|  |  |
| --- | --- |
|  | **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](#_Toc153469963)

[2. Bevezetés 2](#_Toc153469964)

[3. A kutatás háttere – SKI kombinátor kalkulus 3](#_Toc153469965)

[3. 1. Története 3](#_Toc153469966)

[3. 2. Elemei és működése 3](#_Toc153469967)

[3. 3. Irodalmi áttekintés 5](#_Toc153469968)

[3. 4. Előnyök, hátrányok 6](#_Toc153469969)

[4. Módszerek 8](#_Toc153469970)

[4. 1. Típusos SKI 8](#_Toc153469971)

[4. 2. Mély beágyazás 8](#_Toc153469972)

[4. 2. 1. Típusok és termek 8](#_Toc153469973)

[4. 2. 2. Parser 17](#_Toc153469974)

[4. 2. 3. Típusellenőrző 22](#_Toc153469975)

[4. 2. 3. 1. Kétirányú típusellenőrző 23](#_Toc153469976)

[4. 2. 3. 2. Egységesítés 27](#_Toc153469977)

[4. 2. 3. 3. A kétirányú típusellenőrző kiterjesztése ismeretlen típusra 33](#_Toc153469978)

[4. 2. 4. Kiértékelő 35](#_Toc153469979)

[4. 3. Sekély beágyazás 36](#_Toc153469980)

[5. Eredmények 38](#_Toc153469981)

[5. 1. Mély és sekély beágyazás összehasonlítása 38](#_Toc153469982)

[5. 2. Mély beágyazás típuskiköveztető módszereinek összehasonlítása 41](#_Toc153469983)

[5. 3. Esettanulmány 41](#_Toc153469984)

[7. Irodalomjegyzék 43](#_Toc153469985)

# 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.

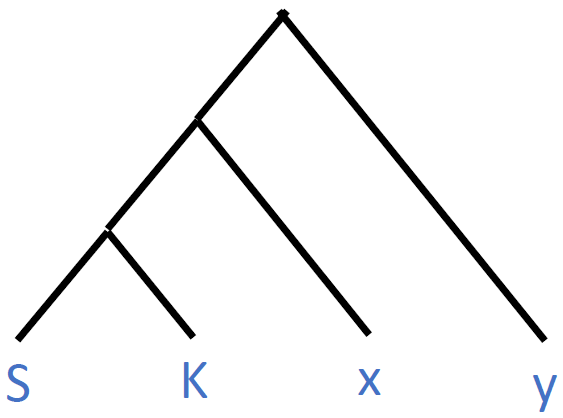
A példákban az egyes termek közötti szóközök a függvényalkalmazás műveletét jelölik.

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

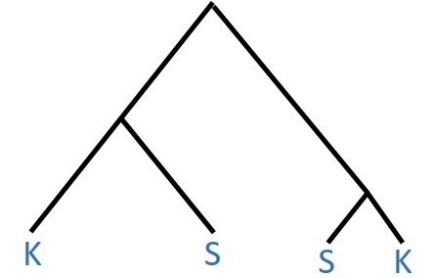
Az I kombinátor pedig az identitás függvényt reprezentálja, vagyis egyszerűen csak a kapott paraméterével tér vissza.

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.



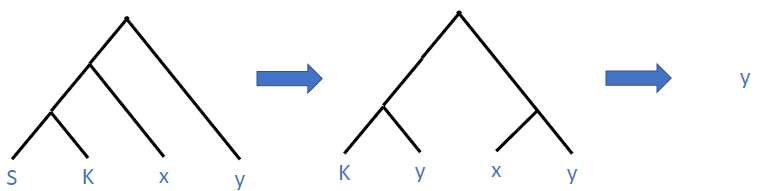
Például a helyett elég csak -t írni.

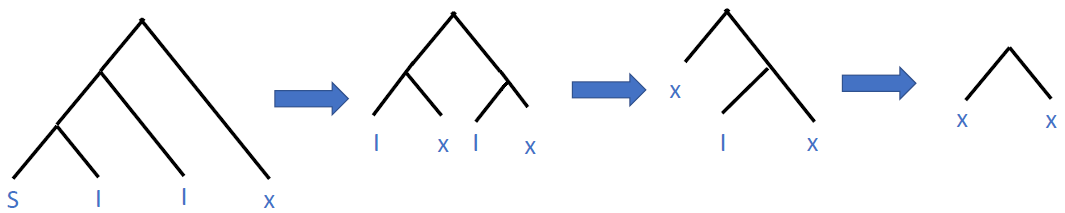


Az egyes kombinátorokon csak akkor lehet átírási szabályokat alkalmazni, ha azok a megfelelő számú paramétert kapták, S: 3 db., K: 2 db. és I: 1 db.

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 kifejezésnyelv 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, ezek mind Java osztályokként lettek megvalósítva. Ez azért nevezhető mély beágyazásnak, mert parsolás, típusellenőrzés és kiértékelés folyamatok nem a Java szintjén történnek, hanem eggyel mélyebben, az azzal megvalósított komponensek segítségével.

