|  |  |
| --- | --- |
|  | **EÖTVÖS LORÁND TUDOMÁNYEGYETEM**  Informatikai Kar  Programozási Nyelvek és Fordítóprogramok Tanszék |

**Praktikus programozás SKI kombinátor kalkulussal**

**Témavezető: Készítette:**

Kaposi Ambrus Fülöp Olivér - PG8SJ5

egyetemi docens programtervező informatikus MSc szak

**Budapest, 2023**

# 1. Tartalomjegyzék

[1. Tartalomjegyzék 1](#_Toc146228388)

[2. Bevezetés 2](#_Toc146228389)

[3. A kutatás háttere – SKI kombinátor kalkulus 3](#_Toc146228390)

[3. 1. Története 3](#_Toc146228391)

[3. 2. Elemei és működése 3](#_Toc146228392)

[3. 3. Irodalmi áttekintés 5](#_Toc146228393)

[3. 4. Előnyök, hátrányok 6](#_Toc146228394)

[4. Módszerek 7](#_Toc146228395)

[4. 1. Típusos SKI 7](#_Toc146228396)

[4. 2. Mély beágyazás 7](#_Toc146228397)

[4. 2. 1. Típusok 7](#_Toc146228398)

[4. 2. 2. Termek 12](#_Toc146228399)

[7. Irodalomjegyzék 13](#_Toc146228400)

.

# 2. Bevezetés

Diplomamunkámként egy gyakorlatban is használható, az SKI kombinátorokra épülő típusos kifejezésnyelvet készítettem, amely nem tartalmaz változókat. A nyelvhez parsert, típusellenőrzőt és egy kiértékelőt is implementáltam Java nyelven.

Az elkészült nyelvben példaként egyszerű programokat adtam meg és értékeltem ki és fő kérdésként azt vizsgáltam, hogy lehetséges-e egy SKI kombinátor alapú nyelvvel programozni és milyen módszerekkel lehet nagyobb programokat egyszerűbben és olvashatóan megadni. A kutatásban felhasznált módszerek elsősorban a típusparaméterek használata és a típuskikövetkeztetés, valamint összehasonlítottam a nyelv két különböző, mély és sekély implementációját.

# 3. A kutatás háttere – SKI kombinátor kalkulus

## 3. 1. Története

Az SKI kombinátor kalkulus az elméleti számítástudomány és matematikai logika egyik alapvető számítási módszere. Az 1920-as években Moses Schönfinkel mutatta be és később az 1930-as években Haskell Curry kiegészítette.

Történelmileg, az SKI kalkulus eredete David Hilbert munkájára vezethető vissza. Egy formális rendszert szeretett volna létrehozni matematikai bizonyítások kifejezésére, amely a lehető legkevesebb axiómát és következtetési szabályt tartalmazza. Schönfinkelnek Hilbert munkája adta az ihletet az SKI kalkulus elkészítéséhez, amellyel változók használata nélkül tudott függvényeket kifejezni. Az SKI elnevezés a három alapkombinátorból következik: S, K és I.

## 3. 2. Elemei és működése

A kombinátor egy magasabb rendű függvény, amely csak függvény alkalmazásokat és korábban meghatározott kombinátorokat használ az eredmény kiszámításához. Az SKI kalkulus primitív kombinátorai az S, K és I, melyekkel az alapvető müveletek végezhetőek el. E három alapkombinátor kombinációjával és a paramétereken való alkalmazásával, komplex számításokat lehet kifejezni és kiértékelni. A fő ötlet, hogy bármely kiszámítható függvény reprezentálható a három alapkombinátorral, mivel a lambda-kalkulus esetén a Church-Rosser tétel garantálja, hogy a számítások eredményei nem függenek az alkalmazott redukciós lépések sorrendjétől és hogy a lambda-kalkulus és a kombinátor logika algebrailag ekvivalens egymással [1], tehát bármely lambda kifejezésnek van egy ekvivalens SKI kombinátor kifejezése, és fordítva, bármely SKI kombinátor kifejezésnek van egy ekvivalens lambda kifejezése.

Az S kombinátor függvények kompozícióját és alkalmazását fejezi ki. Három paramétert vár, az első paramétert alkalmazza a harmadikra, majd ezt alkalmazza a második és a harmadik paraméterek alkalmazásának eredményére.

