|  |  |
| --- | --- |
|  | **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](#_Toc151281677)

[2. Bevezetés 2](#_Toc151281678)

[3. A kutatás háttere – SKI kombinátor kalkulus 3](#_Toc151281679)

[3. 1. Története 3](#_Toc151281680)

[3. 2. Elemei és működése 3](#_Toc151281681)

[3. 3. Irodalmi áttekintés 5](#_Toc151281682)

[3. 4. Előnyök, hátrányok 6](#_Toc151281683)

[4. Módszerek 8](#_Toc151281684)

[4. 1. Típusos SKI 8](#_Toc151281685)

[4. 2. Mély beágyazás 8](#_Toc151281686)

[4. 2. 1. Típusok és termek 8](#_Toc151281687)

[4. 2. 2. Parser 17](#_Toc151281688)

[4. 2. 3. Típusellenőrző 22](#_Toc151281689)

[4. 2. 3. 1. Kétirányú típusellenőrző 23](#_Toc151281690)

[4. 2. 3. 2. Egységesítés 27](#_Toc151281691)

[7. Irodalomjegyzék 29](#_Toc151281692)

# 2. Bevezetés

Diplomamunkámként egy gyakorlatban is használható, az SKI kombinátorokra épülő típusos kifejezésnyelvet készítettem. Az SKI kombinátor kalkulus egy olyan egyszerű funkcionális programozási nyelv, mely nem tartalmaz változókat, hanem azok szerepét kombinátorok töltik be. A kutatásom célja annak kiderítése volt, hogy lehetséges-e egy ilyen nyelvvel programozni és milyen módszerekkel lehet nagyobb programokat egyszerűbben és olvashatóan megadni. A nyelvhez parsert, típusellenőrzőt és egy kiértékelőt is implementáltam Java nyelven. Összehasonlítottam az implementáció egyszerűsége szempontjából a nyelv sekély és mély beágyazását. A mély beágyazásos változaton belül az egyszerű típuskikövetkeztetést, kétirányú típusellenőrzést és egységesítő algoritmusokat hasonlítottam össze használhatóságuk és az implementáció bonyolultsága szerint. Ezen szempontok szerint a legjobb megoldásnak a mély beágyazás, egységesítéssel ellátott típuskikövetkeztető algoritmusa bizonyult.

# 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. [1], [2]

## 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 műveletek végezhetőek el. E három alapkombinátor kombinációjával és a paramétereken való alkalmazásával, bármely kiszámítható függvény reprezentálható és kiszámítható. Ez abból adódik, hogy az SKI kombinátor logika és a lambda-kalkulus algebrailag ekvivalens egymással, vagyis minden SKI kombinátorral megadott kifejezés átírható lambda-kalkulusra és a lambda-kalkulus ekvivalens a Turing-géppel. [3]

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.

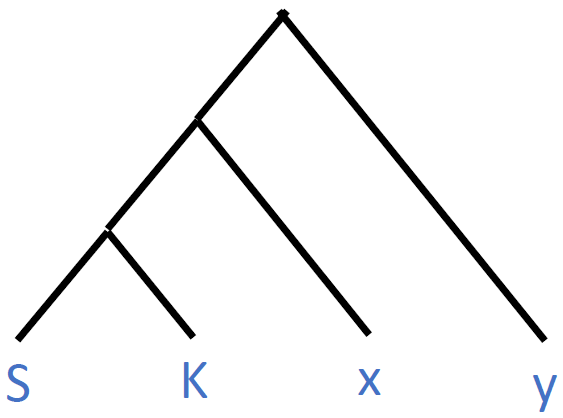
A példákban az egyes termek közötti szóközök a függvényalkalmazás műveletét jelölik.

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

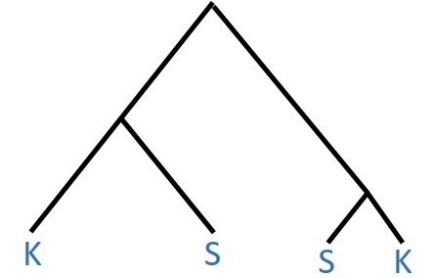
Az I kombinátor pedig az identitás függvényt reprezentálja, vagyis egyszerűen csak a kapott paraméterével tér vissza.

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.



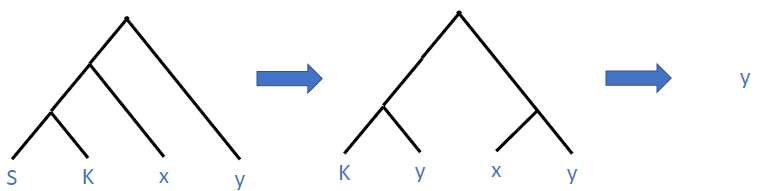
Például a helyett elég csak -t írni.

