma \;\vdash\; A \leq B$ az A típus a B típus altípusa

A következtetések érvényességére alkalmazható szabályok a következők:

5.7.3. Definíció. Az $F^1_{\leq}$ rendszer szabályai:

Az F_1 rendszer szabályai a következőkkel bővülnek:

A típusra vonatkozó új szabályok:

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathit{Top}} \quad [\textsc{Type Top}]$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \leq \mathit{Top}} \quad [\textsc{Sub Top}]$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \leq A} \quad [\textsc{Sub Refl}]$$

$$\frac{\Gamma \vdash A \leq B \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \quad [\textsc{Sub Trans}]$$

$$\frac{\Gamma \vdash A' \leq A \quad \Gamma \vdash B \leq B'}{\Gamma \vdash A \to B \leq A' \to B'} \quad [\textsc{Sub Arrow}]$$

Kifejezés típusára vonatkozó új szabály:

$$\frac{\Gamma \vdash x : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash x : B} \quad [\textsc{Val Subsumption}]$$

Az $F^1_{\leq}$ rendszerben tehát a *Top* típus a „legbővebb" típus, azaz minden jól típusozott kifejezés típusa a *Top*. Látható, hogy az altípus reláció reflexív és tranzitív, és a *Top* típus a maximális elem.

A Val Subsumption, azaz a kiterjesztés szabálya azt mondja ki, hogy ha egy kifejezés típusa A, és B az A-nál "bővebb" típus, azaz $A \leq B$, akkor a kifejezés típusának a B-t is megadhatjuk. Ez azt jelenti, hogy az altípus reláció a halmazokra megadható tartalmazás relációnak felel meg.

A kiterjesztés szabálya a *biztonságos helyettesítés elvét* is leírja, ez az elv azt mondja ki, hogy $A \leq B$ esetén a B típus minden esetben biztonságosan helyettesíthető A-val, vagyis az A biztonságosan alkalmazható a B típus helyett.

Két függvénytípus közötti altípusra vonatkozó Sub Arrow szabály a következőképpen magyarázható meg: Ha az E függvény típusa $A \to B$, akkor a függvény argumentumának típusa A, és argumentum lehet nyilván minden olyan kifejezés is, amelyiknek a típusa az A egy A' altípusa. Ugyanakkor, mivel a függvény értékének típusa B, a függvény értékének típusa egyben egy, a B-t tartalmazó bővebb B' típus is. Így az E függvény elfogadja az A' típusú argumentumot, és ad egy B' típusú eredményt, azaz típusa $A' \to B'$. Ez a tulajdonság azt jelenti, hogy minden $A \to B$ típus az $A' \to B'$ típus altípusa is, azaz $A \to B \leq A' \to B'$. Látható, hogy az altípus kontravariáns (ellenkező irányú) a függvények argumentumára, és kovariáns (azonos irányú) a függvények értékére.

Az altípus reláció megadható struktúrált típusokra is, a kiterjesztés Val Subsumption szabályának ezekre a típusokra is teljesülnie kell.

A *Pair* típusra az altípus az elemenkénti altípus figyelembevételével definiálható:

$$\frac{\Gamma \vdash A_1 \leq B_1 \quad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash \mathit{Pair}_{A_1 \times A_2} \leq \mathit{Pair}_{B_1 \times B_2}} \quad [\textsc{Sub Pair}]$$

A *Record* típusra az altípus reláció az elemek darabszámának és az azonos címkéjű elemek típusának figyelembevételével határozható meg:

$$\frac{\Gamma \vdash A_1 \leq B_1 \ \ldots \ \Gamma \vdash A_m \leq B_m \quad \Gamma \vdash A_{m+1} \ \ldots \ \Gamma \vdash A_n \quad (m \leq n)}{\{\!|\ l_1 : A_1, \ldots, l_n : A_n |\!\} \leq \{\!|\ l_1 : B_1, \ldots, l_m : B_m |\!\}}$$

[Sub Record]

A *Record* típusokra tehát a nagyobb elemszámú rekord lesz az altípusa a kisebb elemszámú rekordnak, aminek az az alapja, hogy a nagyobb elemszámú rekordban több "megkötés" van. Az azonos címkéjű elemekre temészetesen az $A_i \leq B_i$ $(1 \leq i \leq m)$ elemenkénti altípus relációnak teljesülnie kell.

5.8. Az F_1 rendszerrel definiálható függvények

Az elsőrendű típusos λ-kalkulusban a rekurzív függvényekhez nem rendelhető típus, így az F_1 rendszer kifejező ereje a típusnélküli λ-kalkulusénál lényegesen kisebb, hiszen ott a parciális függvények is leírhatók voltak. (lásd **??** pont).

Az F_1 rendszerrel a *kiterjesztett polinomok* írhatók le, ezek osztálya lényegesen kisebb, mint a rekurzív függvények osztálya.

Legyenek a *Nat* típus elemei a Church-számok, ebben a számjegyrendszerben az n természetes számot reprezentálja a következő típusos λ-kifejezés:

$$\mathsf{c_n} \equiv \lambda f : \alpha \to \alpha \,.\, \lambda x : \alpha \,.\, f^n(x) \quad (n = 0, 1, 2, \dots).$$

Vizsgáljuk az $f : \mathbb{N}^n \to \mathbb{N}$ függvények definiálhatóságát. Az f függvény típusának rövid leírására vezessük be a következő jelölést:

$$Nat^k \equiv Nat \to Nat \to \cdots \to Nat,$$

ahol a jobboldalon pontosan k darab *Nat* szerepel.

5.8.1. Definíció. λ-definiálható függvény:

Azt mondjuk, hogy az $f : \mathbb{N}^n \to \mathbb{N}$ függvény λ-definiálható, ha ezt a függvényt az

$$F : Nat^n \to Nat$$

λ-kifejezés reprezentálja, azaz minden $m_1, m_2, \dots m_n$ -re

$$F \ \mathsf{c}_{\mathsf{m}_1} \ \mathsf{c}_{\mathsf{m}_2} \ \dots \ \mathsf{c}_{\mathsf{m}_n} \ = \mathsf{c}_{f(m_1, m_2, \dots, m_n)}.$$

5.8.2. Példa. *(A konstans függvény)*

Az F_1 rendszerben a $K_m(x) = m$ konstans függvény λ-definiálható:

$$K_m \equiv \lambda x : Nat \,.\, \mathsf{c}_m$$ □

5.8.3. Példa. *(Az előjelfüggvény)*

Legyen a $sg(x)$ előjelfüggvény a következő:

$$sg(0) = 0,$$
$$sg(x) = 1 \quad (x \neq 0),$$

Ekkor a $sg(x)$ függvény alakja az F_1 rendszerben:

$\mathsf{sg} \equiv \lambda x : Nat\,.\,x(\mathsf{true}\,\mathsf{c}_1)\;\mathsf{c}_0 \equiv \lambda x : Nat\,.\,x(\underline{(\lambda x : Nat\,.\,\lambda y : Nat\,.\,x)\;\mathsf{c}_1})\;\mathsf{c}_0 \twoheadrightarrow$
$\lambda x : Nat\,.\,x(\lambda y : Nat\,.\,\mathsf{c}_1)\;\mathsf{c}_0.$

Például, $\mathsf{sg}\;\mathsf{c}_0 = \mathsf{c}_0$,

$\mathsf{sg}\;\mathsf{c}_0 \equiv \underline{(\lambda x : Nat\,.\,x(\lambda y : Nat\,.\,\mathsf{c}_1)\;\mathsf{c}_0)\;\mathsf{c}_0} \twoheadrightarrow$
$\mathsf{c}_0\;(\lambda y : Nat\,.\,\mathsf{c}_1)\;\mathsf{c}_0 \equiv$
$\underline{(\lambda f x\,.\,x)(\lambda y : Nat\,.\,c_1)}\;\mathsf{c}_0 \twoheadrightarrow$
$(\lambda x\,.\,x)\;\mathsf{c}_0 \rightarrow \mathsf{c}_0,$

és $\mathsf{sg}\;\mathsf{c}_1 = \mathsf{c}_1$,

$\mathsf{sg}\;\mathsf{c}_0 \equiv \underline{(\lambda x : Nat\,.\,x(\lambda y : Nat\,.\,\mathsf{c}_1)\;\mathsf{c}_0)\;\mathsf{c}_1} \twoheadrightarrow$
$\mathsf{c}_1\;(\lambda y : Nat\,.\,\mathsf{c}_1)\;\mathsf{c}_0 \equiv$
$\underline{(\lambda f : Nat \rightarrow Nat\,.\,\lambda x : Nat\,.\,fx)(\lambda y : Nat\,.\,\mathsf{c}_1)}\;\mathsf{c}_0 \twoheadrightarrow$
$\underline{(\lambda x : Nat\,.\,(\lambda y : Nat\,.\,\mathsf{c}_1)\;x)\;\mathsf{c}_0} \twoheadrightarrow$
$\underline{(\lambda y : Nat\,.\,c_1)\;\mathsf{c}_0} \rightarrow \mathsf{c}_1$ □

5.8.4. Példa. *(Az U_i^n szelektor függvény)*
Az F_1 rendszerben az $U_i^n(x_1, x_2, \ldots, x_n) = x_i$ szelektor függvény λ-definiálható:

$\mathsf{U}_\mathsf{i}^\mathsf{n}(x_1, x_2, \ldots, x_n) \equiv \lambda x_1 : Nat\,.\,\lambda x_2 : Nat\,.\,\ldots\,.\,\lambda x_n : Nat\,.\,x_i$ □.