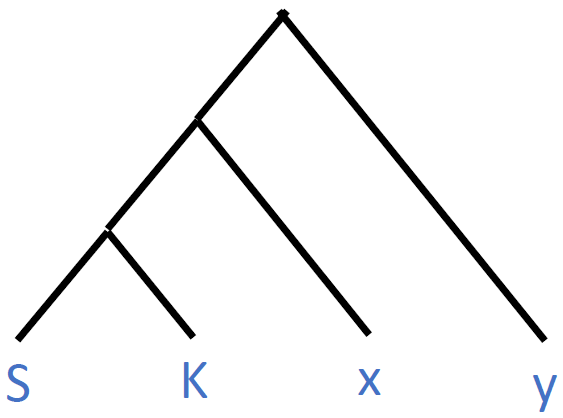
A példákban az egyes termek közötti szóközök a függvényalkalmazás műveletét jelölik.

A K kombinátor a konstans függvényként viselkedik. A két paramétere közül mindig az elsővel tér vissza, a másodikat figyelmen kívül hagyja.

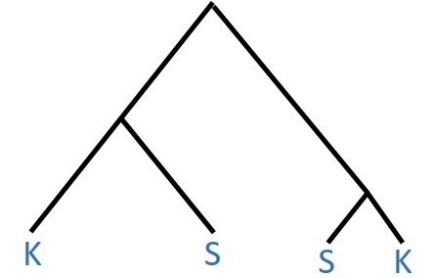
Az I kombinátor pedig az identitás függvényt reprezentálja, vagyis egyszerűen csak a kapott paraméterével tér vissza.

Az SKI kalkulus nem a minimális rendszer, ami megegyezik a lambda-kalkulus számítási képességeivel, mivel az I kombinátor kifejezhető az S K K vagy S K S kifejezésekkel. I kombinátort akár el is lehetne hagyni, csak kényelemből használjuk és azért, hogy a kifejezések rövidebbek legyenek.

Az SKI kifejezések bináris fákként reprezentálhatóak, amelynek a gyökere és csúcsai az egyes függvény alkalmazások, levelei pedig a kombinátorok és paramétereik. A közös csúcsokhoz tartozó részfák az írásos formában bezárójelezhetőek, de az olvashatóság kedvéért csak abban az esetben írjuk ki a zárójeleket, ha azok tényleg szükségesek. Zárójelek nélkül a kifejezések bal asszociatívak, tehát az kifejezést -ként kell értelmezni.



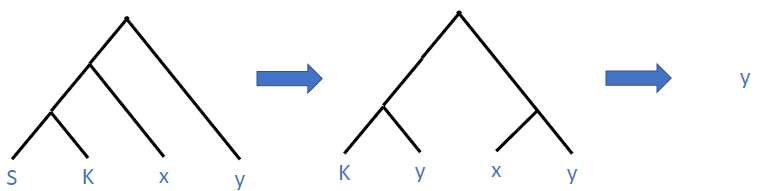
Például a helyett elég csak -t írni.

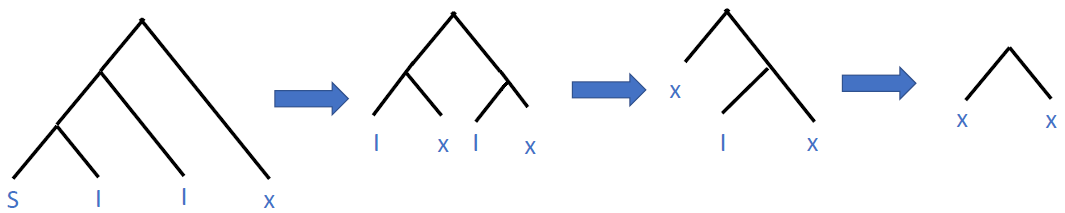


Az egyes kombinátorokon csak akkor lehet átírási szabályokat alkalmazni, ha azok a megfelelő számú paramétert kapták, S: 3 db., K: 2 db. és I: 1 db.

Például a következő kifejezések esetén nem tudunk átírási szabályt alkalmazni: vagy , de a esetben már igen: . Ha egy kombinátornak a szükségesnél több paraméter áll rendelkezésére, akkor csak a kellő számút használja fel az átírási szabály alkalmazásához, a maradék paramétereket pedig helyben hagyja.