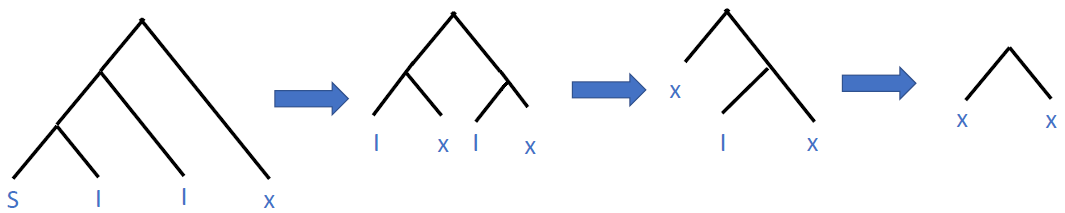


Az egyes kombinátorokon csak akkor lehet átírási szabályokat alkalmazni, ha azok a megfelelő számú paramétert kapták, S: 3 db., K: 2 db. és I: 1 db.

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:





[2]

## 3. 3. Irodalmi áttekintés

Az SKI kalkulus tulajdonságai már jóval az első számítógép megjelenése előtt ismertek voltak. [5] 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. [2]
* Az Unlambda egy majdnem tisztán funkcionális programozási nyelv. Kombinátor logikán alapszik, egy lambda operátorok és szabad változók nélküli kifejezés rendszer. [6]
* Iota és Jot: két nagyon minimalista formális rendszerrel rendelkező nyelv, úgy tervezve, hogy még egyszerűbb legyen, mint az ismertebb lambda és SKI kombinátor kalkulusok. [4]
* Robert Atkey BCI kombinátor kalkulust használt egy programozás nyelv szemantikájának megadásához [3]
* Stephen Wolfram fizikai szimulációkhoz használt kombinátorokat [4]
* Példa típusozott SKI kalkulus sekélyen beágyazott implementációjára Java nyelven [9]
* Példa SKI kombinátorok sekélyen beágyazott implementációjára C# nyelven [11]
* Kombinátorok használata funkcionális nyelvek fordítására [12]
* Online kombinátor kalkulátor és táblázat nevezetes kombinátorok lambda és SK kombinátor formáival [13]

## 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. [9]

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

A mély beágyazás esetén a Típusos SKI kifejezés nyelv a Java programozási nyelvbe lett beágyazva. Ami azt jelenti, hogy a Típusos SKI valójában egy Java alkalmazás, amely magában foglalja a Típusos SKI adattípusait, beépített termjeit, egy parsert, típusellenőrzőt, kiértékelőt és az egészet koordináló futtató keretrendszer.

### 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. [10]

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 |  |  |

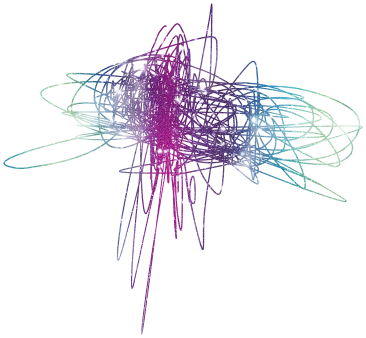
Egyszerűbb aritmetikai műveletek, mint összeadás és szorzás definiálhatóak a Church számokhoz, de azt be kell látni, hogy az eredmény értelmezéséhez mindig meg kell számolni, hogy a rákövetkezési függvény hányszor lett alkalmazva. Ez a fajta számábrázolás hatással van a teljesítményre is, mert ha hozzá 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: [11]

* 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 . Egy új listát úgy lehet elkészíteni, hogy a -nak első paraméterként egy új elemet adunk, amit el szeretnénk tárolni, másodiknak pedig egy listát, ami lehet már egy meglévő vagy akár üres is. Egy listán belül minden elemnek azonos típusúnak kell lennie. 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:

**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. A nyelv minden olyan literált amit nem ismer fel beépített termként, azt szöveges literálként fog kezelni. A következő példában, az I felismerhető kombinátornak, a HelloWorld! pedig szöveges literál lesz:

**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:

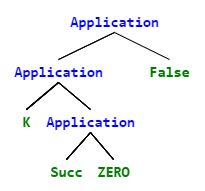
Vagy ha explicit határozzuk meg típusát:

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

**Application**: Az alkalmazás nem egy típus, hanem egy beépített term. A kifejezésekben a szóközök jelölik az alkalmazásokat. Az alkalmazás csak akkor végezhető el, ha az alkalmazás bal oldalán egy függvény típusú kifejezés áll, a jobb oldalán pedig egy olyan típusú kifejezés, ami megegyezik a bal oldali kifejezés bemeneti típusával.