5.8.5. Példa. *(Az összeadás és a szorzás művelet)*
Az F_1 rendszerben az összeadás és a szorzás művelete is megadható:

$\mathsf{add} \equiv \lambda x : Nat\,.\,\lambda y : Nat\,.\,\lambda p : Nat\,.\,\lambda q : Nat\,.\,xp(ypq),$
$\mathsf{mul} \equiv \lambda x : Nat\,.\,\lambda y : Nat\,.\,\lambda p : Nat\,.\,x(yp)$ □

5.8.6. Definíció. A kiterjesztett polinomok:

Kiterjesztett polinomok a következő függvények:

1.a $K_m(x) = m \quad m = 1, 2, \ldots$ *konstans függvények,*
2.az $sg(x)$ *előjelfüggvény*
3.az $U_i^n(x_1, x_2, \ldots, x_n) = x_i \quad (1 \leq i \leq n)$ *szelektor függvény.*

Kiterjesztett polinomokból további új kiterjesztett polinomokat az összeadás és a szozás műveletével konstruálhatunk.

4.Az összeadás:
legyen $f : \mathbb{N}^i \to \mathbb{N}$ *és* $g : \mathbb{N}^j \to \mathbb{N}$ *két kiterjesztett polinom,*
ekkor az $f + g$ *kiterjesztett polinom legyen*
$(f + g)(x_1, \ldots, x_i, y_1, \ldots, y_j) = f(x_1, \ldots, x_i) + g(y_1, \ldots, y_j),$
5.A szorzás:
legyen $f : \mathbb{N}^i \to \mathbb{N}$ *és* $g : \mathbb{N}^j \to \mathbb{N}$ *két kiterjesztett polinom,*
ekkor az $f \cdot g$ *kiterjesztett polinom legyen*
$(f \cdot g)(x_1, \ldots, x_i, y_1, \ldots, y_j) = f(x_1, \ldots, x_i) \cdot g(y_1, \ldots, y_j),$

5.8.7. Tétel. Schwichtenberg tétel [1976.]

Az F_1 *rendszerben definiálható függvények osztálya pontosan azonos a kiterjesztett polinomok osztályával.*

Az F_1 rendszer által leírható függvények halmaza kétféleképpen növelhető:

- a *Nat* típusnak nem a Church-számokat feleltetjük meg,
- az F_1 rendszerbe bevezetjük a *primitív rekurziót* és az *if-then-else* műveletet.

A $\mathsf{c_n}$ Church-szám

$\mathsf{c_n} \equiv \lambda f : \alpha \to \alpha \,.\, \lambda x : \alpha \,.\, f^n(x) \quad (n = 0, 1, 2, \ldots),$

a szám típusa

$(\alpha \to \alpha) \to \alpha \to \alpha,$

nem ennyire speciális típusú számjegyekkel bővebb λ-definiálható függvényhalmazt kaphatunk.

A másik módszer az, hogy a típusok alaphalmazának a *Nat* és *Bool* típust választjuk, és az F_1 rendszert a *primitív rekurzió* és az *if-then-else* művelettel bővítjük. Ezt a bővített rendszert *Gödel T rendszerének* nevezzük, és λT-vel jelöljük.

A λT rendszerben *Nat* és *Bool* típusok, mint alaptípusok szerepelnek, és ezekre a típusokra alakalmazhatók az F_1 rendszer szabályai.

A rendszer konstansai a true és false konstansok, az if konstans, azaz δ-függvény, ezekre érvényesek a **??** pontban megadott szabályok:

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{true} : Bool} \quad [\textsc{Val True}]$$

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{false} : Bool} \quad [\textsc{Val False}]$$

$$\frac{\Gamma \vdash E : Bool \quad \Gamma \vdash F : A \quad \Gamma \vdash G : A}{\Gamma \vdash (\mathsf{if}\ E\ F\ G) : A} \quad [\textsc{Fun If}]$$

A rendszernek két *Nat* típusú konstansa van, a c_0 nulla konstans, és a succ δ-függvény, ezekre is érvényesek a **??** pontban leírt szabályok:

$$\frac{\Gamma \vdash wf}{\Gamma \vdash \mathsf{c}_0 : Nat} \quad [\textsc{Val}\ c_0]$$

$$\frac{\Gamma \vdash E : Nat}{\Gamma \vdash \mathsf{succ}\ E : Nat} \quad [\textsc{Val Succ}]$$

A rendszerben van még egy δ-függvény, a recur konstans, ami a **??**. Definícióban szereplő, primitív rekurzióval meghatározott f függvény háromváltozós alakja:

$$\begin{aligned} f(0, x_2, x_3) &= g(x_2, x_3), \\ f(succ(x_1), x_2, x_3) &= h(f(x_1, x_2, x_3), x_1, x_2, x_3), \end{aligned}$$

a λT rendszerben speciálisan

$g(x_2, x_3) = U_1^2(x_2, x_3) = x_2,$

és h az x_3 kétváltozós függvény, azaz speciálisan

$h = x_3(u, v) = x_3(f(x_1, x_2, x_3), U_1^3(x_1, x_2, x_3)) = x_3(f(x_1, x_2, x_3), x_1),$

tehát

$$\begin{aligned}\mathsf{recur}(0, x_2, x_3) &= x_2,\\ \mathsf{recur}(\mathsf{succ}(x_1), x_2, x_3) &= x_3(\mathsf{recur}(x_1, x_2, x_3), x_1).\end{aligned}$$

A recur függvényre vonatkozó típusszabály a következő:

$$\frac{\Gamma \vdash E : Nat \quad \Gamma \vdash F : A \quad \Gamma \vdash G : A \to Nat \to A}{\Gamma \vdash (\mathsf{recur}\ E\ F\ G) : A} \quad [\text{FUN RECUR}]$$

Most adjuk meg a λT rendszer operációs szemantikáját, a redukciókat $\to_T$-vel jelöljük. A λT rendszerben a következő redukciók alkalmazhatók:

- β-redukció,
- az if-re vonatkozó redukció:
  $\mathsf{if\ true}\ E\ F \to_T E$
  $\mathsf{if\ false}\ E\ F \to_T F$
- a recur-ra vonatkozó redukció:
  $\mathsf{recur}\ \mathsf{c}_0\ x\ f \to_T x$
  $\mathsf{recur}\ (\mathsf{succ}\ y)\ x\ f \to_T f\ (\mathsf{recur}\ y\ x\ f)\ y$

A λT rendszerre is igaz az *erős normalizálás tétele*, és érvényes az *I. Church-Rosser tétel*, azaz a bevezetett két függvény az F_1 rendszer alapvető tulajdonságait nem változtatja meg.