Néhány további példa az SKI kifejezések kiértékelésére az átírási szabályok használatával:





## 3. 3. Irodalmi áttekintés

Az SKI kalkulus tulajdonságai már jóval az első számítógép megjelenése előtt ismertek voltak. Szorosan kapcsolódik a lambda-kalkulushoz és funkcionális programozási nyelvek alapjául szolgál.

Néhány példa a kombinátorok gyakorlati használatára:

* Az első kombinátor alapú programozási nyelv az APL volt, az 1960-as évekből. Arra lett tervezve, hogy műveletsorozatokat lehessen definiálni nagy méretű adathalmazokra. Az alapvető adattípusa a többdimenziós tömb volt. Hatással volt későbbi programozási nyelvekre, többek között a MatLab-ra és MapReduce-ra.
* NumPy: egy kiegészítő csomag a Python programozási nyelvhez, főleg többdimenziós tömbök műveleteinek elvégzésére, kombinátorok segítségével.
* FP (Functional Programming) egy kombinátor alapú funkcionális nyelv.
* MapReduce: a Google által készített kombinátor nyelv big data feldolgozására, párhuzamos, elosztott számítási algoritmusok használatával.
* A Haskell fordítóprogramja, a GHC bizonyos esetekben a kódot egy kombinátor alapú formára fordítja optimalizáció céljából.
* Az Ulambda egy majdnem tisztán funkcionális programozási nyelv. Kombinátor logikán alapszik, egy lambda operátorok és szabad változók nélküli kifejezés rendszer.
* Iota és Jot: két nagyon minimalista formális rendszerrel rendelkező nyelv, úgy tervezve, hogy még egyszerűbb legyen, mint az ismertebb lambda és SKI kombinátor kalkulusok.
* Robert Atkey BCI kombinátor kalkulust használt egy programozás nyelv szemantikájának megadásához [2]
* Stephen Wolfram fizikai szimulációkhoz használt kombinátorokat [3]

## 3. 4. Előnyök, hátrányok

Előnyei:

* A kombinátor kalkulus a legegyszerűbb számítási formák közé tartozik.
* Nem használ változókat és nem kell foglalkozni a láthatósággal, névütközéssel és átnevezéssel
* Jól használható párhuzamos és elosztott tömeges adatműveletekhez, általában gyűjteményeken/tömbökön, pl.:
  + Egy függvény alkalmazása egy adathalmaz minden elemére
  + Egy adathalmaz egyetlen értékké redukálása egy asszociatív művelettel
  + Mátrix transzponálás

A párhuzamos/elosztott műveleteknél a változók használata nehézséget okozhat.

Hátrányai:

* Bár a kombinátorok általánosan alkalmazhatók, nem minden probléma megoldására alkalmasak hatékonyan. Bizonyos típusú problémákhoz más programozási módszerek lehetnek jobban illeszkedőek. Például nem használható szekvenciális számítási modellként.
* Néhány esetben a kombinátor alapú megoldások lassabbak vagy kevésbé hatékonyak az imperatív vagy objektumorientált megközelítéssel szemben. Számítógépeink memóriával és belső adattárolóval rendelkeznek, ezek a tulajdonságok a változókat használó programozási nyelveknek hasznosak.
* A kombinátor kalkulussal megfogalmazott kifejezések gyakran bonyolultak és nehezen olvashatóak lehetnek.

# 4. Módszerek

## 4. 1. Típusos SKI

Ebben a fejezetben az általam elkészített kifejezésnyelvet mutatom be, a Típusos SKI-t. Ezzel a nyelvvel főleg olyan kifejezéseket lehet megadni, amelyek az S, K és I kombinátorokból állnak, de a nyelv rendelkezik még néhány további beépített termmel is. A termek szintén kombinátorokként viselkednek és a kifejezésekben nincs lehetőség a változók használatára. A nyelv egy egyszerű típusrendszerrel is rendelkezik, ami biztosítja, hogy a kiértékelés csak helyesen típusozott utasításokra lesz alkalmazva. A Típusos SKI-ban megadott kifejezéseket először egy Parser olvassa be és elkészíti belőle a részben már típusos szintaxisfát. A szintaxisfában a típusellenőrző megpróbálja kikövetkeztetni a hiányzó típusokat, ha szükséges és elvégzi a típusellenőrzést. Végül az így elkészült jól típusozott fában a kiértékelő végrehajtja a lehetséges kombinátor alkalmazásokat és visszatér a végeredménnyel.