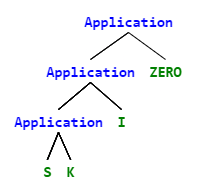
### 4. 2. 2. Parser

A Típusos SKI parserének feladatai a lexikális és szintaktikus elemzés. A parser a bemenetét karaktersorozatként kapja és a korábban már bemutatott termekhez hasonló köztes termekké, pre-termekké alakítja és eredményül egy pre-termekből álló szintaxisfát készít. A parser a pre-termek mellett, az inputban explicit módon megadott típusokat is felismer és előzetes típusokat (pre-type) hoz létre belőlük. Így a szintaxisfa néhány pre-termje ilyen előzetes típusokat is hordozhat.



A pre-termekből elkészített szintaxisfa azt jelenti, hogy a hozzá tartozó kifejezés a parser által felismerhető szimbólumokat tartalmaz és szintaktikusan helyes. Ez a szintaxisfa annyiban különbözik a később a típusellenőrző által elkészített szintaxisfától, hogy itt még nem várjuk el, hogy tartalmazzon típusokat és ha helyenként vannak is benne típusok, nem vizsgáljuk a helyességüket.

A Típusos SKI programok kódjában az egyes termeket elválasztó szóközökből a parser kombinátor alkalmazásokat készít. A szintaxisfa csúcsai az alkalmazások lesznek, a csúcshoz tartozó gyerekek pedig a szóköz melletti termek. Az alkalmazások a forráskódban balról jobbra haladva jönnek létre, vagyis, ha van 4 termünk szóközökkel elválasztva, akkor először a második termet alkalmazzuk az első termen, majd a harmadik termet alkalmazzuk az előbb elkészített alkalmazáson és így tovább…

Például:

A nyelvben lehetőség van zárójelezett kifejezések megadására is. A zárójelezést nem tartalmazó kifejezések szintaxisfái úgy néznek ki fentről lefelé, hogy az egyes alkalmazás csúcsok baloldali gyereke egy részfa és a jobb oldali pedig egy term. Ha viszont van zárójelezett kifejezés is és alkalmazva van egy korábbi kifejezésen akkor a zárójelezett kifejezés a szintaxisfában egy olyan részfa lesz, ami egy alkalmazás csúcs jobboldali gyereke. Példa erre a korábban mutatott .

A Típusos SKI-val megadott kifejezések csak akkor kerülnek kiértékelésre, ha a típusellenőrző a kifejezésben szereplő összes termnek meg tudja határozni a típusát. A típusellenőrző általában képes kikövetkeztetni a termek típusát, de a nyelvben bármely term vagy akár kifejezés megadható úgy, hogy a felhasználó által explicit jelölve legyen a típusa. Az ilyen típusokkal ellátott termek az AnnotatedPretermek.

Szintaxisa: , például: .

Néhány termnek egyértelmű a típusa. Ilyenek például a ZERO, ami csakis típusú lehet, és a Succ, ami pedig típusú. De vegyük például a kombinátort, aminek a típusa . Ha csak egy paramétert kap, akkor egy olyan alkalmazást kéne eredményül adnia, ami bármely paraméter esetén önmagát adja vissza. Viszont, ha -t nem AnnotatedPretermként írjuk fel, hanem simán csak típus nélkül és csak egy paramétert adunk neki, akkor ezt a kifejezést a típusellenőrző hibásnak jelöli és nem értékeli ki, mivel típus értéke ismeretlen.

estén típusában ismert: , viszont ismeretlen és az eredmény egy olyan alkalmazás term lenne, aminek a típusa: .

Ha viszont tudjuk, hogy milyen típusú lesz második paramétere (például Bool), akkor a kifejezés felírható AnnotatedPretermként is.

Így már kiértékelhető lesz a kifejezés és tudjuk, hogy az eredmény alkalmazás lesz, aminek a típusa: .

Típussal annotálni nem csak egyszerű termeket lehet, hanem alkalmazásokat is, vagyis egészen bonyolult kifejezéseket. Ennek módja, hogy az annotálandó kifejezést zárójelek közé tesszük, majd a kettőspont után megadjuk a típusát.

Például az kombinátor, ami két egész szám összeadására használható:

Amikor annotálni kell egy nagyobb kifejezés típusát, akkor egyszerűbb lehet bizonyos esetekben az egész kifejezést zárójelbe tenni és úgy megadni a típusát, minthogy külön kiírni a típust a kifejezés minden egyes tagjánál.

Az AnnotatedPretermek típusának megadásakor, ha a típus egy függvény típus, akkor annak a jelölése a függvény bementi és kimeneti típusai egy nyíllal elválasztva.

A típus leírásakor lehet zárójelezést is használni. A zárójelezést nem tartalmazó, de több argumentumos függvény típusokat úgy kell értelmezni, mintha jobbra lenne zárójelezve, vagyis mintha a korábban már említett Curry-zett formában lenne.