### 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 |  |  |

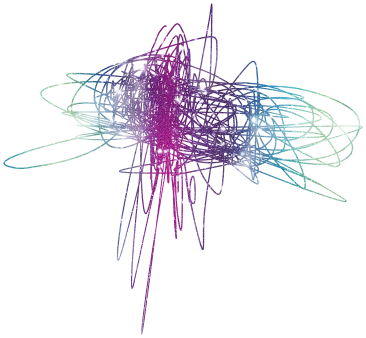
Egyszerűbb aritmetikai műveletek, mint összeadás és szorzás definiálhatóak a Church számokhoz, de azt be kell látni, hogy az eredmény értelmezéséhez mindig meg kell számolni, hogy a rákövetkezési függvény hányszor lett alkalmazva. Ez a fajta számábrázolás hatással van a teljesítményre is, mert ha hozzá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

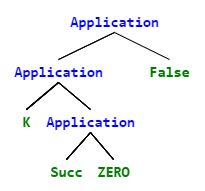
Vagy ha explicit határozzuk meg típusát:

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.

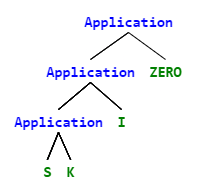
### 4. 2. 2. Parser

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:

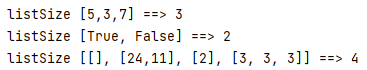
Pretty print használatával:

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:

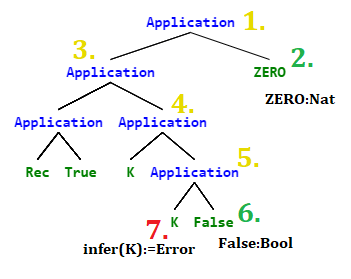
működése:

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:

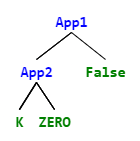
Ha az egységesítés sikeresen befejeződött, eredményében az AnnotatedPreterm típusát lehet változtatás nélkül felhasználni, mivel az AnnotatedPreterm típusa nem tartalmazhat ismeretlen típust. Az eredmény: .

:

A feladata, hogy megoldja a paraméteréül kapott típusegyenleteket és közben frissíti is az ismeretlen típusok gyűjteményét a feloldott típusokkal. Az egységesítés akkor fejeződik be, ha a típusegyenletek listája üres vagy ha az egységesítés hibás állapotba kerül. Ha az egyenletek listája tartalmaz elemeket, az algoritmus minden iterációban kivesz egy egyenletet a listából, ami feldolgozásra kerül, annak alapján, hogy milyen típusok állnak a típusegyenlet két oldalán.

* Az egyik eset, ha a típusegyenletnek legalább az egyik oldalán ismeretlen típus áll. Először elnevezzük az egyenlet oldalait. Azt az oldalt, ahol az ismeretlen típus áll ismeretlennek, a másik oldalt pedig típusos oldalnak nevezzük. Az algoritmus hibás állapotba kerül, ha a típusos oldal típusa bármilyen módon tartalmaz olyan ismeretlen típust, aminek az azonosítója megegyezik az egyenlet ismeretlen típusos oldalán álló típus azonosítójával. Abban az esetben, ha mindkét oldalon ismeretlen típus volt, a típusos oldal is ismeretlen típust fog tartalmazni. Az ismeretlen típusok gyűjteménye úgy frissül, hogy az egyenlet ismeretlen oldalán lévő ismeretlenjének egyedi azonosítójához tartozó érték a típusos oldalon álló típusra módosul és a többi bejegyzés értékében is, ha megtalálható az adott ismeretlen típus, azt az aktuális típusra cserélődik. Hasonló módon a típusegyenletek listájában is, az ismeretlen típus minden előfordulása a többi egyenletben kicserélődik a típusos oldalon álló típusra. Végül a meghívódik a maradék típusegyenletekre is.
* Egy másik amikor az egyenlet mindként oldalán függvény típus áll. Ekkor két új típusegyenletet hozunk létre. Az egyik a két függvény bemeneti típusaiból, a másikat pedig a kimeneti típusaiból. A két új típusegyenlet bekerül a típusegyenletek listájába, majd újra hívjuk a -t.
* Ha az egyenlet mindkét oldalán lista típus áll, akkor a típusparamétereikből készül egy új egyenlet, ami hozzáadódik a típusegyenletek listájához. A most is meghívódik.
* Végül, ha az egyenletre egyik korábbi esetet se lehetett alkalmazni, akkor egyszerűen csak az egyenlet két oldalán álló típus egyenlőségét vizsgáljuk. Abban az esetben, ha a típusok megegyeznek, meghívjuk a -t a maradék típusegyenletekre is, különben pedig az algoritmus hibás állapotba kerül.