## 4. 2. Mély beágyazás

### 4. 2. 1. Típusok és termek

A Típusos SKI kifejezések kiértékelésekor muszáj, hogy a kifejezés minden tagjának a típusa egyértelmű legyen. A nyelv a következő típusokat támogatja:

**Nat**: A természetes számok típusa. Mivel ez a nyelv kombinátorokat használ és a benne leírt kifejezéseket gyakran egyszerűbb először lambda-kalkulussal megadni, majd átírni kombinátorokká, ezért a számok reprezentációja Church számokkal történik. [4]

A Church számokat Alonzo Church-ről nevezték el, aki először kódolt el adatokat lambda-kalkulus segítségével. Ez a módszer nagyon hasonlít a természetes számok funkcionális ábrázolásához, ahol adott egy természetes szám 0 és egy függvény, ami a paraméteréül kapott szám rákövetkezőjét adja vissza. A Church számok ennek a kiterjesztése. Minden Church szám egy függvény két paraméterrel:

Az első paraméter , az alkalmazandó ,,rákövetkezés’’ függvény. A második paraméter , az érték, ami a nullát jelöli. Így a 0 Church számokkal kifejezve:

Itt az bármilyen függvény lehet, mert nem lesz alkalmazva, a kifejezés -szel, a 0-át jelölő értékkel tér vissza. Az 1-et kódoló Church szám pedig pontosan 1-szer alkalmazza a rákövetkezési függvényt -en:

A Típusos SKI esetében a nulla értéket a ZERO term jelöli, a számok rákövetkezőjét előállító függvényt pedig a Succ term.

Például:

|  |  |  |
| --- | --- | --- |
| **Természetes számok** | **Church számok** | **Típusos SKI kifejezések** |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| n |  |  |

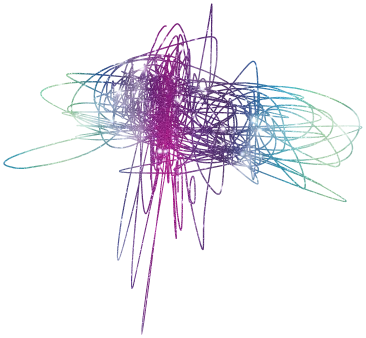
Egyszerűbb aritmetikai műveletek, mint összeadás és szorzás definiálhatóak a Church számokhoz, de azt be kell látni, hogy az eredmény értelmezéséhez mindig meg kell számolni, hogy a rákövetkezési függvény hányszor lett alkalmazva. Ez a fajta számábrázolás hatással van a teljesítményre is, mert ha hozzá szeretnénk férni egy Nat típusú értékhez, a művelet költsége helyett lesz, ahol a rákövetkezési függvény alkalmazásainak száma.

Példa az összeadásra:



* Church kódolással: Azt lehet megfigyelni, hogy a Church számok mindent a 0-hoz képest csinálnak. 1-gyel több mint , és 1-gyel több mint , ezért az 1-et jelenti -hoz képest és szintén 1-et jelent -hoz képest. Ha Church számokkal szeretnénk összeadni 3-at és 4-eat, akkor egy új Church számot kell létrehoznunk, úgy, hogy a két összeg közül az egyiknek vesszük a Church szám megfelelőjét (pl.: 3 = ) és ebben a 0 helyére beírjuk a másik összeg Church szám megfelelőjét (4 = ).

egy függvény két paraméterrel, ami -at alkalmazza a rákövetkezési függvényen, majd az eredményt alkalmazza -en, ami itt a 0-át helyettesíti.





Így tehát az M és N Church számokat összeadó függvény a következő:

* A Típusos SKI-hoz az összeadás először lambda-kalkulussal adható meg a legkönnyebben, majd átírható kombinátor kalkulusra.

Lambda kifejezés:

SKI kifejezés:

A lambda forma átírása SKI kifejezéssé úgy történik, hogy a lambda kifejezés kötött változóin elindulva jobbról balra haladva az aktuális változót alkalmazni próbáljuk a kifejezésen, viszont, ha az adott változó nem épp a kifejezés jobb szélén szerepel, akkor úgy kell módosítani a kifejezést (pl.: zárójelezéssel, kombinátorok beszúrásával), hogy a változó a megfelelő helyre kerüljön a kifejezésben.