Például:

Van néhány term, aminek nem csak a típusát lehet explicit megadni, hanem típusokkal is fel lehet paraméterezni. A típusparaméter szóköz nélkül, rögtön a hozzátartozó term után kerül, termtől függően akár több is.

Szintaxisa:

Típusparamétert az S, K, I, Rec és RecList termek kaphatnak. A típusparaméterek a kombinátorok típusdefiníciójában szereplő határozatlan típusokat jelölik. Ez egy módszer a Típusos SKI használatának egyszerűsítésére, rövidebbé teszi a kifejezéseket a forráskódban. A típusparaméterek bevezetéséről később még lesz szó a Típusellenőrző fejezetben.

Termek típusainak explicit megadása AnnotatedPretem-ként összehasonlítása a típusparaméterek használatával:

Ha egy term a típusparaméteres formában van használva, akkor muszáj kiírni az összes típusparaméter zárójeleit, még akkor is, ha egyes típusparamétereket nem adunk meg, akkor azok a zárójelek csak üresen szerepelnek. A típusellenőrző megpróbálja majd kikövetkeztetni a típusparaméterekkel nem megadott típust.

Példák:

egy helyes és végrehajtható kifejezés, mert első, hiányzó típusparamétere az első paraméter típusából kikövetkeztethető, hiszen egyértelműen Bool típusú. Az eredmény egy alkalmazás lesz, ami egy Nat típusú paramétert vár és a Bool típusú -t adja vissza.

kifejezés szintaktikusan helyes, de a típusozása nem, mert második paraméterének típusa ismeretlen és ebben a formában nem is következtethető ki.

típusellenőrzése szintén hibát jelez, mert a második típusparaméter nem egyeztethető össze második paraméterének típusával.

Ha egy AnnotatedPreterm típusa lista típust is tartalmaz, a lista mellé is ki kell írni a kapcsos zárójeleket, amik közé a lista típusparamétere kerül, ez a típusparaméter jelzi a listában lévő elemek típusát.

Egy másik módszer a nyelv használatának egyszerűsítésére, a listák megadásának alternatív módja, hogy a listaelemeket vesszővel elválasztva szögletes zárójelek között szerepelnek. Például: . A beolvasás után a parser ebből a kifejezésből majd előállítja -os, kombinátoros formát:

A szögletes zárójeles szintaxis nemcsak rövidíti a kifejezéseket, hanem átláthatóbbá is teszi azokat és feltételezhetően a használata intuitív a legtöbb fejlesztő számára, mivel ez a fajta ábrázolás közelebb áll a gyakorlatban gyakran használt programozási nyelvek tömb, illetve lista ábrázolásához.

A parser ad segítséget az egész számok leírására is. Azt már láttuk, hogy a és termek használatával hogyan lehet természetes számokat definiálni, de ezen kívül egy természetes szám megadható a szokásos arab számjegyekkel is. Az arab számokkal leírt értékeket a parser alakítja át a kombinátoros formára. Ez a módszer nagyon hasznos már kisebb érékek esetén is, jelentősen le tudja rövidíteni a forráskódot és javítja az olvashatóságot. Például, ha az 5 értéket fejeznénk ki kombinátorokkal, a következő hosszú kifejezést kapjuk:

A természetes számok megjelenítésének javítására a Típusos SKI futtató keretrendszerébe épített pretty print funkció használható, amely az egyes kifejezések kiértékelése után, a kiíratáskor alakítja át kombinátoros formáról a természetes számokat arab számokká. A pretty print a listák megjelenítését is egyszerűsíti, kiíratáskor a szögletes zárójeles alakban írja ki azokat. A pretty print alapértelmezés szerint be van kapcsolva, de tetszőlegesen ki- és bekapcsolható. Kikapcsolása debug-oláskor lehet előnyős, mert nem csak a lerövidített kifejezések eredeti formáját mutatja meg, hanem hiba esetén egy rövid hibaüzeneten kívül a stacktrace-t is lehet látni. A következő példa azt mutatja, hogy a pretty print, hogyan képes jelentősen lerövidíteni egy SKI kifejezést.