Példa a működésére:

Legyen a kifejezés.

A típusellenőrzés az csúcsban kezdődik és a bal gyereken,  alkalmazáson meghívódik . Ezen a szinten is kiértékelődik az mindkét gyerekre:

Ezekből lehet elkészíteni a típusegyenletet. Az egyik típus típusa lesz:

A másik típussal kapcsolatban pedig azt várjuk el, hogy alakú legyen.

Ekkor az ismeretlen típusok gyűjteménye így néz ki:

Az elkészült típusegyenlettel meghívjuk .

A típusegyenletek listájából kivesszük az egyetlen elemet és mivel az egyenlet mindkét oldalán függvény típusok állnak, a bemeneti és kimeneti típusaikból újabb egyenleteket készítünk és hozzáadjuk a típusegyenletek listájához.

A újabb iterációba lép és kivesz egy típusegyenletet a listából:

Ebben az esetben a bal oldalt nevezzük ismeretlen típus oldalnak és a jobb oldal pedig a típusos. Az ismeretlen típus azonosítója: 0, ezt az azonosítót használjuk az ismeretlen típusok gyűjteményében, hogy 0-hoz új elemet rendeljünk. A módosítás után így fog kinézni az ismeretlen típusok gyűjteménye:

A többi ismeretlen típus nem tartalmazza a 0 azonosítójú ismeretlen típust, ezért azok nem változnak meg. Viszont a típusegyenletek listája tartalmaz olyan egyenletet, ami hivatkozik -ra, ezt is át kell írni -ra. A módosítás után a típusegyenletek listája lesz.

A következő iterációjában megint van egy megoldatlan típusegyenlet.

Ebben az esetben a jobb oldalon lesz az ismeretlen típus és így változik az ismeretlen típusok gyűjteménye:

Ezután a típusegyenletek listája már üres lesz és a terminál.

Az szintjén a korábban már az által meghatározott típusokat most frissítjük az ismeretlen típusok gyűjteménye szerint:

Mielőtt visszatérne szintről egy típusozott applikáció termmel, még a hozzátartozó típust is frissíteni kell az ismeretlen típusok gyűjteményével:

Amikor visszatértünk szintre, -t végrehajtjuk jobb gyerekén is:

Most újra fel kell írni egy típusegyenletet. Az egyik típus típusa lesz, a másik pedig

Az ismeretlen típusok gyűjteményét bővíteni kell az új ismeretlen típussal:

Az új típusegyenlettel meghívjuk -t.

A típusegyenlet mindkét oldalán függvénytípusok szerepelnek, ezért elkészítjük belőlük a két új típusegyenletet és hozzáadjuk a listához.

A következő iterációban kiválasztjuk az egyik egyenletet:

Beállítjuk -t az ismeretlen típusok gyűjteményében az 1-es azonosítóhoz és mivel a 2-es azonosítóhoz tartozó érték is hivatkozik -re, ezért azt is frissíteni kell.

A következő iterációban kiolvassuk az utolsó típusegyenletet is:

Frissítjük az ismeretlen típusok gyűjteményét:

A típusegyenletek listája kiürült, ezzel termál és visszatérünk -be az szintre. Frissítjük gyerekeinek a típusait az ismeretlen típusok gyűjteménye alapján:

Végül típusát is frissíteni kell és visszatér a jól típusozott termmel.

Amikor a típusellenőrző elért a szintaxisfa gyökeréig és az ismeretlen típusok gyűjteményében még mindig maradt feloldatlan ismeretlen típus, a típusellenőrző hibával fog visszatérni. Ha viszont a szintaxisfa jól típusozott és az összes ismeretlen típust sikerült kikövetkeztetni, a szintaxisfa átadható a kiértékelőnek.

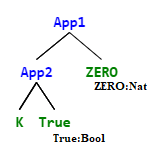
Az egységesítés használatával már szinte a legtöbb esetben nincs szükség a típusok explicit kiírására. Az kifejezés már ilyen formában is felírhatóvá válik:

#### 4. 2. 3. 3. A kétirányú típusellenőrző kiterjesztése ismeretlen típusra

Az egységesítés implementálásával bevezetésre került az ismeretlen típus. Az ismeretlen típus használatának az előnye az, hogy azokhoz a termekhez, amikhez az nem tudott típust rendelni vagy függvény típus esetén nem a teljes típust tudta kikövetkeztetni, ott az ismeretlen típust használta, majd amikor az algoritmus a szintaxisfa másik ágaiban kikövetkeztette az adott ismeretlen típust, képes volt visszajuttatni ezt az információt a hozzá tartozó termhez. Az ismeretlen típus használatával a kétirányú típusellenőrző is képes lesz erre.

A kétirányú típusellenőrzőhöz tartozó a kiterjesztés után nagyon hasonlított az egységesítés -jéhez, az alkalmazás pre-term eset kivételével. Már nem került hibás állapotba, ha egy pre-term típusparamétereit nem sikerült hiánytalanul kikövetkeztetnie (pl.: K pre-term típusparaméterek nélkül), hanem a egy olyan típust rendelt hozzá, amiben a hiányzó típust az ismeretlen típussal helyettesítette.

