A Clean dinamikus típusrendszere és implementációjának továbbfejlesztése

Ivicsics Mátyás

Eötvös Loránd Tudományegyetem
Természettudományi Kar
programtervező-matematikus szak
e-mail: mathiasr@inf.elte.hu

Témavezetők:

Dr. Horváth Zoltán
Eötvös Loránd Tudományegyetem
hz@inf.elte.hu

Martijn Vervoort
Katholieke Universiteit Nijmegen
martijnv@cs.kun.nl

A dolgozat az ELTE Általános Számítástudományi Tanszék és a Nijmegeni Egyetem Clean csoportja közötti együttműködés keretében, a SOCRATES/ERASMUS program és a T37742 sz., „Elosztott funkcionális programok helyessége" c. pályázat támogatásával készült.

Tartalomjegyzék

Előszó **4**

Bevezetés **4**

1. A funkcionális programozás **5**
1.1. Gráfátírás 5
1.2. Lustaság 5

2. A Clean nyelv néhány eszköze **7**
2.1. Alapfogalmak 7
2.2. Első programok 7
2.3. Mintaillesztés 8
2.4. Feltételek 9
2.5. Infix operátorok 10
2.6. Magasabb rendű függvények 10
2.7. Típusváltozók - Polimorfizmus 10
2.8. Rendezett n-es 11
2.9. Listák 12
2.10. Lista generátorok 13
2.11. Algebrai adattípusok 14
2.12. Láthatóság, struktúráltság 15
2.13. A Unique tulajdonság 15

3. A dinamikus szerkesztés **19**
3.1. A dinamikus szerkesztésben rejlő lehetőségek közül néhány . . 19
3.2. A `Dynamic` típus 20
3.2.1. Elvárások a típussal és az általa nyújtott lehetőségekkel szemben 20
3.2.2. A `Dynamic` típus műveletei 20
3.3. A rendszer működése 22
3.4. Alkalmazási példa 24
3.4.1. A példa felépítése 24
3.4.2. Fordítás, futtatás 26
3.5. A továbbfejlesztéshez vezető kérdések, feladatok 28
Függőségek 28
3.5.1. Nehézkes kezelhetőség 29

3.5.2. Szemétgyűjtés 29
3.5.3. Kódállományok másolása, küldése más számítógépre . . 29
3.6. A továbbfejlesztés közben felmerült kérdések 29
3.6.1. Körülhatárolható „véges rendszer” problémája 29
3.6.2. Egyedi fájlnevek problémája 30
3.7. A rendszeren végrehajtott módosítások 33
3.7.1. Új fájlnévmeghatározás 34
3.7.2. A fájlok elrejtése 34
3.7.3. Szemétgyűjtő 35
3.7.4. Küldő 36
3.8. A változtatásokkal elért eredmények 36

Konklúzió **38**

Irodalomjegyzék **39**

Előszó

A Clean funkcionális programozási nyelvet a Nijmegeni Egyetemen, Hollandiában fejlesztik. Erasmus ösztöndíj keretében volt szerencsém öt hónapot eltölteni ezen az egyetemen és részt venni az ott folyó munkában. A fejlesztés jelenleg több szálon folyik egyszerre: generic lehetőségek biztosításán, helyességbizonyítási eszközök kialakításán és végül a dinamikus programszerkesztés elkészítésén dolgoznak az ottaniak. Munkámmal a legutóbbi projectbe kapcsolódtam be.

Ezen TDK dolgozat a munkám során felmerült problémákat és megoldásaikat tartalmazza. Igyekeztem a dolgozat tárgyának kifejtése elé olyan bevezető fejezeteket írni, hogy a csupán imperatív programozást ismerő olvasó is megértse a programozás funkcionális megközelítését és a szemléltető programozási példákat. Aki bővebben szeretne ebben a témában tájékozódni, annak ajánlom az irodalomjegyzékben szereplő [1] illetve [7] irodalmakat.

Bevezetés

A Clean csoport már régóta dolgozik azon, hogy a programok Clean nyelven megírt és lefordított kódot tudjanak valamiféle fájlként a háttértárra menteni, majd az így elmentett mobil kódot később más programok futásidőben betölthessék és végre hajthassák. A cél megvalósítása közben fájlok közötti szemétgyűjtési probléma merült fel, melynek kapcsán mint később kiderült, a rendszer működésének átgondolása, módosítása és kiegészítő programok írása vált szükségessé.

Az első fejezetben néhány szóban a funkcionális programozásról és a későbbiekhez szükséges két alapfogalomról írok (gráfátírás, lustaság), a második fejezet példákon keresztül bevezeti az olvasót a funkcionális szemléletbe a Clean nyelv kiválasztott eszközeinek ismertetésével és a harmadik fejezet szól a `Dynamic` típusról, annak megvalósításáról, a megvalósítás módjában felmerült problémákról és megoldásaikról.

1. A funkcionális programozás

Tisztán funkcionális programozásról akkor beszélünk, ha a programban kizárólag a matematikai függvényfogalmnak megfelelő programnyelvi eszközöket használunk. A funkcionális nyelvekben -mint a Haskell vagy a Clean- a programozó nem használhat értékadást, sem ciklust. A program elkészítése egy olyan matematikai függvény megírását jelenti, amelynek kiértékelése során a számítógép (ha szükséges interaktívan) megoldja a rábított feladatot. Rekurzióval, függvénykompozícióval lehet dolgozni, továbbá számos hasznos típus áll rendelkezésre mint például a „MayBe".