Egy természetes számokat tartalmazó lista, pretty print nélkül:

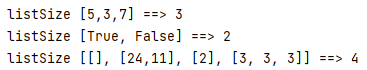
Pretty print használatával:

A Típusos SKI egy másik, talán a leghatékonyabb parserhez kapcsolódó módszere, ami a nyelv használatának egyszerűsítésére vonatkozik, a definíciók használata. A definíciókat nem szabad összetéveszteni a változókkal. A változókkal ellentétben a definíciók a kifejezések kiértékelés nélküli formáját tárolják el. A definíciók egy Map adatszerkezetbe kerülnek, azonosító – érték párok formájában, ahol az azonosító egy szöveges kifejezés, az érték pedig egy pre-term. Az azonosítóra vonatkoznak megszorítások, azonosító nem lehet egy a nyelvbe beépített term/kulcsszó sem és nem tartalmazhat úgynevezett whitespace-eket és zárójel, kettőspont és egyenlőségjel karaktereket. Egy új definíció tárolásakor, ha a definíciókat tartalmazó Map már rendel értéket az új definíció azonosítójához, a régi érték felül lesz írva az új értékkel. Az érték tetszőleges Típusos SKI kifejezés lehet. A definíció eltárolása előtt az adott kifejezést csak a parser ellenőrzi és alakítja pre-termé, ami legfeljebb a felhasználó által explicit megadott pre-type-ot tartalmaz, de a típusellenőrző már nem fogja átvizsgálni és az eltárolt érték nem lesz valós termekké alakítva, nem lesznek típusok hozzárendelve. Ez azért hasznos mert az egyes definíciók generikusan használhatók. Például a korábbi példákban már bemutatott és kombinátorok, amelyek nincsenek beépítve a Típusos SKI-ba, de gyakran szükség van rájuk.

Ahhoz, hogy ne kelljen minden alkalommal kiírni a és kombinátorokhoz tartozó teljes kifejezést amikor használni szeretnénk, el kell tárolni őket definícióként és utána már lehet rájuk hivatkozni a és termek használatával. Például a logikai kombinátor kifejezése tartalmazza mind a két kombinátort. Ha és tárolva van definícióként, akkor így adható meg:

Viszont, ha , nem lennének eltárolva, ki kéne írni a hozzájuk tartozó teljes kifejezéseket minden alkalommal, ahol használni szeretnénk őket, az így nézne ki:

A definícióként eltárolt kifejezés generikus használatára példa a listák elemszámát kiszámoló kombinátor. egyaránt használható természetes számok, logikai értékek vagy listák listáinak hosszának a megállapításához.



A definícióknál számít az eltárolásuk sorrendje. Egy definíció az értékében hivatkozhat egy másik definícióra, annak azonosítójának felhasználásával. Új definíció eltárolásakor, ha a parser megtalálja az eltárolandó definíció értékében egy már meglévő definíció azonosítóját, akkor ezt azt azonosítót lecseréli a hozzátartozó értékre az új definíció értékében. Erre jó példa a és kombinátorok. Ha definíció már tárolva van és el szeretnénk tárolni definíciót is, ami hivatkozik -re, akkor a azonosítóhoz tárolt érték: lesz. Viszont abban az esetben, ha még nincs tárolva akkor a definíció módosítás nélkül lesz elmentve. Az értékben szereplő -t a parser szöveges literálként fogja értelmezni, mert nem ismerhető fel egyik beépített vagy definícióként mentett termként se. Majd, ha ez után mentjük definícióját az már nem lesz hatással a korábban eltárolt definíciójára.

### 4. 2. 3. Típusellenőrző

A típusellenőrző feladata, hogy a parser által elkészített szintaxisfa pre-termjeit termekké alakítsa és amelyikhez tartozik pre-type, azokból tényleges típusokat készítsen, amelyikhez pedig nem tartozik pre-type, azoknak próbálja meg kikövetkeztetni a típusát és végül ellenőrizze is a típusok helyességét. A Típusos SKI-hoz két típusellenőrző készült.

#### 4. 2. 3. 1. Kétirányú típusellenőrző

[17] Az első megvalósítás az és függvényeket használja. Az , ahogy a neve is jelzi, a paraméteréül kapott pre-term típusának kikövetkeztetéséért felelős. Ha képes típust rendelni az adott pre-termhez, akkor eredményül egy olyan párral tér vissza, ami a pre-term aktuális term változatát és a hozzátartozó típust tartalmazza.

A két paramétert kap, egy pre-termet és egy típust, és azt ellenőrzi, hogy a kapott típus megfeleltethető-e a pre-term típusaként. Ha igen, a függvény visszatér a pre-term valós term formájával. Abban az esetben, ha az nem képes kikövetkeztet a pre-term típusát vagy a hibát jelez a típusellenőrzés folyamán, a teljes típusellenőrzés sikertelen lesz.

működése:

:=

A fenti pre-termek típusairól annyit tud csak megállapítani, hogy függvény típusúak és azt hogy hány paramétere, típusparamétere van, de azt nem tudja, hogy ténylegesen milyen típus lesz behelyettesítve a típusparaméterekbe. Például esetén tudjuk, hogy a típusa egy alakú függvény, de azt már nem ismerjük, hogy és milyen típusokat jelölnek.