Az if bevezetése után a and, or és not logikai függvények is könnyen definiálhatók:

$$\begin{aligned}\mathsf{and}\ E\ F &\equiv \mathsf{if}\ E\ F\ \mathsf{false}\\ \mathsf{or}\ E\ F &\equiv \mathsf{if}\ E\ \mathsf{true}\ F\\ \mathsf{not}\ E &\equiv \mathsf{if}\ E\ \mathsf{false\ true}\end{aligned}$$

Így például

$$\begin{array}{lll}\mathsf{not\ true} & \equiv & \mathsf{if\ true\ false\ true}\ \to_T\ \mathsf{false}\\ \mathsf{and\ false}\ F & \equiv & \mathsf{if\ false}\ F\ \mathsf{false}\ \to_T\ \mathsf{false}\\ \mathsf{or\ true}\ F & \equiv & \mathsf{if\ true\ true}\ F\ \to_T\ \mathsf{true}\end{array}$$

Ugyanakkor

$$\begin{array}{ll}\mathsf{and}\ E\ \mathsf{false} & \not\to_T\ \mathsf{false}\\ \mathsf{or}\ E\ \mathsf{true} & \not\to_T\ \mathsf{true}\end{array}$$

mivel

$$\begin{array}{lll}\mathsf{and}\ E\ \mathsf{false} & \equiv & \mathsf{if}\ E\ \mathsf{false\ false}\\ \mathsf{or}\ E\ \mathsf{true} & \equiv & \mathsf{if}\ E\ \mathsf{true\ true}\end{array}$$

és ezek nem redukálhatók tovább, mivel normál formában vannak.

A recur primitív rekurzió egy egyszerűbb alakja az *iteráció*, amit iter-rel jelölünk, és amelynek a redukciós szabályai a következők:

$$\begin{array}{lll}\mathsf{iter}\ \mathsf{c}_0\ c\ f & \to_T & c\\ \mathsf{iter}\ (\mathsf{succ}\ n)\ c\ f & \to_T & f\ (\mathsf{iter}(n\ cf))\end{array}$$

Az iter csak abban különbözik a recur függvénytől. hogy a harmadik argumentuma egy egyváltozós függvény. Ez az iter függvény megegyezik a 5.5. pontban is szereplő *iter* függvénnyel.

Látható, hogy az recur függvény egy magasabbrendű függvény, és ennek a magasabb rendű függvénynek a definiálása primitív rekurzióval történt. Lényegében ez a magasabb rendű recur függvény növeli meg a λT rendszer kifejező erejét.

Bizonyítható, hogy a *Bool* típusú konstansok és az if függvény nem növeli a λT rendszerben leírható függvények halmazát, így ezek csupán arra szolgálnak, hogy a λT rendszer szintaktikája változatosabb legyen.

5.8.8. Tétel. A λT rendszerben definiálható függvények [1]

[1]Legyen $\mathcal{N} =< \mathbb{N}, +, \cdot, \prime, \infty, => $ az aritmetika standard modellje, ahol $+, \cdot$ az operátorok, $0, 1$ a kitüntetett elemek, és $=$ egy reláció az $\mathbb{N}$ felett. $Th(\mathcal{N})$ olyan

A λT rendszerben definiálható függvények osztálya megegyezik a Peano Aritmetikában bizonyíthatóan teljes függvények osztályával.

A λT rendszerben definiálható függvények osztálya még nem egyezik meg a rekurzív függvények osztályával, de például egy nem primitív rekurzív függvény, az *Ackermann függvény* már leírható. Az *Ackermann függvény* definíciója a következő:

$$\begin{array}{lcl} A(0,n) & = & n+1 \\ A(m+1,0) & = & A(m,1) \\ A(m+1,n+1) & = & A(m,A(m+1,n)) \end{array}$$

Az Ackermann függvény néhány értéke a következő táblázatban látható:

$A(m,n)$	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$m=0$	1	2	3	4	5	6
$m=1$	2	3	4	5	6	7
$m=2$	3	5	7	9	11	13
$m=3$	5	13	29	61	125	253
$m=4$	13	65333	$2^{65336}-3$	$2^{2^{65536}}-3$	$A(3,2^{2^{65536}}-3)$	$A(3,A(4,4))$
$m=5$	65533	$A(4,65533)$	$A(4,A(4,65536))$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,4))$
$m=6$	$A(4,65533)$	$A(5,A(4,65533))$	$A(5,A(6,1))$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,4))$

Írjuk át a függvényt egyváltozós alakra:

$$\begin{array}{lcl} A_0(n) & = & n+1 \\ A_{m+1}(0) & = & A_m(1) \\ A_{m+1}(n+1) & = & A_m(A_{m+1}(n)) \end{array}$$

Nyilvánvalóan,

$A_0 \equiv \mathsf{succ}$,

elsőrendű zárt formulák, azaz mondatok halmaza, amelyek igazak $\mathcal{N}$-ben. A Peano Aritmetika (PA) egy kísérlet a $Th(\mathcal{N})$ axiomatizálására, azaz arra, hogy $Th(\mathcal{N})$ minden mondata a PA következménye. Egy f függvény bizonyíthatóan teljes PA-ban, ha $PA \vdash \forall\vec{x}\exists y(t_f(\vec{(x)},y)=0)$, ahol t_f az f-t kiszámító algoritmus, a jobboldali kifejezés azt jelöli, hogy ez a kiszámító algoritmus terminál.

és A_m leírása a λT rendszerben a következőképpen adható meg.

Legyen Ack egy olyan függvény, amelyre

$A_m \equiv \mathsf{Ack}\ \mathsf{c_m}$,

és

$A_{m+1} \equiv \lambda x\,.\,(\mathsf{recur}\ x\ (\mathsf{Ack}\ \mathsf{c_m c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y))$.

Ekkor

$A_{m+1}(0) \equiv (\lambda x\,.\,(\mathsf{recur}\ x\ (\mathsf{Ack}\ \mathsf{c_m c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y)))\ \mathsf{c_0} \twoheadrightarrow_\beta$
$\mathsf{recur}\ \mathsf{c_0}\ (\mathsf{Ack}\ \mathsf{c_m c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y) \rightarrow_T$
$\mathsf{Ack}\ \mathsf{c_m c_1} \equiv A_m(1)$,

és

$A_{m+1}(n+1) \equiv (\lambda x\,.\,(\mathsf{recur}\ x\ (\mathsf{Ack}\ \mathsf{c_m}\ \mathsf{c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y)))\ (\mathsf{succ}\ \mathsf{c_n}) \twoheadrightarrow_\beta$
$\mathsf{recur}\ (\mathsf{succ}\ \mathsf{c_n})\ (\mathsf{Ack}\ \mathsf{c_m}\ \mathsf{c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y) \rightarrow_T$
$(\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y)\ (\mathsf{recur}\ \mathsf{c_n}\ (\mathsf{Ack}\ \mathsf{c_m}\ \mathsf{c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y))\ \mathsf{c_n} \twoheadrightarrow_\beta$
$(\lambda z\,.\,(\mathsf{Ack}\ \mathsf{c_m}\ \ (\mathsf{recur}\ \mathsf{c_n}\ (\mathsf{Ack}\ \mathsf{c_m}\ \mathsf{c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y)))\mathsf{c_n} \twoheadrightarrow_\beta$
$\mathsf{Ack}\ \mathsf{c_m}\ \ (\mathsf{recur}\ \mathsf{c_n}\ (\mathsf{Ack}\ \mathsf{c_m}\ \mathsf{c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y)) \leftarrow_\beta$
$\mathsf{Ack}\ \mathsf{c_m}\ \ (\lambda x\,.\,\mathsf{recur}\ x\ (\mathsf{Ack}\ \mathsf{c_m}\ \mathsf{c_1})\ (\lambda yz\,.\,\mathsf{Ack}\ \mathsf{c_m}\ y))\ \mathsf{c_n}) \equiv$
$A_m(A_{m+1}(n))$.

Legyen G olyan, hogy

$A_{m+1} \equiv G(\mathsf{Ack}\ \mathsf{c_m})$.

Így az Ack, azaz az *Ackermann függvény* λT-beli alakja

$\mathsf{Ack} \equiv \lambda x\,.\,(\mathsf{recur}\ x\ \mathsf{succ}\ (\lambda yz\,.\,Gy))$,

hiszen

$A_0 \equiv \mathsf{Ack}\ \mathsf{c_0} \equiv (llx\,.\,(\mathsf{recur}\ x\ \mathsf{succ}\ (\lambda yz\,.\,Gy)))\ \mathsf{c_0} \twoheadrightarrow_\beta$

$\mathsf{recur}\ \mathsf{c_0}\ \mathsf{succ}\ (\lambda yz\,.\,Gy) \rightarrow_T\ \mathsf{succ}$

és

$A_{m+1} \equiv \mathsf{Ack}\ (\mathsf{succ}\ \mathsf{c_m} \equiv (llx\,.\,(\mathsf{recur}\ x\ \mathsf{succ}\ (\lambda yz\,.\,Gy)))\ (\mathsf{succ}\ \mathsf{c_m})\ \twoheadrightarrow_\beta$
$\mathsf{recur}\ (\mathsf{succ}\ \mathsf{c_m})\ \mathsf{succ}\ (\lambda yz\,.\,Gy)\ \rightarrow_T$
$(\lambda yz\,.\,Gy)\ (\mathsf{recur}\ \mathsf{c_m}\ \mathsf{succ}\ (\lambda yz\,.\,Gy)\ \mathsf{c_m}\ \twoheadrightarrow_\beta$
$(\lambda z\,.\,G\ (\mathsf{recur}\ \mathsf{c_m}\ \mathsf{succ}\ (\lambda yz\,.\,Gy)))\ \mathsf{c_m}\ \twoheadrightarrow_\beta$
$G\ (\mathsf{recur}\ \mathsf{c_m}\ \mathsf{succ}\ (\lambda yz\,.\,Gy))$

$G\ (\lambda x\,.\,\mathsf{recur}\ \mathsf{c_m}\ \mathsf{succ}\ (\lambda yz\,.\,Gy))\mathsf{c_m} \equiv$
$G(\mathsf{Ack}\ \mathsf{c_m})\mathsf{c_m} \equiv A_{m+1}$.