Ez a megoldás nem használja az ismeretlen típusok gyűjteményét és az ismeretlen típusok nem lesznek egyedi azonosítókkal ellátva. A legjelentősebb változtatás az -ben az alkalmazás pre-term esetében történt. Először az rekurzív módon meghívódik az alkalmazás jobb oldali gyerekén. Az eredmény egy term – típus páros lesz. Ha az alkalmazás bal oldali gyereke olyan pre-term, aminek egy vagy több típusparamétere van és közülük az első ismeretlen típusú, akkor fel lehet használni a jobb oldali term típusát, hogy felülírjuk az ismeretlen típust. Ezután a bal oldali gyereken is el kell végezni az -t. Az eredmény itt is egy term – típus pár lesz és a típustól azt várjuk el, hogy egy függvény legyen, aminek a bemeneti típusa ugyan az, mint az alkalmazás jobb oldali gyerekének a típusa. Ha a bal gyerek típusa ismeretlen, vagy ha függvény, de a meneti típusa nem ismert, akkor megint fel lehet használni a jobb gyerek típusát és behelyettesíteni az alkalmazás bal gyerekének típusába. Ezután az alkalmazás visszatér az eredmény típussal és a szintaxisfa következő szintjén hasonló módon folytatódik a típusok kikövetkeztetése.

Példa:

az szinten először kiértékeli a jobb oldali pre-termet, aminek az eredménye lesz, majd tovább lép szintre. Itt a jobb oldali gyerek eredménye , a bal oldali pre-term pedig típusparaméterek nélkül. Ebben az esetben első ismeretlen típusparaméterét felül lehet írni -lal: . Ezután -ra is ki kell értékelni az -t, aminek az eredménye lesz. alkalmazás bal oldali gyereke függvény típusú, aminek a bemeneti típusa nem ismeretlen és megegyezik a jobb oldali gyerek típusával, vagyis az alkalmazás helyes és visszatérhetünk szintre. eredménye: . Mivel szinten tudjuk, hogy a jobb oldali gyerek típusa , a bal gyerek típusa pedig függvény, de a függvény bemeneti típusa nem ismert, ezért azt behelyettesítjük a jobb oldali gyerek típusával. új típusa lesz és így már alkalmazás is elvégezhetővé válik.

Ezzel a kiterjesztéssel a kétirányú típusellenőrző használatakor sem szükséges kiírni a termek típusait, mert a típusellenőrző képes lesz kikövetkeztetni azokat, ha a kombinátorok megfelelő típusú paramétereket kaptak.

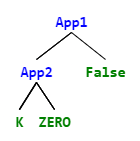
A korábban már példaként használt kifejezés ilyen módon is leírható és a típusellenőrző által kiértékelhető:

Van olyan eset, ahol továbbra is ki kell írni a típust.

A listSize a paraméteréül adott lista méretét adja vissza. A paraméterben szereplő lista típusát akkor szükséges kiírni, ha a lista üres, mert a típusellenőrző a lista elemeinek a típusából határozza meg a lista típusparaméterét, ha az nem lett explicit megadva. Üres lista paraméter esetén több módon is megadható a lista típusa. A hosszabb megoldás, ha teljesen kiírjuk listSize definícióját és megadjuk RecList típusparamétereit, ahol az első típusparaméter jelöli a lista típusparaméterét:

A rövidebb megoldás pedig, ha a lista paramétert típussal annotáljuk:

### 4. 2. 4. Kiértékelő

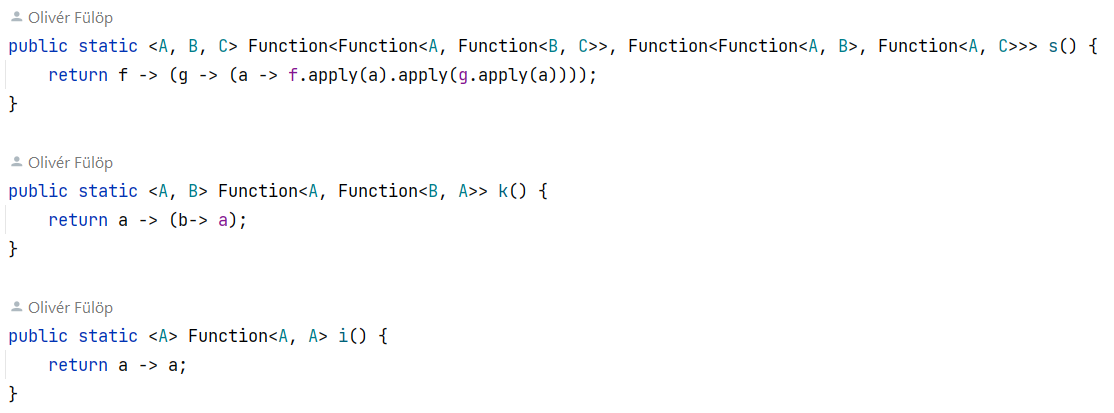
A kiértékelő egy jól típusozott szintaxisfát kap bemenetéül, amit a korábbi lépésekben a parser és a típusellenőrző építettek fel. A feladata az, hogy végrehajtsa a kifejezésben szereplő összes kombinátor alkalmazást és eredményül egyetlen termmel térjen vissza. Ha a kiértékelő bemenete olyan term, ami nem tartalmaz alkalmazást, akkor változtatás nélkül adja vissza azt. Ha viszont a szintaxisfa alkalmazást tartalmaz, a kiértékelés a fa gyökeréből indul és a kiértékelés rekurzív módon megtörténik előbb a bal, majd a jobb oldali gyerek csúcson is és a gyökérben lévő alkalmazást ezeknek az eredményein hajtja végre.