A példákban szereplő aposztrófos kifejezések a pre-termekből készített valódi termeket jelölik.

Ha egy literál, akkor .

Alkalmazásra:

A példában szereplő \_\_ tetszőleges értéket jelöl.

AnnotatedPreterm:

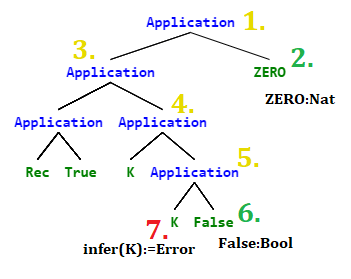
működése:

A lista típus a nyelv ezen változatába még nem volt beépítve.

A kétirányú típusellenőrző elnevezés abból származik, hogy a típusellenőrzés/következtetés során általában az hívja a -et, de az alkalmazás term esetén a is meghívja az -t. Az végzi a típus kikövetkezést, de a nyelv jelenlegi állapotában csak kevés pre-term típusát tudja teljesen meghatározni. Azoknál a pre-termeknél, amelyeknek a típusa függ a paramétereinek a típusától, mint például a kombinátor, ahol legfeljebb annyit tud, hogy egy alakú függvény, pedig paramétereinek típusával egyezik meg, de a paraméterek típusai nem ismertek még az -en belül. A nyelv itt még nem tartalmazta a pre-type-okat és az ismeretlen típust, ezért az, hogy a kombinátorhoz egy függvényt rendeljen típusként, még nem értelmezett. Ahhoz, hogy a típusellenőrzőnek megfelelő kifejezéseket lehessen írni, szinte minden termet AnnotatedPreterm formában kellett felírni. A típusok kiírása viszont már nagyon egyszerű programok megadása esetén is nagyon bonyolult és olvashatatlan kifejezéseket eredményezett. Vegyük például az kombinátort:

Ha a értéket alkalmazzuk -ra, és kiírjuk a termek típusait, úgy, hogy a típusellenőrző is elfogadja, akkor a következő kifejezést kapjuk, alkalmazásoknál tagolva új sorokba:

A következő ábrán kifejezés szintaxisfája szerepel és az látható rajta, hogy a típusellenőrzés milyen lépésekben járja be azt, akkor, ha a termek típusai nincsenek annotálva. A sárga színű sorszám azt jelenti, hogy az adott elemre már rálépett a típusellenőrző, de még nem végzett. A zöld sorszám jelentése, hogy sikerült meghatározni a típust, a piros sorszám pedig hibát jelez.

Az ábrán az látszik, hogy a 2-es és 6-os csúcsok típusait egyértelműen meg tudta határozni az algoritmus, de a 7. lépésben hibával tér vissza. Ha a kifejezésen változtatnánk és a 7. csúcsban lévő típusát annotálnánk, akkor az algoritmus egy szinttel feljebb, a 4. csúcs bal oldali gyerekénél, szintén -nál jelezne hibát. Ha ott is annotálva lesz a típus, akkor már csak típusa fog hiányozni.

Az első módszer a kifejezések hosszának rövidítésére a kombinátor típusparaméterrel való kiterjesztése: . típusa csak egy ismeretlen típust tartalmaz és általában nem nehéz kitalálni azt. A típusparaméter használata jelentősen lerövidíti a kifejezést AnnotatedPreterm változatához képest:

Nem sokkal ezután típusparaméterek lettek bevezetve további termekhez is: . Ekkor a Típusos SKI továbbra se tartalmazott Unknown típust és nem kezelte jól a hiányzó/ismeretlen típusokat, szóval a típusparaméterek használatának főleg akkor volt értelme, ha a termek egyik típusparamétere se maradt kitöltetlen. A típusparaméterek használatával az példa már így néz ki:

Így a példa kifejezés már szinte a felére rövidült, viszont az ennél összetettebb és bonyolultabb kifejezések továbbra is túl hosszúak lehetnek és sokszor a típusparaméter kitalálása se triviális.

#### 4. 2. 3. 2. Egységesítés

A kétirányú típusellenőrzésnél a hosszú kifejezéseket általában az okozta, hogy habár a kifejezés minden kombinátorának típusa egyértelműen meghatározható lett volna, de a szintaxisfa csúcsai nem osztották meg egymás között a kikövetkeztetett típusokat, vagy az, ha az azelőtt értékelt ki egy kombinátort, minthogy a hozzá tartozó paramétereket értékelte volna ki.