5.8.9. Példa. ...

$A(0, n)$
$A(m+1, 0)$
$A(m+1, n+1)$ □

6. FEJEZET

Curry-típusos λ-kalkulus

Az implicit típusos rendszer paradigmáját Curry adta meg 1934-ben [?] a kombinátor logikára, majd Curry és Feys [?], és később Curry, Hindley és Seldin [?] definiálta a típusnélküli λ-kalkulus kifejezéseire. Ebben a fejezetben először a Curry-típusos λ-kalkulust definiáljuk, majd megvizsgáljuk a Curry-típusos λ-kalkulus tulajdonságait, ezeket összehasonlítjuk a Church-típusos λ-kalkulus hasonló eredményeivel, végül a két típusos λ-kalkulus kapcsolatát elemezzük.

6.1. A Curry-típusrendszer

A Curry-típusos λ-kalkulust röviden Curry-típusrendszernek nevezzük. A Curry-típusrendszer típusai nem alaptípusokból, hanem *típusváltozókból* épülnek fel, ezért a Church-típusos λ-kalkulus, azaz az F_1 rendszer alaptípus halmazának elemeit – megkülönböztetésképpen – *típuskonstansoknak* is nevezhetjük. A típusváltozók konkrét típusértéket majd a *típuslevezetés* folyamán kapnak.

6.1.1. Definíció. A Curry-típusrendszer szintaxisa:

$$\begin{array}{lll}
<\textit{típus}> & ::= & <\textit{típusváltozó}> \\
 & | & (<\textit{típus}> \to <\textit{típus}>) \\
<\lambda\textit{-kifejezés}> & ::= & <\textit{változó}> \\
 & | & (\lambda\ <\textit{változó}> .\ <\lambda\textit{-kifejezés}>) \\
 & | & (<\lambda\textit{-kifejezés}>\ <\lambda\textit{-kifejezés}>)
\end{array}$$

A λ-kifejezésekre és a típusokra vonatkozó precedencia szabályok azonosak az F_1 rendszer megfelelő szabályaival.

Mint az 5.1. pontban láttuk, az F_1 rendszert röviden a

$$\mathbb{T} = \mathbb{U} \mid \mathbb{T} \to \mathbb{T},$$
$$\Lambda_{\mathbb{T}} = V \mid \lambda x : \mathbb{T} \, . \, \Lambda_{\mathbb{T}} \mid (\Lambda_{\mathbb{T}} \Lambda_{\mathbb{T}})$$

alakban írtuk le, ahol $\mathbb{T}$ a típusokat, $\mathbb{U}$ az alaptípusokat, $\Lambda_{\mathbb{T}}$ a kifejezéseket jelölte. A Curry-típusos λ-kifejezések hasonló stílusú leírása a következő:

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T},$$
$$\Lambda = V \mid \lambda x \, . \, \Lambda \mid (\Lambda \ \Lambda),$$

ahol $\mathbb{V}$ a típusváltozók, V a változók és Λ a kifejezések halmaza.

A típusváltozókat most is az $\alpha, \beta, \ldots$ betűkkel jelöljük, a típusváltozókat tartalmazó típuskifejezések jelölésére az $A, B, \ldots$ karaktereket használjuk. A típuskifejezéseket ebben a típusrendszerben *típussémáknak* is nevezhetjük.

A Γ típuskörnyezet típusváltozókból és olyan $x : A$ párok halmazából áll, ahol x a λ-kalkulus egy változója, A pedig az x változó típusa.

A *következtetések* $\Gamma \vdash wf$, $\Gamma \vdash A$ és $\Gamma \vdash E : A$ alakúak.

Mivel az $\Gamma \vdash E : A$ következtetésben az A tetszőleges $\mathbb{T}$-beli típust jelöl, a Curry-típusos λ-kalkulusban az u.n. *korlátozott polimorfizmusról* beszélhetünk, ami a következőket jelenti.

Ha az A típussémában az $\alpha_1, \alpha_2, \ldots, \alpha_m$ típusváltozók szerepelnek, akkor ez a típusséma valójában a

$$\forall \alpha_1 \, \forall \alpha_2 \, \ldots \, \forall \alpha_m \, . \, A \equiv \forall \alpha_1 \alpha_2 \ldots \alpha_m \, . \, A$$

kifejezésnek felel meg, azaz a típussémákban a $\forall$ kvantor *köti* a típusváltozókat. Így a típussémákban csak az univerzális kvantorok által *kötött* típusváltozók vannak, és a típusváltozó *hatásköre* a teljes típusséma.

Azt mondjuk, hogy egy $E : A$ következtetés *érvényes* a Γ típuskörnyezetben, azaz $\Gamma \vdash E : A$, ha bizonyítható a következő szabályok felhasználásával.

6.1.2. Definíció. A Curry-típusrendszer szabályai:

A környezetre vonatkozó szabályok:

$$\frac{}{\emptyset \vdash wf} \quad [\textsc{Env}\ \emptyset]$$

$$\frac{\Gamma \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash wf} \quad [\textsc{Env}\ x]$$

A típusra vonatkozó szabályok:

$$\frac{\Gamma \vdash wf \quad \alpha \in \mathbb{V}}{\Gamma \vdash \alpha} \quad [\textsc{Type Var}]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B} \quad [\textsc{Type Arrow}]$$

Kifejezés típusára vonatkozó szabályok:

$$\frac{\Gamma', x : A, \Gamma'' \vdash wf}{\Gamma', x : A, \Gamma'' \vdash x : A} \quad [\textsc{Val}\ x]$$

$$\frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x \,.\, E : A \to B} \quad [\textsc{Val Fun}]$$

$$\frac{\Gamma \vdash E : A \to B \quad \Gamma \vdash F : A}{\Gamma \vdash EF : B} \quad [\textsc{Val Appl}]$$

Az F_1 rendszer szabályaihoz hasonlóan, itt is látható, hogy a típusra vonatkozó szabályok (Type Var, Type Arrow) csak a szintaktikai szabályokat ismétlik meg, így, ha egy kifejezés A típusa jól formált, azaz szintaktikusan helyes, akkor a $\Gamma \vdash A$ következtetés mindig érvényes.

A Val Fun szabály feltételében levő következtetés típuskörnyezetében $\Gamma, x : A$ szerepel, ami azt jelenti, hogy az x nincs benne a Γ-ban, vagyis $x \notin dom(\Gamma)$. Látható, hogy a feltétel típuskörnyezetéből az $x : A$ itt is beépül a λ-absztrakcióba.

A Val Fun szabályt, mivel egy λ-kifejezésbe egy típust épít be, itt is „típusbevezetésnek”, a Val Appl szabályt pedig, mivel függvénytípusból egy egyszerű típust ad eredményül, „típuseliminációnak” is nevezzük.

6.1.3. Példa. *(Az* id_Nat $\equiv \lambda x\,.\,x$ *kifejezés típusának meghatározása)*

$$\cfrac{\cfrac{\cfrac{\cfrac{\emptyset \vdash wf \quad \alpha \in \mathbb{V}}{\emptyset \vdash \alpha} \qquad x \notin dom(\emptyset)}{x : \alpha \vdash wf}}{x : \alpha \vdash x : \alpha}}{\emptyset \vdash \lambda x\,.\,x : \alpha \to \alpha}$$

□

6.2. A Curry-típusrendszer operációs szemantikája

Az F_1 rendszerben leírtakkal azonos módon definiálhatjuk itt is a *szabad* és *kötött változókat* (5.6.1. Definíció) és a *helyettesítés* műveletet (5.6.2. Definíció).