Példa: Az ábrán szereplő szintaxisfa kiértékelése az csúcsban kezdődik és a kiértékelést meghívja csúcsra, ami szintén egy alkalmazás. csúcsból kiértékeljük -t és -t. -t elvégezzük, de mivel -nak két paraméterre van szüksége, de csak egyet kapott, ezért azt még nem lehet végrehajtani és így az alkalmazás eredménye is egy alkalmazás lesz, , aminek a típusát a szintaxisfából tudhatjuk, függvény. Az eredmény visszatér , szintre, és kiértékelődik a jobb oldali gyerek is, ami a term. kiértékelése most már elvégezhető, termet alkalmazza a alkalmazásra, így a két paraméterével végrehajthatóvá válik és az eredménye a term lesz. Az SKI kombinátor kalkulussal megadott kifejezések kiértékelése balról jobbra történik, ez okozza azt, hogy a kiértékelő is a bal oldali csúcs kiértékelésével kezd és az algoritmus lemegyek egészen a szintaxisfa legalsó, bal oldali levelébe és az eredmények onnan kerülnek vissza a fentebbi szintekre, majd amikor egy adott szinten, ahol az alkalmazás rendelkezik a bal oldalon álló kombinátor összes szükséges paraméterével, akkor az alkalmazás kiértékeli a kombinátort az argumentumaival.

## 4. 3. Sekély beágyazás

A Típusos SKI sekélyen beágyazott megvalósítása a Java programozási nyelv beépített Function funkcionális interfészét használja a kombinátorok kifejezésére. Minden egyes kombinátor a funkcionális interfész egy külön megvalósítása. A Function egy egyparaméteres függvényt reprezentál, amelynek két generikus típusparamétere van, egy a bemeneti típusnak, egy másik pedig az eredmény típusának. Rendelkezik még egy absztrakt (alkalmazás) függvénnyel, amit definiálni kell a Function megvalósításakor. Az definiálásával lehet megadni, hogy az adott kombinátor hogyan viselkedjen, amikor alkalmazzák az argumentumain. A Típusos SKI típusait a sekély beágyazás esetén nem kell külön implementálni, hanem a Java saját beépített típusait lehet használni.

Az S, K és I kombinátorok sekély beágyazott megvalósításai:



Ahhoz, hogy a kombinátorokat a Function segítségével lehessen kifejezni, a kombinátorok típusait át kell írni Curry formátumba, mivel a Function csak két típusparaméterrel rendelkezik.

Példa S típusának átírására Curry formára:

Az előbbi ábrán szereplő megvalósításában szereplő paraméterek típusai rendre a típusparaméterből kiolvasható típusok lesznek:

A sekély beágyazás implementálásakor a kombinátorok definiálásán kívül elég volt csak egy parsert elkészíteni. Ez a parser ugyanazt a szintaxist ismeri fel, mint a mély beágyazott megvalósításnál, de a parsolás során nem jönnek létre pre-termek és nem épül fel a komplett szintaxisfa, hanem az egyes kombinátorokat kifejező funkcionális interfészekre hivatkozik és rögtön elvégzi a kombinátorok közötti alkalmazásokat, végül a kiértékelt eredménnyel tér vissza. Ez miatt a sekély beágyazott megvalósításhoz nem volt szükség külön típusellenőrző és kiértékelő készítésére. A típusellenőrzést és kikövetkeztetést egy az egyben a Java fogja elvégezni az paramétereinek alapján, a kiértékelés pedig automatikusan megtörténik az végrehajtásával. Ugyanúgy, mint az egységesítés típusellenőrző használatlánál, itt sem kell explicit kiírni a kombinátorok típusait. A mély beágyazásnál használt példa kifejezés a sekély megvalósítás esetén is felírható a legrövidebb formájában:

# 5. Eredmények

## 5. 1. Mély és sekély beágyazás összehasonlítása

A Típusos SKI mély és sekély beágyazott megvalósításait három szempontból hasonlítom össze: megvalósíthatóság, használhatóság és hatékonyság. Az összehasonlításban a mély beágyazásnál az egységesítést használó típusellenőrzős változat szerepel.