Az egységesítés egy olyan eljárás, amely a típusegyenletek megoldásával képes az implicit típusok meghatározására. Az algoritmust Robinson adta meg 1965-ben. [18], [19] Az ismeretlen típus az egységesítés implementálásakor került bevezetésre a Típusos SKI-ba, mivel az egységesítés típusegyenleteiben ismeretlen típusok is szerepelhetnek. Ez a módszer is használ egy függvényt, ami az inputként kapott pre-termek típusait következteti ki és egy valós termből és a hozzátartozó típusból álló párral tér vissza. Ez a változat már nem fog hibát jelezni pre-termekre. Az egységesítésnél nyilvántartunk egy globális gyűjteményt, ami az ismeretlen típusokat tárolja. Minden ismeretlen típusnak van egy egyedi azonosítója és ezzel az azonosítóval van indexelve a gyűjteményben, ennek segítségével lehet elérni őket. Az azoknál a pre-termeknél, amelyek típusában vannak nem ismert elemek, minden egyes különböző nem ismert típusra elkészít egy új ismeretlen típust és hozzáadja az ismeretlen típusok gyűjteményéhez és az eredményében szereplő típusban is ezek az ismeretlen típusok fognak szerepelni.

Például: és az ismeretlen típusok gyűjteményébe bekerül a két új elem:

hasonlóan működik pre-termek típusparaméteres változataira is, annyi ismeretlen típust készít amennyi paraméter nem lett explicit meghatározva.

Például, ha K pre-termnél csak egy ismeretlen típust kell létrehozni.

a többi pre-termmel is hasonlóan működik, az AnnotatedPreterm és alkalmazás pre-termeket kivételével. Ebben a két esetben készülhetnek típusegyenletek, amelyek megoldását az egységesítés végzi.

működése az alkalmazás pre-termekre:

Először az rekurzív módon meghívódik az alkalmazás bal és jobb oldali elemeivel. Ha mind a két hívás hiba nélkül tér vissza, akkor el lehet kezdeni a típusegyenletek létrehozását. A típusegyenlet valójában olyan típusok párja, amikkel kapcsolatban azt várjuk el, hogy megegyezzenek. Egy alkalmazás akkor helyes, ha a bal oldali kifejezésének a típusa egy olyan függvény, aminek bemeneti típusa ugyan az, mint az alkalmazás jobb oldali kifejezésének típusa, ezért létre tudunk hozni egy függvény típust, aminek a bemeneti típusa az alkalmazás jobb oldali kifejezésén végrehajtott eredményében szereplő típus lesz, a kimeneti típusra pedig létrehozunk egy új ismeretlen típust, amit hozzáadunk az ismeretlen típusok gyűjteményéhez is. Ezt a függvényt felhasználva már létre tudunk hozni egy típusegyenletet. A típusegyenlet másik típusa, az alkalmazás bal oldali kifejezésén végrehajtott által meghatározott típus lesz. A típusegyenlettel meghívjuk a egységesítő függvényt, ami a típusegyenletek megoldása során frissíti az ismeretlen típusok gyűjteményét. Végül vesszük az ismeretlen típust a típusegyenletben szereplő függvény eredmény típusából és az azonosítóját használva lekérdezzük az ismeretlen típusok gyűjteményéből a hozzá rendelt típust. Ez a típus tartozik majd az alkalmazás termhez eredményében.

Például kifejezésre:

Az alkalmazás bal és jobb pre-termjeinek kiértékelése -rel:

,

Ismeretlen típus hozzáadása az ismeretlen típusok gyűjteményéhez:

Az alkalmazás bal kifejezésének várható függvény típusának elkészítése és a benne lévő újabb ismeretlen típus hozzáadása a gyűjteményhez:

A típusegyenlet megadása és a meghívása:

Miután végzett a , ez lesz az gyűjtemény tartalma:

Ezután az ismeretlen típusát kell meghatározni.

Végül az eredménye lesz.

működése az AnnotatedPretermeknél:

Először a pre-term tagon hajtódik végre az . Ha nem történt hiba, el lehet készíteni a típusegyenletet az eredményében lévő típusból és az AnnotatedPreterm típus tagjából. A típusegyenleteket átadjuk a -nak, ami elvégzi az egységesítést. Ha az egységesítés során nem történt hiba, eredményül az AnnotatedPreterm termjét és típusát tartalmazó párral tér vissza.