Egy helyettesítés típushelyességét, az F_1 típusrendszerhez hasonlóan, a következő lemmával adhatjuk meg:

6.2.1. Lemma. (Helyettesítési lemma)

$$\frac{\Gamma, x : A \vdash E : B \quad \Gamma \vdash F : A}{\Gamma \vdash E[x := F] : B}$$

A lemma azt mondja ki, hogy egy helyettesítés a típus szempontjából csak akkor lesz helyes, ha a helyettesítendő változó típussémája megegyezik a behelyettesített kifejezés típussémájával. Az is látható, hogy a helyettesítés a kifejezés típussémáját nem változtatja meg.

A Curry-típusrendszerben a helyettesítést típusváltozókra is megadhatjuk.

6.2.2. Definíció. Típusváltozó helyettesítése:

1. $\alpha[\beta := B] \equiv \begin{cases} B, & \textit{ha } \alpha \equiv \beta, \\ \alpha, & \textit{egyébként,} \end{cases}$

2. $(A_1 \to A_2)[\beta := B] \equiv (\ A_1[\beta := B])(A_2[\beta := B])$.

Láttuk, hogy a típussémákban a típusváltozókat implicit módon egy univerzális kvantor köti, de a változókra definiált „kötött” vagy „szabad változó” fogalmakhoz hasonlóakról itt nem beszélhetünk, hiszen a típusváltozókkal nem definiáltunk függvényeket. Ezért a típusváltozó helyettesítését a tipusváltozó minden előfordulására el kell végeznünk. Tehát az $A[\alpha := B]$ azt jelöli, hogy az α típusváltozót az A típusban minden helyen a B típussal helyettesítjük. Természetesen ha A-ban nincs α típusváltozó, akkor az $A[\alpha := B]$ helyettesítés az A-t nem változtatja meg.

A típushelyettesítést definiálnunk kell a típuskörnyezetre is.

6.2.3. Definíció. Típusváltozó helyettesítése típuskörnyezetben:

$\Gamma[\alpha := B] \equiv \{(x : C[\alpha := B]) \mid (x : C) \in \Gamma\}$

$\Gamma[\alpha := B]$ tehát azt jelenti, hogy a Γ típuskörnyezet minden $x : C$ elemére végrehajtjuk a $C[\alpha := B]$ helyettesítést.

A típuskörnyezetre végrehajtott helyettesítés a kifejezések típusát is módosíthatja, ezt monda ki a következő lemma.

6.2.4. Lemma. (Típushelyettesítési lemma)

$$\frac{\Gamma \vdash E : A}{\Gamma[\alpha := B] \vdash E : A[\alpha := B]}$$

A váltózóra megadott helyettesítéssel definiálhatjuk az α-konverziót és a β-redukciót, és a redukciós sorozatokat.

Az F_1 rendszerhez hasonlóan, a Curry-típusrendszerben is igaz a következő állítás:

6.2.5. Tétel. (A tárgyredukció tétele)

Ha $\Gamma \vdash E : A$ *és* $E \twoheadrightarrow^* F$, *akkor* $\Gamma \vdash F : A$.

A tétel csak a jól típusozott E kifejezésekre igaz, azaz a $\Gamma \vdash E : A$ következtetésnek igaznak kell lennie. Ha E nem típusozható, akkor ettől az F kifejezésnek még lehet típusa. A tételben tehát a $\twoheadrightarrow^*$ iránya is lényeges, azaz a tétel csak β-redukciókra vonatkozik, az állítás β-absztrakciókra már nem teljesül. Példaként most is a 5.6.9. példában szereplő

$\mathsf{I} \leftarrow_\beta \mathsf{K I \Omega}$

β-absztrakció adható meg.

Ha egy kifejezésnek létezik típusa, akkor ez a típus egyértelmű.

6.2.6. Tétel. (Típusok egyértelműsége)

Ha $\Gamma \vdash E : A$ és $\Gamma \vdash E : B$, akkor $A \equiv B$.

Az F_1 rendszerben láttuk, hogy ha az $E \twoheadrightarrow^* F$ esetén az E és F mindegyike jól típusozott, akkor az E és F típusa mind a β-redukció mind a β-absztrakció alkalmazása esetén megegyezik. Ez az állítás a Curry-tipusrendszerben nem áll fenn.

6.2.7. Tétel. (Típusok egyértelműsége és a β-konverzió)

Ha $\Gamma \vdash E : A$, $E \twoheadrightarrow^* F$ és $\Gamma \vdash F : B$, akkor $A \not\equiv B$.

A tétel bizonyítására a következő példát adjuk.

6.2.8. Példa. *(β-konverzió esetén a típusok nem azonosak)*

Nyilvánvaló, hogy

$\mathsf{SK} \equiv (\lambda xyz\,.\,xy(xz))(\lambda uv\,.\,u) \twoheadrightarrow^+$
$\lambda yz\,.\,z,$

és

$\mathsf{SK} : (\beta \twoheadrightarrow \alpha) \twoheadrightarrow (\beta \twoheadrightarrow \beta),$

ugyanakkor

$\lambda yz\,.\,z : \alpha \twoheadrightarrow (\beta \twoheadrightarrow \beta).$ □

6.3. A Church- és Curry-típusos λ-kalkulus kapcsolata

Bár a Church- és a Curry-típusos λ-kalkulus között nagyon sok különbség van, a két kalkulus között viszonylag egyszerűen lehet kapcsolatot teremteni. A kapcsolat lényege az, hogy a Church-típusos λ-kalkulus egy helyesen típusozott kifejezéséből az absztrakció változójának típusát elhagyva a Curry-típusos λ-kalkulus kifejezését kapjuk, és a Curry-típusos λ-kalkulus egy kifejezéséből a Church-típusos λ-kalkulus egy kifejezése lesz, ha az absztrakciók változóinak típusait megadjuk.

Jelöljük a Church-típusos λ-kalkulus kifejezéseinek halmazát $\Lambda_{\mathbb{T}}$-vel, a következtetéseket $\vdash_{Church}$-csel, a Curry-típusos λ-kalkulus kifejezéseit pedig Λ-val, a következtetéseket $\vdash_{Curry}$-val. Legyen $|\cdot|$ a következő, „felejtésnek”, vagy „törlésnek” is nevezhető függvény:

$$|\cdot| : \Lambda_{\mathbb{T}} \Rightarrow \Lambda,$$

ahol

$$\begin{array}{ll} |x| & \mapsto x \\ |EF| & \mapsto |E|\,|F| \\ |\lambda x : A.E| & \mapsto \lambda x.\,|E| \end{array}$$

Látható, hogy ez a függvény a $\Lambda_{\mathbb{T}}$ kifejezéseit változtatja meg, az absztrakciók változóinak a típusát hagyja el, azaz a típusos λ-kifejezésekből típusnélküli kifejezések lesznek. A függvény definíciójából bizonyítható a következő két állítás.

6.3.1. Tétel. (Church $\rightarrow$ Curry konverzió)

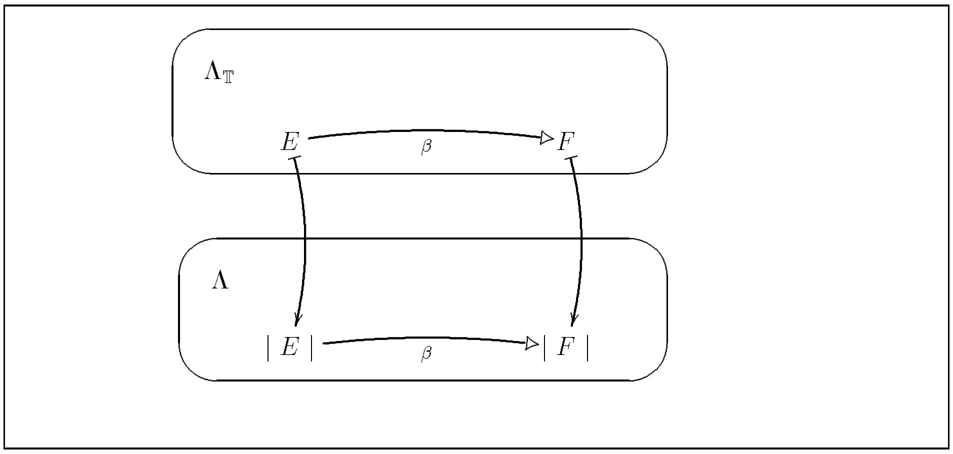
Legyen $E \in \Lambda_{\mathbb{T}}$. Ha $\Gamma \vdash_{Church} E : A$, akkor $\Gamma \vdash_{Curry} |E| : A$.