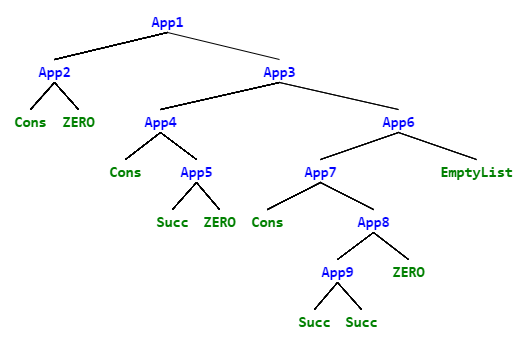
A mély beágyazott implementáció elkészítése során a Java nyelv csak a domain nyelv volt, amibe a Típusos SKI be lett ágyazva, de a tényleges megvalósítás és az azzal megadott programok futtatása egy másik absztrakciós szinten történt, ezért a nyelv minden elemét külön kellett elkészíteni: típusokat, termeket, parsert, kiértékelőt és a legtöbb munkát igénylő komponenseket, a típusellenőrzőt és a kombinátor alkalmazásokat is. Az alkalmazások logikája határozza meg, hogy mi történjen amikor a kiértékelő végrehajtja azt, attól függően, hogy melyik kombinátor áll az alkalmazás bal oldalán. Ezzel szemben a sekély beágyazás elkészítése sokkal egyszerűbb. A sekélyhez tartozó parser hasonló bonyolultságú, mint a mély beágyazásé, de a parseren kívül már csak a kombinátorokat kifejező funkcionális interfészeket kellett létrehozni. Az egyes kombinátorok implementációja csak néhány kódsor, ami jelentősen kevesebb mint a mély megvalósítás terjedelme, ahol a típusellenőrző, termek és típusok kódja több száz sorból áll.

Használhatóság vizsgálatakor azt a szempontot vettem figyelembe, hogy melyik esetben lehet egyszerűbben, rövidebben és olvashatóbban programokat megadni a Típusos SKI nyelvben. A legtöbb rövidítést és egyszerűsítést okozó megoldást a parser tartalmazza. (pl.: listák megadása szögletes zárójelekkel, természetes számok felismerése számjegyekből, definíciók használata) A két beágyazás ebben nem különbözik, mind a két parser ugyanazokra az egyszerűsítési módszerekre képes. A kifejezések hosszát még úgy lehetett rövidíteni, ha a kombinátorok típusa helyett csak a típusparamétereit kellett megadni, vagy ami még jobb, ha egyáltalán nem kellett kiírni a típust. Ezért a mély beágyazásnál nagy szerepe volt annak, hogy milyen típusellenőrző és kikövetkeztető implementációt használunk. A sekély beágyazásnál ez automatikusan működött a Java típusrendszerének köszönhetően. Helyesen paraméterezett kombinátor kifejezéseknél mind a két beágyaz megvalósítása képes ugyanazokat a kifejezéseket elfogadni, amelyekben nincsenek explicit kiírva a kombinátorok típusai.

A hatékonyság összehasonlításánál a főbb szempontok a kifejezések kiértékelésének gyorsasága és a különböző beágyazási módszerek limitációi voltak. Egy kifejezés teljes kiértékelésének ideje a parsolásból, típusellenőrzésből és a végrehajtásból tevődik össze. Mindkét megvalósítás esetén a parsolási idő azonos. A mély beágyazást a típusellenőrzés lépés észrevehetően lassítja, de az azt követő kiértékelés lépés már ugyan olyan gyors, mint a sekély beágyazásnál. Összegezve, a kifejezések végrehajtása a sekély beágyazással gyorsabb, mert a típusellenőrzés lépést a kiértékeléssel együtt végzi el, elhanyagolható időn belül.

A hatékonyságot mindkét megvalósítás esetén limitálja a nyelv Java-ba való beágyazása. A leggyakrabban előforduló probléma, hogy a program bizonyos esetekben meghaladja a JVM veremmemóriájának (Stack) kapacitását, StackOverflowError. Ennek az az oka, hogy a kombinátorok használatával az algoritmusok gyakran mély rekurzióba kerülnek és minden egyes metódus hívás egy új veremkeretet (Stack Frame) hoz létre a veremben. A különböző beágyazású Típusos SKI megvalósítások különböző esetekben kerülnek hibás állapotba. A mély beágyazás egyik limitációja a természetes számok ábrázolása. Körülbelül az 5000 értékig képes számokat feldolgozni. Nagyobb értékek parsolása és típusellenőrzése is sikeresen elvégezhető, a StackOverFlowError a kiértékelés/kiíratás közben jelentkezik, mert a hozzáférés kombinátorokkal megadott számokhoz, az adott érték nagyságával egyenesen arányos mélységű rekurzív hívás. A sekély beágyazás természetes szám értékei a Java nyelv beépített Integer típusával vannak eltárolva, ezért képes elfogadni az Integer típus maximális értékét . Egy másik példa a kisebb-egyenlő kombinátor, ami két rekurzió segítségével dolgozza fel a paraméterül kapott két természetes számot:

Itt a második paraméter növelésével nagyon gyorsan növekszik a kifejezés végrehajtásának ideje. Például: az kifejezés szinte rögtön kiértékelődik, de az kifejezés már a sekély kiértékeléssel körülbelül 5 másodpercig tartott, a méllyel pedig körülbelül 50. Viszont, ha megcseréljük a paraméterek sorrendjét, az és az kifejezések mindkét beágyazás esetén rögtön kiértékelődnek.

A sekély beágyazás limitációja pedig azok az algoritmusok, amelyek listákkal dolgoznak. Például a 0, 1, 2 természetes számokat tartalmazó lista kifejezésnek a szintaxisfája a következő:

Már ennél az egyszerű példánál is a szintaxisfa meglehetősen terebélyes és a lista teljes kiértékeléséhez a rekurzív hívásokon keresztül el kell jutni a fa legmélyére és a legbelső alkalmazások feloldását kell elvégezni. A sekély beágyazás megvalósításnál a kombinátor alkalmazások a funkcionális interfész Function megvalósításának az metódusának hívásai. Minden egyes hívás egy új veremkeretet ad a veremmemóriához és a sok függőben lévő alkalmazás miatt kialakul a StackOverFlowError. A mély megvalósításnál az egyes kombinátor alkalmazások az Application osztály objektumai, amelyek a verem helyett a heap-en jönnek létre és nem foglalja a helyet a veremmemórián. A limitáció például az kombinátor használatakor figyelhető meg. Az kombinátor az első paraméteréül kapott természetes számot szúrja be a második paraméter, rendezett lista megfelelő pozíciójába. A sekély beágyazás képes egy elemet elhelyezni egy üres listában:

Viszont, ha már legalább egy elemet tartalmaz a lista, a beszúrás sikertelen. Ezzel szemben a mély beágyazás jól működik sokkal hosszabb listákkal is.

## 5. 2. Mély beágyazás típuskiköveztető módszereinek összehasonlítása

A mély beágyazás típuskikövetkeztető módszereit ugyanazon szempontok szerint hasonlítom össze, mint a mély és a sekély beágyazott megvalósításokat: megvalósíthatóság, használhatóság és hatékonyság.

Az első típusellenőrző/kikövetkeztető módszer a kétirányú típusellenőrző volt, amihez az és függvényeket kellett elkészíteni. A másik típusellenőrző módszer pedig az egységesítés volt, ami az és a függvényeket használta. Az elkészítése mindkét esetben hasonló bonyolultságú volt, de a -et és -t összehasonlítva viszont a elkészítése tűnt egyszerűbbnek és még terjedelmében is rövidebb volt. Továbbá, ha a nyelvet kibővítenék még több olyan beépített termmel aminek a típusa már létezik, mint például a kombinátor, típusa függvény: , akkor az egységesítésnél a -t nem kéne módosítani, viszont a kétirányú típusellenőrző -jét ki kéne bővíteni az új esettel. Az ismeretlen típus bevezetése tovább bonyolította a kétirányú típusellenőrzőt, így összeségében az egységesítés megvalósítását találom egyszerűbbnek.

A kétirányú típusellenőrző első változata a legtöbb esetben nem volt képes kikövetkeztetni a termek típusait a paraméterekből és típusannotációval vagy típusparaméterekkel kellett megadni azokat explicit. Bonyolultabb kifejezéseknél, ez jelentősen megnövelte a kifejezés hosszát. Az ismeretlen típusokkal való kibővítés után már képes lett a helyesen paraméterezett kifejezésekben a típuskikövetkeztetésre csak a paraméterek alapján, ugyan úgy, mint ez egységesítés is. Tehát a kétirányú típusellenőrző és az egységesítés azonos kifejezéseket képes kiértékelni, így használhatóság szempontból megegyeznek.

A hatékonyságot tekintve a két megoldás nagyon hasonló, de a kétirányú típusellenőrző előnye, hogy a természetes számokat gyorsabban értékeli ki és nagyobb értékeket képes kezelni, mint az egységesítés. Körülbelül 8000-9000, míg az egységesítéssel csak 5000-6000.

## 5. 3. Esettanulmány

Ez az alfejezet példaként a beszúró rendezés algoritmus Típusos SKI implementációját mutatja be. Az algoritmus két rekurzív függvényből áll, egy rendezésből és egy beszúrásból. Először ezek kell megadni lambda-kalkulussal, majd át lehet őket írni SKI kombinátorokra.