Például kifejezésre:

végrehajtása -n:

Típusegyenlet elkészítése és meghívása:

Ha az egységesítés sikeresen befejeződött, eredményében az AnnotatedPreterm típusát lehet változtatás nélkül felhasználni, mivel az AnnotatedPreterm típusa nem tartalmazhat ismeretlen típust. Az eredmény: .

példán keresztül mutatni, hogy működik az unification, szintaxisfa, vagy ábrákkal

Végén leírni, hogy az Infer-check módszer kiterjesztésével a Pretermekre és az ismeretlen típusra, ugyan úgy képes típust kikövetkeztetni mint az Unify, kifejezések hossza ugyan akkora, részletezni, hogy hogy történt a kiterjesztést.   
Ha érdekes leírni az infert és checket a listákra és eddig be nem mutatott termekre

# 7. Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | A. Kaposi, „Bitbucket repository,” 7. december 2021. [Online]. Available: https://bitbucket.org/akaposi/typesystems/src/79c97d8d49d1fd04691646ceea04f03c3a8be837/src/main.pdf. [Hozzáférés dátuma: 29. szeptember 2023]. |
| [2] | A. Aiken, „Practical Combinator Languages, CS242, Lecture 3,” 2021. |
| [3] | T. Altenkirch, A. Kaposi, A. Šinkarovs és T. Végh, „Combinatory Logic and Lambda Calculus Are Equal, Algebraically. FSCD 2023: 24:1-24:19T.,” *https://drops.dagstuhl.de/opus/volltexte/2023/18008/pdf/LIPIcs-FSCD-2023-24.pdf.* |
| [4] | A. Aiken, „Combinator Calculus, CS242, Lecture 2,” 2022. |
| [5] | M. Schönfinkel, „Über die Bausteine der mathematischen Logik (PDF). Mathematische Annalen (in German). 92 (3-4): 305-316. doi:10.1007/bf01448013. S2CID 118507515. "English translation: Schönfinkel (1967)",” 1924. |
| [6] | „Wikipedia, the free encyclopedia,” 20. június 2023. [Online]. Available: https://en.wikipedia.org/wiki/Unlambda. [Hozzáférés dátuma: 30. szeptember 2023]. |
| [7] | „Wikipedia, the free encyclopedia,” 21. augusztus 2023. [Online]. Available: https://en.wikipedia.org/wiki/Iota\_and\_Jot. [Hozzáférés dátuma: 30. szeptember 2023]. |
| [8] | R. Atkey, 9-12. július 2018. [Online]. Available: https://bentnib.org/quantitative-type-theory.pdf. [Hozzáférés dátuma: 21. szeptember 2023]. |
| [9] | 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]. |
| [10] | „Pastebin,” 13. június 2012. [Online]. Available: https://pastebin.com/zz19xx8n. [Hozzáférés dátuma: 30. szeptember 2023]. |
| [11] | „Dixin's Blog,” Microsoft, 21. november 2018. [Online]. Available: https://weblogs.asp.net/dixin/lambda-calculus-via-c-sharp-21-ski-combinator-calculus. [Hozzáférés dátuma: 30. szeptember 2023]. |
| [12] | P. Hudak és D. A. Kranz, „A Combinator-Based Compiler for a Functional Language. POPL 1984: 122-132”. |
| [13] | C. Rathman, „Chris Rathman Home Page,” [Online]. Available: https://www.angelfire.com/tx4/cus/combinator/birds.html. [Hozzáférés dátuma: 30. szeptember 2023]. |
| [14] | A. Aiken, „Combinators II., CS242, Lecture 3,” 2022. |
| [15] | 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]. |
| [16] | „Wikipedia, the free encyclopedia,” 13. szeptember 2023. [Online]. Available: https://en.wikipedia.org/wiki/SKI\_combinator\_calculus. [Hozzáférés dátuma: 22. szeptember 2023]. |
| [17] | T. Coquand, „An Algorithm for Type-Checking Dependent Types. Sci. Comput. Program. 26(1-3): 167-177,” 1996. |
| [18] | Z. Csörnyei, Bevezetés a típusrendszerek elméletébe, Budapest: ELTE Eötvös Kiadó, 2012. |
| [19] | R. J. Alan, „A machine oriented logic based on the resolution principle. Journal of the Association for Computing Machinery, 12. évf. (1965) 1. sz., 23-41. p.”. |

**Notes**

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

Sekély implementációnak imputként már a parser és typechecker által elkészített, jól típusozott szintaxisfa lesz. Evaluator helyett a függvényt használjuk. Ha parser, typechecker nélkül szeretnék kifejezéseket megadni a sekélyben, akkor mivel tisztán java függvényeket hívunk, csakis típushelyes kifejezéseket lehet megadni, mert a Java kikényszeríti.

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

---

Typechecker- kikövetkeztetéshez: példa arra, hogy mikor kell mégis kiírni a típust

listSize-t 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)

Vagy, ha nem paraméterezem fel a RecList-et, de ki kell írni a lista típusát ha üres, annotatedPretermként: listSize []:List{Nat}, de ezt kipróbálni még List{}-vel esetleg

Ha még nem volt, akkor a program példákhoz (+ Fact számítás és InsertionSort)

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

Lista elemek összegzése

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

---

Rizsa:

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

Kérdések:

a gyakorlati példáknál csak a sekély beágyazásra találtam példát

github open ticket lambda convertálás algoritmus

Iter vs Rec