6.3.2. Tétel. (Curry $\rightarrow$ Church konverzió)

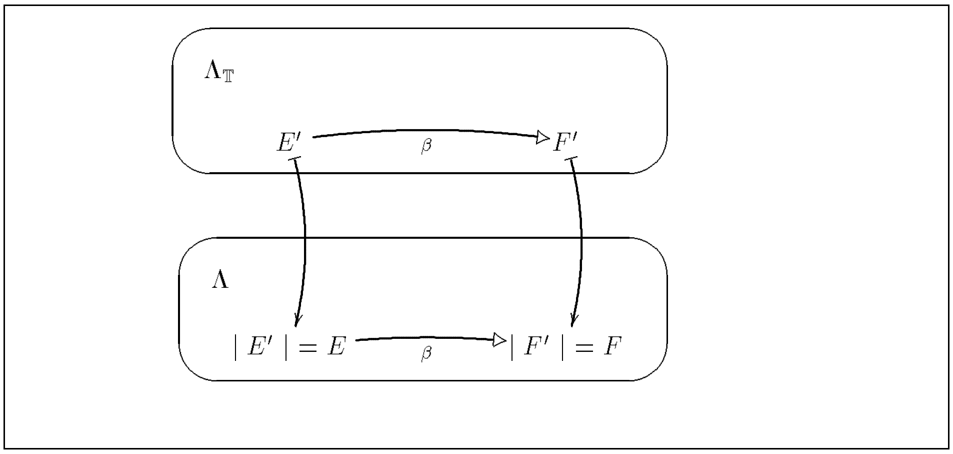
Legyen $E \in \Lambda$. Ha $\Gamma \vdash_{Curry} E : A$, akkor $\exists E' \in \Lambda_{\mathbb{T}}$, melyre $\Gamma \vdash_{Church} E' : A$, és $|E'| \mapsto E$.

Könnyen belátható az is, hogy a $|\cdot|$ függvény a kifejezések redukálását nem változtatja meg, azaz

- ha $E, F \in \Lambda_{\mathbb{T}}$ és $E \twoheadrightarrow_\beta F$, akkor $|E| \twoheadrightarrow_\beta |F|$ (6.1. ábra),
- ha $E, F \in \Lambda$ és $E \twoheadrightarrow_\beta F$, akkor minden $E' \in \Lambda_{\mathbb{T}}$-hez, melyre $|E'| \equiv E$, létezik olyan $F' \in \Lambda_{\mathbb{T}}$, amelyre $E' \twoheadrightarrow_\beta F'$ és $|F'| \equiv F$ (6.2. ábra).



6.1. ábra. $\Lambda_{\mathbb{T}} \Mapsto \Lambda$ és a $\Lambda_{\mathbb{T}}$ β-redukciója



6.2. ábra. $\Lambda_{\mathbb{T}} \Mapsto \Lambda$ és a Λ β-redukciója

7. FEJEZET

Másodrendű (polimorfikus) típusos λ-kalkulus

Tekintsük a következő F_1-beli kifejezést:

$\mathsf{id\_Nat} \equiv \lambda x : Nat.x.$

Ez az *identitás* függvény a természetes számokra. Ha az identitás függvényt meg akarjuk adni *Int*, vagy *Bool* típusra, akkor olyan F_1-beli kifejezéseket kell megadnunk, amelyek lényegében megegyeznek a fenti kifejezéssel, csak a *Nat* helyett *Int*-et vagy *Bool*-t kell írnunk.

$$\begin{aligned}\mathsf{id\_Int} &\equiv \lambda x : Int.x,\\ \mathsf{id\_Bool} &\equiv \lambda x : Bool.x.\end{aligned}$$

Ez természetesen igaz tetszőleges típusra is, a típust mindig a *Nat* helyére kell írnunk.

Azért, hogy ne kelljen ilyen sok identitás függvényt megadnunk, jelöljük a típust egy α *típusváltozóval*:

$\mathsf{id\_}\alpha \equiv \lambda x : \alpha.x,$

ahol az α típusváltozó lehetséges értékei például a fenti típusok.

A típusváltozókat tartalmazó λ-kifejezéseket *típussémáknak* nevezzük.

A típusnélküli λ-kalkulusban egy λ-absztrakció x változója úgy kapott értéket, hogy először felírtunk egy applikációt, amelynek az első tagja ez a λ-absztrakció, a második tagja, azaz az argumentum pedig az érték volt, majd erre az applikációra egy β-redukciót hajtottunk végre. Ehhez a módszerhez hasonlóan tudunk a típusváltozóhoz is értéket rendelni.

Először készítsünk az α típusváltozóval egy absztrakciót:

$$\mathsf{id} \equiv \Lambda\alpha.\lambda x : \alpha.x,$$

ahol a típusváltozó absztrakcióját Λ-val jelöltük, azért nem λ-val, mert ehhez az absztrakcióhoz tartozó argumentum nem λ-kifejezés, hanem egy típus lesz.

Ezt az absztrakciót *típusabsztrakciónak*, a típusabsztrakcióval definiált függvényeket *polimorfikus függvényeknek* nevezzük. A fenti id kifejezés neve: polimorfikus identitás függvény.

Egy típusabsztrakcióval és egy típussal, mint egy függvénnyel és a függvény argumentumával képzett applikáció neve *típusapplikáció*.

Egy típusapplikációban a típusabsztrakció argumentumát [ ] zárójelbe tesszük, hangsúlyozva, hogy a kifejezésben ez egy típus, és nem λ-kifejezés.

A típusapplikáció β-redukcióját a λ-kifejezések applikációjának β-redukciójához hasonlóan definiálhatjuk, ekkor a típusabsztrakció az argumentumban megadott típust egy λ-kifejezésre képezi le.

Így az id típusabsztrakció az argumentumként adott típus identitás függvényét adja:

$$\begin{aligned}
\mathsf{id}\ [Nat] &\twoheadrightarrow_\beta \mathsf{id_Nat},\\
\mathsf{id}\ [Int] &\twoheadrightarrow_\beta \mathsf{id_Int},\\
\mathsf{id}\ [Bool] &\twoheadrightarrow_\beta \mathsf{id_Bool},
\end{aligned}$$

és például

$$\mathsf{id}\ [Nat]\ 3 \equiv \underline{(\Lambda\alpha.\lambda x : \alpha.x)\ [Nat]}\ 3 \twoheadrightarrow_\beta \underline{(\lambda x : Nat.x)\ 3} \equiv \mathsf{id\_Nat}\ 3 \twoheadrightarrow_\beta 3.$$

Ha a típusos λ-kalkulust kiegészítjük a polimorfikus függvények kifejezéseivel, akkor meg kell adnunk e függvények típusát is. Mint láttuk, ezek a függvények a típust egy λ-kifejezésre képezik le. Az id függvény a Nat-ot egy $Nat \to Nat$ típusú függvényre, általánosan, az A típust egy

$A \to A$ típusú függvényre képezi le, és látható, hogy az eredmény típusa csak az A-tól függ.

Ezt a tulajdonságot, azaz az id függvény típusát úgy jelöljük, hogy

$$\mathsf{id} : \forall\alpha.\alpha \to \alpha.$$

Megjegyezzük, hogy egy polimorfikus függvény $\forall\alpha.A$ típusát gyakran $\lambda[\alpha]A$-val, $\Lambda\alpha.A$-val, $\Delta\alpha.A$-val, és $\forall[\alpha]A$-val is jelölték.

7.1. Az F_2 rendszer szintaxisa

A típusabsztrakcióval, a típusapplikációval és a típusabsztrakció típusával bővített típusos λ-kalkulust F_2 *rendszernek*, *másodrendű típusos λ-kalkulusnak*, vagy *polimorfikus λ-kalkulusnak* nevezzük.