Például az kifejezésben a változókat kell elhelyezni a kifejezésben. Először az -mel kezdünk, ami éppen csak a kifejezés jobb oldalán szerepel, vagyis, ha alkalmazni szeretnénk a kifejezésen, akkor pont a megfelelő helyen van, ezért nincs szükség a kifejezés átalakítására, egyszerűen csak el kell hagyni belőle -met. Ezután a lépés után így fog kinézni a kifejezés:

Most pedig következik. Ha a kifejezés változtatása nélkül alkalmaznánk, akkor a kifejezés végére kerülne, pedig a és tagok között szerepel a kifejezésben. Ilyenkor használható a kombinátor, ami három paramétert vár, és a harmadikat alkalmazza csak az első paraméterén.

Ha a kifejezés elejére kerül, akkor az első paramétere , a második , a harmadik pedig az éppen alkalmazandó változó lesz. Tehát a kifejezés végleges formája kombinátor kalkulusban:

A fentebb említett kombinátorok közül , és a nyelvbe beépített termek, kombinátor pedig nem, de kifejezhető az SKI kombinátorok felhasználásával.

A kombinátorral rekurziót lehet kifejezni annak érdekében, hogy az egész számokra műveleteket lehessen megadni. Definíciója:

Ha harmadik paramétere akkor -vel tér vissza, ha pedig egy nullánál nagyobb szám, , akkor alkalmazza -t -en, majd ezen alkalmazza a rekurzív kifejezést, benne eggyel kisebb számmal, -nel.

Példa a működésére:

Először a C kombinátort kell alkalmazni, aminek hatására bekerül és közé.

Mivel harmadik paramétere nem , hanem , alkalmazása után a következő kifejezést kapjuk:

alkalmazása után:

harmadik paramétere ismét nem , ezért megint a rekurzív lépést kell alkalmazni:

Most már harmadik paramétere , ezért -t fog visszaadni:

Egy másik példa a lambda kifejezés SKI-ra való átírására és a Rec kombinátor használatára, a ’’Less than or Equal” (LE), vagyis a kisebb, vagy egyenlő kifejezés.

Tervezett működése:

Tehát először az első paramétert nézzük meg, hogy ZERO-e, ha igen, akkor True-t ad vissza, a második paraméter bármilyen természetes szám lehet. Ha pedig az első paraméter nem ZERO, de a második igen, akkor False értéked ad vissza, és ha a két paraméter és , akkor -t meghívjuk -re és -re.

Ez lambda-kalkulussal kifejezve:

Átírás kombinátorokra:

**Bool**: A logikai értékek típusa. A Típusos SKI-nak mindkét logikai értékre van beépített termje, True – igaz és False – hamis, viszont nem tartalmazza a logikai aritmetikai műveleteket, de természetesen kombinátorok segítségével definiálni lehet azokat.

Az és műveleteket az -hoz hasonlóan, először lambda-kalkulussal adjuk meg, majd írjuk át SKI kalkulusra.

A kifejezésben használt kombinátor nincs beépítve a nyelvbe, de a -hez hasonlóan ezt is definiálni lehet az SKI kombinátorokkal. nem rendezi át a paraméterek sorrendjét, hanem a harmadik paramétert alkalmazza a másodikon, és az eredményt alkalmazza az első paraméteren.

ITE kombinátor egy beépített term, ami az If-Then-Else konstrukciót jelöli. Úgy működik, hogy három paramétert vár, amelyekből az elsőnek Bool típusú kifejezésnek kell lennie, a második és harmadik paraméterek tetszőleges típusúak lehetnek, de ugyan olyanoknak kell lenniük. Ha az első paraméter értéke igaz, akkor a második paraméterrel tér vissza, különben pedig a harmadikkal.