Sort:

A rendezés függvény leveszi a paraméteréül kapott lista első elemét, rekurzív meghívja magát a maradék listán, majd az eredményébe, ami a rendezett maradék lista, beszúrja az első elemet a beszúrás függvény segítségével.

, ahol a eredménye

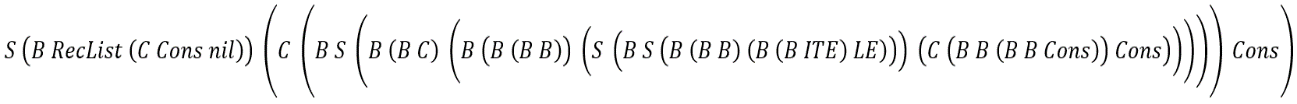
Az átírást kombinátorokra úgy kell elvégezni, hogy a lambda változókat bentről kifelé haladva el kell távolítani a kifejezésből, konzisztens módon.

Insert:

A beszúrás függvény egy elemet és egy rendezett listát kap paraméterül és beszúrja az elemet a listába a rendezésnek megfelelő pozícióba.

ahol az rekurzív hívás eredménye.

Az a kombinátorokra való átírás után így néz:



Az eredményül kapott kifejezést érdemes definícióként eltárolni, hiszen meglehetősen hosszú és a kitalálása nem igazán intuitív, ráadásul a kifejezés hivatkozik a B, C és LE kombinátorokra, amelyek nincsenek beépítve a nyelvbe, a beszúró rendezés definiálása előtt ezt a három kombinátort szintén meg kell adni definícióként. Ha a B és C kombinátorok nem lennének definícióként eltárolva, az -ben explicit kellene leírni azokat és a már fenti formában látható hosszú kifejezés körülbelül a kétszeresére növekedne.

Az algoritmus kifejezésében nem kell kiírni a kombinátorok típusait se a mély, se a sekély beágyazott megvalósítással sem. Az egyes kombinátorainak a típusát program nélkül nagyon bonyolult és hosszadalmas lenne meghatározni és ha minden kombinátorhoz kiírnánk a típusát, az eredmény több sor hosszúságú kifejezés lenne. A beszúró rendezést a két mély beágyazott megoldás közül a kétirányú típusellenőrző hajtja végre gyorsabban, majdnem fele annyi idő alatt, mint az egységesítéssel. A sekély beágyazás viszont már kevésbé eredményes a beszúró rendezés végrehajtásával. Üres és egy elemű lista bemenetre el tudja végezni a műveletet, de hosszabb listák esetén StackOverflowError-t kapunk. Ezzel szemben a mély megvalósítással akár 100-200 elemű listákra is működött a rendezés.

# 6. Összefoglalás

A kutatás során sikeresen elkészült az SKI kombinátor kalkulusra épülő típusos nyelv, a Típusos SKI. A mély beágyazott technikák közül a kétirányú típusellenőrzővel kezdetben, amikor a legtöbb kombinátor típusát explicit kellett kiírni nagyon nehéz volt akár kisebb programokat is megadni. Ezen sokat javított a típusparaméterek bevezetése, a kétirányú típusellenőrző ismeretlen típussal való kiegészítése és az egységesítés típusellenőrző implementálása. Mivel az összetettebb kifejezések még a típusok kiírása nélkül is hosszúak és nehezen kitalálhatóak, be lett vezetve a definíciók használata. Kombinátorokkal programozni nem egyszerű és nem intuitív, de ezekkel a megoldásokkal sikerült egy kicsit felhasználóbarátabbá tenni.

A sekély és a két mély beágyazott megvalósítás mindegyikének megvan a maga előnye és hátránya. A kétirányú típusellenőrző előnye, hogy hatékonyabb, mint az egységesítés olyan szempontból, hogy gyorsabb és nagyobb értékeket is képes kezelni és a sekély beágyazásnál mélyebb rekurziót tartalmazó kifejezéseket is ki tud értékelni. Hátránya pedig az elkészítésének a bonyolultsága és a nyelv további nyelvi elemekkel történi kibővítése esetén, ehhez a megvalósításhoz lenne szükség a legtöbb változtatásra. Az egységesítés módszer előnye a kétirányú típusellenőrzőhöz képest az elkészítésének és karbantarthatóságának az egyszerűsége, a sekély beágyazáshoz képest pedig, a mély rekurziók jobb kezelése. Pl.: a beszúró rendezés. A hátránya

végén leírni, hogy a sekély jobb ha csak gyorsan kell elkészíteni, a mély pedig ha személyre szeretném szabni vagy debugget akarok készíteni

# 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.”. |

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

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

Lista elemek összegzése

Összehasonlításhoz:

sekély vs mély 1-2

mély a leglassabb, mikor jobb mégis

: a mély személyreszabhatóbb, pl debuggolás fejlesztés közben vagy debugger vagy ha IDE-t készítünk

hogyan lehet a stackoverflowkat fixálni: hibrid impl? , stack size növelése,optimalizáció