7.1.1. Definíció. Az F_2 rendszer szintaxisa:

$$\begin{array}{lcl}
<\textit{típus}> & ::= & <\textit{típusváltozó}> \\
 & | & (\ <\textit{típus}> \to <\textit{típus}>\ ) \\
 & | & (\ \forall <\textit{típusváltozó}>\ .\ <\textit{típus}>\ ) \\
<\lambda\textit{-kifejezés}> & ::= & <\textit{változó}> \\
 & | & (\ \lambda\ \ <\textit{változó}>:<\textit{típus}>\ .\ <\lambda\textit{-kifejezés}>\ ) \\
 & | & (\ <\lambda\textit{-kifejezés}>\ <\lambda\textit{-kifejezés}>\ ) \\
 & | & (\ \Lambda <\textit{típusváltozó}>\ .\ <\lambda\textit{-kifejezés}>\ ) \\
 & | & (\ <\lambda\textit{-kifejezés}>\ [\ <\textit{típus}>\ ]\ \ )
\end{array}$$

A legkülső zárójelpárt elhagyhatjuk. A továbbiakban a típusváltozókat az $\alpha, \beta, \ldots$ görög kisbetűkkel jelöljük.

A fentiekben a típusabsztrakciót polimorfikus függvénynek neveztük, egy típusabsztrakció, azaz polimorfikus függvény típusát a továbbiakban röviden *polimorfikus függvénytípusnak* nevezzük. Tehát a $\forall\alpha\,.\,\alpha \to \alpha$ egy polimorfikus függvénytípus, az id polimorfikus függvénynek a típusa.

A $\Lambda\alpha\,.\,E$ típusabsztrakciónak, azaz polimorfikus függvénynek a típusát a $\forall\alpha\,.\,A$ típus írja le, ahol A az E λ-kifejezés típusa. Az α változót a polimorfikus függvénytípus *változójának* vagy *formális paraméterének*, az A típust a függvénytípus *törzsének* nevezzük. A $\forall\alpha\,.\,A$ polimorfikus függvénytípusban az A típus az α változó *hatásköre*, és azt mondjuk, hogy a hatáskörbe tartozó α változókat a polimorfikus függvénytípus α változója köti.

A polimorfikus függvénytípusban az A típus nagyobb *precedenciájú*, mint a típusváltozó, így a redundáns zárójelpárok a törzsében elhagyhatók:

$$\forall\alpha\,.\,(\alpha \to \alpha) \equiv \forall\alpha\,.\,\alpha \to \alpha.$$

A $\Lambda\alpha\,.\,E$ típusabsztrakcióban az α változót a típusabsztrakció *változójának* vagy *formális paraméterének*, az E λ-kifejezést a típusabsztrakció *törzsének* nevezzük.

Az E kifejezés a típusabsztrakció α változójának a *hatásköre*, és azt mondjuk, hogy a hatáskörbe tartozó α változót a típusabsztrakció α változója köti. A típusabsztrakció *jobbasszociatív*, például, ha az E kifejezés is egy $\Lambda\beta\,.\,F$ típusabsztrakció, akkor $\Lambda\alpha\,.\,(\Lambda\beta\,.\,F)$ röviden így is írható:

$$\Lambda\alpha\,.\,(\Lambda\beta\,.\,F) \equiv \Lambda\alpha\,.\,\Lambda\beta\,.\,F \equiv \Lambda\alpha\beta\,.\,F,$$

azaz a redundáns zárójelpárok elhagyhatók, és a típusabsztrakció lambdái összevonhatók.

A típusapplikáció *balasszociatív*, így például

$$(E[A])[B] \equiv E[A][B],$$

azaz a redundáns zárójelpárok elhagyhatók.

A típusapplikációnak nagyobb a *precedenciája*, mint a típusabsztrakciónak, ezt figyelembevéve a redundáns zárójelpárok a típusabsztrakció törzsében is elhagyhatók:

$$\Lambda\alpha\,.\,(E[A]) \equiv \Lambda\alpha\,.\,E[A].$$

7.1.2. Példa. *(Polimorfikus függvénytípus)*

A 5.8. pontban láttuk, hogy 2-t reprezentáló Church-féle 2-es szám, vagy más szemlélettel a $\mathsf{double} \equiv \lambda fx\,.\,f(fx)$ függvény, F_1-beli alakja

$\lambda f : Nat \to Nat \,.\, \lambda x : Nat \,.\, f(fx)$

volt, és ehhez a függvényhez az

$(Nat \to Nat) \to Nat \to Nat$

típust rendeltük hozzá. Ha a Nat helyett az α típusváltozót használjuk, akkor

$\lambda f : \alpha \to \alpha \,.\, \lambda x : \alpha \,.\, f(fx),$

és ennek a típusa

$(\alpha \to \alpha) \to \alpha \to \alpha.$

A λ-kifejezésből típusabsztrakcióval a

$\Lambda\alpha \,.\, \lambda f : \alpha \to \alpha \,.\, \lambda x : \alpha \,.\, f(fx)$

polimorfikus függvényt kapjuk, melynek a típusa

$\forall\alpha \,.\, (\alpha \to \alpha) \to \alpha \to \alpha.$ □

7.2. Az F_2 típusrendszer

Az F_2 rendszer következtetései megegyeznek az F_1 rendszer következtetéseivel.

7.2.1. Definíció. Az F_2 rendszer következtetései:

$\Gamma \vdash wf$	Γ jól formált környezet
$\Gamma \vdash A$	Γ-ban az A jól formált típus
$\Gamma \vdash E : A$	Γ-ban az E kifejezés típusa A

A következtetések tehát formailag megegyeznek, de az F_2 rendszerben a Γ típuskörnyezet bővebb, mint az F_1-ben, itt a típuskörnyezet már típusváltozókat is tartalmaz.

7.2.2. Definíció. Az F_2 rendszer szabályai:

A környezetre vonatkozó szabályok:

$$\frac{}{\emptyset \vdash wf} \quad [\textsc{Env } \emptyset]$$

$$\frac{\Gamma \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash wf} \quad [\textsc{Env } x]$$

$$\frac{\Gamma \ \vdash wf \quad \alpha \notin Dom(\Gamma)}{\Gamma, \alpha \vdash wf} \quad [\textsc{Env } \alpha]$$

A típusra vonatkozó szabályok:

$$\frac{\Gamma', \alpha, \Gamma'' \vdash \ wf}{\Gamma', \alpha, \Gamma'' \vdash \alpha} \quad [\textsc{Type } \alpha]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B} \quad [\textsc{Type Arrow}]$$

$$\frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall \alpha \,.\, A} \quad [\textsc{Type Forall}]$$

Kifejezés típusára vonatkozó szabályok:

$$\frac{\Gamma', x : A, \Gamma'' \vdash wf}{\Gamma', x : A, \Gamma'' \vdash x : A} \quad [\textsc{Val } x]$$

$$\frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A \,.\, E : A \to B} \quad [\textsc{Val Fun}]$$

$$\frac{\Gamma \vdash E : A \to B \quad \Gamma \vdash F : A}{\Gamma \vdash EF : B} \quad [\textsc{Val Appl}]$$

$$\frac{\Gamma, \alpha \vdash E : A}{\Gamma \vdash \Lambda \alpha \,.\, E : \forall \alpha \,.\, A} \quad [\textsc{Val Fun2}]$$

$$\frac{\Gamma \vdash E : \forall \alpha \,.\, A \quad \Gamma \vdash B}{\Gamma \vdash E[B] : A[\alpha := B]} \quad [\textsc{Val Appl2}]$$

FÜGGELÉK

A. FÜGGELÉK

A könyvben alkalmazott jelölések

$x, y, \ldots$	változó
$E, F, \ldots$	λ-kifejezés
$\alpha, \beta, \ldots$	típusváltozó
$A, B, \ldots$	típus (típuskifejezés)
$\Gamma, \Delta, \ldots$	típuskörnyezet
$\rightarrow$	típuskonstruktor
$\rightharpoonup$	redukció

B. FÜGGELÉK

A típusrendszerek összefoglaló leírása

I. A TÍPUSRENDSZEREK SZINTAXISA

1. A típusnélküli λ-kalkulus szintaxisa:

$$\begin{array}{lcl}
<\lambda\text{-}\textit{kifejezés}> & ::= & <\textit{változó}> \\
 & | & \lambda <\textit{változó}> \,.\, <\lambda\text{-}\textit{kifejezés}> \\
 & | & <\lambda\text{-}\textit{kifejezés}> <\lambda\text{-}\textit{kifejezés}>
\end{array}$$