A típus nélküli SKI kalkulus logikai műveletei jóval egyszerűbben kifejezhetőek, mint a típusosé, de ezeket nem használhatjuk a Típusos SKI-ban a típusrendszer megszorításai miatt. Itt a logikai értékek reprezentációja a Church kódoláshoz kötődik:

Néhány logikai művelet a típus nélküli SKI esetében: [4]

* AND, postfix operátor:

Használata:

* OR, infix operátor:

Használata:

* NOT, postfix operátor:

Használata:

**List**: A listák típusa. A Típusos SKI-ban a listák reprezentációja hasonlít az egész számokéhoz. A különbség az, hogy a 0 elem helyett az üres lista konstruktora használható, aminek a jelölése és a rákövetkezés függvény helyett egy konstrukciós függvényt használunk a további listaelemek képzéséhez, ennek a jelölése . A Típusos SKI-ban a listák nem módosíthatóak, vagyis nem lehet elemeket ki és betenni, helyette a segítségével lehet új listát készíteni egy adott elemből és listából. Például egy egész számokat tartalmazó lista, amely az 1-es számot tartalmazza:

Egész számok listáit tartalmazó lista :

A lista típushoz kapcsolódik még egy beépített term, a RecList. A RecList-re azért van szükség, hogy rekurziót lehessen alkalmazni a listákra és ezáltal listaműveleteket lehessen megadni. Működése hasonlít a Rec működéséhez, a különbség az, hogy a RecList-et egész számok helyett listákra lehet alkalmazni.

az a paraméter, amivel akkor tér vissza RecList, ha a harmadik paramétere egy üres lista volt, különben pedig alkalmazza a fent látható rekurzív lépést.

Példa RecList használatára egy olyan kifejezés ami megadja a paraméteréül kapott listáról az elemeinek számát:

Alkalmazzuk a kifejezést egy listán, ami az 1 egész számot tartalmazza:

Mivel RecList harmadik paramétere nem egy üres lista, ezért a rekurzív lépést lehet alkalmazni:

A következő lépésben RecList harmadik paramétere már üres lista, ezért a ZERO-t adja vissza:

**Function**: A függvény típus. A legtöbb kombinátor függvény típusú, hiszen egy vagy több paramétert felhasználva készít egy új kifejezést. Például a korábban már említett kombinátor függvény típusú:

A Típusos SKI függvényei csak egy paraméteresek, ezért a több paraméteres kombinátorok típusa az úgynevezett Curry-zés módszerrel adhatók meg. A Curry-zés azt jelenti, hogy egy több argumentumot elfogadó függvényt függvények sorozatává alakítunk, amik csak egy-egy argumentumot kaphatnak. Például a kombinátor típusa , így a kifejezés típusa pedig , ami Curry-zéssel lesz. Az első paraméter alkalmazása után a függvénnyel tér vissza, amire alkalmazni lehet a második paramétert is.

Függvény típusúak a Típusos SKI alapjait adó beépített termek is, S, K és I.

S típusa:

Az S kombinátor típusa Curry formátumban kifejezve:

K és I típusai:

Néhány már korábban említett term, aminek a típusa nem triviális:

App, Lit,

---

ZERO, Succ, True, Rec, RecList, Cons, [], False, ITE, S, K, I

**Str**: A String literálok típusa. A nyelvben konstans literálokat adhatunk meg, viszont a nyelv nem tartalmaz beépített műveleteket a literálok kezelésére a ki íratáson kívül, vagyis nem tudjuk darabolni/összefűzni őket. Azért került a nyelvbe, hogy változatosabbá tegye azt és több lehetőség legyen példák bemutatására.

**Unknown**: Az ismeretlen típus, ami akkor jön létre, ha egy kifejezésben nem definiáltuk explicit módon minden kombinátor típusát vagy ha a kontextusból nem lehet kikövetkeztetni. A típusellenőrző megpróbál minden ismeretlen típust kikövetkeztetni, ha ez nem sikerül, akkor a kifejezés nem kerül kiértékelésre és a program hibával tér vissza. Például a következő kifejezésben kombinátor típusa nem lett explicit megadva és a kontextus alapján sem lehet a típusát teljesen kikövetkeztetni:

definíciója: , tehát a típusa: . az első argumentumának és visszatérési értékének a típusa, a példában . pedig a második argumentum típusa, ami a kifejezésből nem derül ki, ezért Unknown lesz. Gondolhatjuk, hogy típus kiderítése nem is szükséges, mert kombinátor a második paraméterét ignorálja, de a típusrendszer megköveteli, hogy minden típus egyértelműen meg legyen határozva. A fenti példa jól típusozott lesz, ha kap egy második paramétert is:

Vagy ha explicit határozzuk meg típusát:

Most már tudjuk, hogy második paraméterként egy Bool típusú kifejezést vár és -n az első paramétert alkalmazva egy típusú kifejezést kapunk.

ha elfogytak a típusok , és volt olyan term ami egyiknél se szerepelt akkor azok jöhetnek ide, pl.: Application

### 4. 2. 2. Termek

A termek előtti részekben , bemutatóban stb. hivatkzásokat beállítani, meg amit ambrus mondott még azt beírni, javaba ágyazott nyevlek stb, emailből…

Egyenleteket mindenhol rendezni, hogy az =-k egy helyen legyenek

Nat: a ZERO és Succ számokat is elfogad a bevitelnél

List: minden eleme u.o. típusú, Listának van type param, meg kell adni amikor pl fv típusában Lista is van

AnnotatedPretermet lehet megadni: ([1,3]):List{Nat}

Ezt meg kéne fixálni: impl hint in github ticket

# 7. Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | T. Altenkirch, A. Kaposi, A. Šinkarovs és T. Végh. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/18008/pdf/LIPIcs-FSCD-2023-24.pdf. [Hozzáférés dátuma: 21. szeptember 2023]. |
| [2] | R. Atkey, 9-12. július 2018. [Online]. Available: https://bentnib.org/quantitative-type-theory.pdf. [Hozzáférés dátuma: 21. szeptember 2023]. |
| [3] | S. Wolfram, „Stephen Wolfram: Official Website,” 7. december 2020. [Online]. Available: https://writings.stephenwolfram.com/2020/12/combinators-and-the-story-of-computation/. [Hozzáférés dátuma: 21. szeptember 2023]. |
| [4] | R. Cartwright és M. Ricken, „Department of Computer Science | Rice University,” 3. október 2005. [Online]. Available: https://www.cs.rice.edu/~javaplt/311/Readings/supplemental.pdf. [Hozzáférés dátuma: 22. szeptember 2023]. |
| [5] | „Wikipedia, the free encyclopedia,” 13. szeptember 2023. [Online]. Available: https://en.wikipedia.org/wiki/SKI\_combinator\_calculus. [Hozzáférés dátuma: 22. szeptember 2023]. |

**Notes**

számokra, churck kódolás, + wiki: Typed skinál , pl termek ismertetésénél

Elaboration, unification, típusellenőrzés, implicit paraméterek kitalálása: könyv

típuskikövetkeztetés, unification. Csörnyei Zoltán könyv

https://bookline.hu/product/home.action?\_v=Csornyei\_Zoltan\_Bevezetes\_a\_tipusrendsz&type=22&id=118024

https://en.wikipedia.org/wiki/Type\_system

Amikhez kell irodalom: deep and shallow embedding: ehhez még linkek a docxben

https://en.wikipedia.org/wiki/Domain-specific\_language

---------

Rec használatára példa lehet: pred fv.:

pred := Rec Zero K

Például pred 1 = 0, pred 30 = 29, pred 0 = 0, stb.

és még a LE fv is, ami amúgy is használva lesz később a InsertionSortban:

LE : Nat -> Nat -> Bool

-----------------------

Típusoknál: PreType, Ty ?

Rizsa:

Computing with SKI, sok hülye példával és ábrával: https://writings.stephenwolfram.com/2020/12/combinators-a-centennial-view/

Kérdések:

termek és típusoknál az S típusát hogy adjuk meg, curry formátum vagy csak egyszerűen?

listSize: RecList ZERO (B (K (K Succ)) I)

ha [] hívom meg, akkor meg kell adni RecList típus paramétereit

RecList{A}{B}, A a lista típus paramétere, B a ZERO elem típusa

Így már működik: listSize=RecList{Bool}{Nat} ZERO (B (K (K Succ)) I)

right fold function magyarítása?

Említsem a típusoknál az Str-t és Unknown-t?

Könyvből mennyit lehet idézni? (típusellenőrzésnél)

Konklúziónál:

számítások nem hatékonyak a számábrázolás miatt, O(n) művelet csak a szám hozzáféréséhez

Eredményekhez vagy összehasonlításnál vagy defekhez: azért is jók a definíciók mert azok a kombinátorok amik pl C-t v B-t használnak, pl: AND=C (B C ITE) False,

a definicíió tárolása után kiiratve nagyon hosszú kifejez lenne, ezt nehéz mindig legépelni és átlátni