2. Az F_1 rendszer szintaxisa:

$$\begin{array}{lcl}
\boldsymbol{<\textbf{\textit{típus}}>} & ::= & \boldsymbol{<\textbf{\textit{alaptípus}}>} \\
 & | & \boldsymbol{<\textbf{\textit{típus}}>} \to \boldsymbol{<\textbf{\textit{típus}}>}
\end{array}$$

$$\begin{array}{lcl}
<\lambda\text{-}\textit{kifejezés}> & ::= & <\textit{változó}> \\
 & | & \lambda <\textit{változó}> \boldsymbol{:} \boldsymbol{<\textbf{\textit{típus}}>} \,.\, <\lambda\text{-}\textit{kifejezés}> \\
 & | & <\lambda\text{-}\textit{kifejezés}> <\lambda\text{-}\textit{kifejezés}>
\end{array}$$

3. Az F_2 rendszer szintaxisa:

$$
\begin{array}{lcl}
<\textit{ típus }> & ::= & \boldsymbol{<\textit{típusváltozó}>} \\
 & \mid & <\textit{ típus }> \to <\textit{ típus }> \\
 & \mid & \forall \boldsymbol{<\textit{típusváltozó}>\ .\ <\textit{típus}>} \\
 & & \\
<\lambda\textit{-kifejezés}> & ::= & <\textit{ változó }> \\
 & \mid & \lambda <\textit{ változó }> : <\textit{ típus }>\ .\ <\lambda\textit{-kifejezés}> \\
 & \mid & <\lambda\textit{-kifejezés}> <\lambda\textit{-kifejezés}> \\
 & \mid & \boldsymbol{\Lambda <\textit{típusváltozó}>\ .\ <\lambda\textit{-kifejezés}>} \\
 & \mid & \boldsymbol{<\lambda\textit{-kifejezés}>\ [\ <\textit{típus}>\ ]}
\end{array}
$$

4. Az F_3 rendszer szintaxisa:

$$
\begin{array}{lcl}
\boldsymbol{<\textit{fajta}>} & ::= & * \\
 & \mid & * \to \boldsymbol{<\textit{fajta}>} \\
 & & \\
<\textit{ típus }> & ::= & <\textit{ típusváltozó }> \\
 & \mid & <\textit{ típus }> \to <\textit{ típus }> \\
 & \mid & \forall <\textit{ típusváltozó }> \boldsymbol{:\ <\textit{fajta}>}\ .\ <\textit{ típus }> \\
 & \mid & \boldsymbol{\Lambda <\textit{típusváltozó}>\ .\ <\textit{típus}>} \\
 & \mid & <\textit{ típus }> <\textit{ típus }> \\
 & & \\
<\lambda\textit{-kifejezés}> & ::= & <\textit{ változó }> \\
 & \mid & \lambda <\textit{ változó }> : <\textit{ típus }>\ .\ <\lambda\textit{-kifejezés}> \\
 & \mid & <\lambda\textit{-kifejezés}> <\lambda\textit{-kifejezés}> \\
 & \mid & \Lambda <\textit{ típusváltozó }> \boldsymbol{:\ <\textit{fajta}>}\ .\ <\lambda\textit{-kifejezés}> \\
 & \mid & <\lambda\textit{-kifejezés}>\ [\ <\textit{ típus }>\ ]
\end{array}
$$

5. Az F_ω rendszer szintaxisa:

$$
\begin{array}{lcl}
<\textit{fajta}> & ::= & * \\
 & | & \boldsymbol{<fajta> \to <fajta>} \\
 & & \\
<\textit{típus}> & ::= & <\textit{típusváltozó}> \\
 & | & <\textit{típus}> \to <\textit{típus}> \\
 & | & \forall <\textit{típusváltozó}> : <\textit{fajta}> \;.\; <\textit{típus}> \\
 & | & \Lambda <\textit{típusváltozó}> \;.\; <\textit{típus}> \\
 & | & <\textit{típus}> <\textit{típus}> \\
 & & \\
<\lambda\textit{-kifejezés}> & ::= & <\textit{változó}> \\
 & | & \lambda <\textit{változó}> : <\textit{típus}> \;.\; <\lambda\textit{-kifejezés}> \\
 & | & <\lambda\textit{-kifejezés}> <\lambda\textit{-kifejezés}> \\
 & | & \Lambda <\textit{típusváltozó}> : <\textit{fajta}> \;.\; <\lambda\textit{-kifejezés}> \\
 & | & <\lambda\textit{-kifejezés}> \; [<\textit{típus}>]
\end{array}
$$

II. A TÍPUSRENDSZEREK KÖVETKEZTETÉSEI

1. Az F_1 rendszer következtetései

$\Gamma \vdash wf$	Γ jól formált környezet
$\Gamma \vdash A$	Γ-ban az A jól formált típus
$\Gamma \vdash E : A$	Γ-ban az E kifejezés típusa A

2. Az F_2 rendszer következtetései:

$\Gamma \vdash wf$	Γ jól formált környezet
$\Gamma \vdash A$	Γ-ban az A jól formált típus
$\Gamma \vdash E : A$	Γ-ban az E kifejezés típusa A

A következtetések formailag megegyeznek az F_1 következtetéseivel. de az F_2 rendszerben a Γ típuskörnyezet bővebb, mint az F_1-ben, itt a típuskörnyezet típusváltozókat is tartalmaz.

III. A TÍPUSRENDSZEREK SZABÁLYAI

1. Az F_1 rendszer szabályai:

(a) *A környezetre vonatkozó szabályok:*

$$\frac{}{\emptyset \vdash wf} \quad [\textsc{Env}\ \emptyset]$$

$$\frac{\Gamma \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash wf} \quad [\textsc{Env}\ x]$$

(b) *A típusra vonatkozó szabályok:*

$$\frac{\Gamma \vdash wf \quad A \in alaptípus}{\Gamma \vdash A} \quad [\textsc{Type Const}]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B} \quad [\textsc{Type Arrow}]$$

(c) *Kifejezés típusára vonatkozó szabályok:*

$$\frac{\Gamma', x : A, \Gamma'' \vdash wf}{\Gamma', x : A, \Gamma'' \vdash x : A} \quad [\textsc{Val}\ x]$$

$$\frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A \,.\, E : A \to B} \quad [\textsc{Val Fun}]$$

$$\frac{\Gamma \vdash E : A \to B \quad \Gamma \vdash F : A}{\Gamma \vdash EF : B} \quad [\textsc{Val Appl}]$$

2. Az F_2 rendszer szabályai:

(a) *A környezetre vonatkozó szabályok:*

$$\frac{}{\emptyset \vdash wf} \quad [\textsc{Env } \emptyset]$$

$$\frac{\Gamma \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash wf} \quad [\textsc{Env } x]$$

$$\frac{\Gamma \ \vdash wf \quad \alpha \notin Dom(\Gamma)}{\Gamma, \alpha \vdash wf} \quad [\textsc{Env } \alpha]$$

(b) *A típusra vonatkozó szabályok:*

$$\frac{\Gamma', \alpha, \Gamma'' \vdash \ wf}{\Gamma', \alpha, \Gamma'' \vdash \alpha} \quad [\textsc{Type } \alpha]$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B} \quad [\textsc{Type Arrow}]$$

$$\frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \forall \alpha \,.\, A} \quad [\textsc{Type Forall}]$$

(c) *Kifejezés típusára vonatkozó szabályok:*

$$\frac{\Gamma', x : A, \Gamma'' \vdash wf}{\Gamma', x : A, \Gamma'' \vdash x : A} \quad [\textsc{Val } x]$$

$$\frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x : A \,.\, E : A \to B} \quad [\textsc{Val Fun}]$$

$$\frac{\Gamma \vdash E : A \to B \quad \Gamma \vdash F : A}{\Gamma \vdash EF : B} \quad [\textsc{Val Appl}]$$

$$\frac{\Gamma, \alpha \vdash E : A}{\Gamma \vdash \Lambda \alpha \,.\, E : \forall \alpha \,.\, A} \quad [\textsc{Val Fun2}]$$

$$\frac{\Gamma \vdash E : \forall\alpha\,.\,A \quad \Gamma \vdash B}{\Gamma \vdash E[B] : A[\alpha := B]} \quad [\textsc{Val Appl2}]$$

C. FÜGGELÉK

Definíciók, tételek jegyzéke

Irodalom

[1] Church, Alonzo, A formulation of the simple theory of types, *Journal of Symbolic Logic*, 5 : 56-58, 1940.

[2] Curry, Haskell B., Functionality in combinatory logic, *Proc. Nat. Acad. Science USA*, 20 : 584-590, 1934.

[3] Milner, Robin., A theory of type polimorphism in programming. *Journal of Computer and System Sciences*, 17 : 348-375, August 1978.

Névmutató

Tárgymutató