A programozás ezen stílusa más szemléletet igényel mint az imperatív nyelvek. Előnyei vitathatatlanok: a program viselkedése, működése nagyságrendekkel átláthatóbb, nem kell mellékhatásokra számítani és nem befolyásolják külső körülmények a program sikeres lefutását; a programozó megfogalmazhat a függvényekről vagy az egész programról matematikai nyelven állításokat melyeket aztán matematikai eszközökkel bizonyíthat is.

1.1. Gráfátírás

A Clean a kifejezések kiértékelésére gráfátírást alkalmaz, ami egy hatékony átírási modell. A módszer a következőképpen működik: tulajdonképpen minden egyes függvénydefiníció egy átírási szabály, vagyis ha egy kifejezés tartalmaz egy függvényalkalmazást aktuális paraméterekkel, akkor beírhatjuk a helyére a függvény törzsének definícióját a megfelelő formális paramétereket a megfelelő aktuális paraméterekre cserélve. Ezzel egy új kifejezést kaptunk, mely újabb függvényalkalmazásokat tartalmazhat, így az átírási lépést folytatnunk kell. Amennyiben a kapott kifejezés nem tartalmaz több függvényalkalmazást, az a normálforma[2].

1.2. Lustaság

Az előbb leírt redukciós módszer nem definiálja a behelyettesítések sorrendjét. Ha tehát egy kifejezésben több függvényalkalmazás is található, tetszőleges sorrendben helyettesíthetjük be őket. Ezt a sorrendet igyekeznek optimálisan meghatározni a különböző kiértékelési stratégiák.

A Clean nyelv a *lusta* kiértékelési technikát használja. A módszer lényege az, hogy egy argumentumot csak akkor értékel ki, amikor már épp szükség lenne rá – tehát valóban szükség van rá. Ez azokban az esetekben előnyös,

amikor –mert ez is elképzelhető– a kifejezés kiértékeléséhez elég csupán a benne szereplő első vagy első néhány függvényalkalmazás behelyettesítése, a többitől pedig független az eredmény – ilyenkor nem akarunk feleslegesen dolgozni. Ennek felismerésével a lusta kiértékelés nem csak időt takarít meg, de kikerüli az olyan csapdákat is, amikor a többi függvény kiértékelése lehetetlen akár nem definiált érték miatt, (pl. `True || (1/0 <= 2)` – ahol „||" a logikai *vagy* műveletet jelöli) akár nem termináló számítás miatt (pl. összes prímszám).

A Clean programok lefutása nem más, mint a `Start` függvény átírási lépések sorozatával történő normálformává alakítása, a lusta kiértékelési stratégiát követve.

2. A Clean nyelv néhány eszköze

Az alábbiakban szeretném néhány egyszerű példán bemutatni, hogy hogyan is néz ki egy Clean program, miféle eszközökkel dolgozhat a programozó, hogy a későbbiekben szereplő programozási példák érthetőek legyenek. Bővebb és teljesebb leírás az [1] és [7] irodalmakban található.

2.1. Alapfogalmak

Minden Clean program tartalmaz `Start` függvényt – ennek definiálása kötelező. A futtatás célja ezen függvény kiértékelése.

A Clean lehetővé teszi a moduláris programozást. A definíciós modulok forrásfájljainak kiterjesztése mindig *dcl*, az implementációs moduloké és a főmodulé *icl*. Modulokat a forrásba beszerkeszteni az *import* kulcsszóval lehet. Csak a főmodul tartalmazhatja a `Start` függvény definícióját. Számos előre definiált könyvtári modult használhatunk programozáskor ilyen például az `StdBool` vagy az `StdInt` illetve az `StdEnv` ami az előbbieket és a hasonlóan alapvető modulokat mind magában foglalja.

2.2. Első programok

A „Hello world!" program megírása első körben igen egyszerű. A `Helloworld.icl` így néz ki:

```
module Helloworld
Start= "Hello World!"
```

A fordító látja, hogy a `Start` ebben az esetben nem más mint egy konstans függvény, az idézőjelekből láthatóan a típusa `String`, értéke pedig a *Hello World!*. A program eredményének konzolra történő kiírása automatikus.

A függvények típusát lehetőségünk van külön megadni a kódban, ilyenkor két kettőspont válaszja el a függvény nevét a típusától. Az előbbi kis program esetén tehát a

```
Start :: String
```

sort illeszthettük volna be a két sor közé.

Nézzünk meg egy nem konstans függvényt! Egyszerű példaként definiáljuk az ellentett függvényt `neg` néven, majd számoljuk ki a -5 ellentettjét!

Ahhoz, hogy az `Int` típus műveleteit használhassunk, szükségünk lesz az *StdInt* modulban lévő definíciókra – az import kulcsszóval jelezzük majd, hogy ezt a modult szeretnénk beszerkesztetni a programunkba. A kód tehát:

```
module Ellentett
import StdInt

Start= neg -5

neg :: Int -> Int
neg x= -1 * x
```

A `neg` függvény esetében megadtuk a fordítónak, hogy `Int` típusú paraméterből `Int` típusú eredményt állít elő. A következő sorban adtuk meg a definíciót, amelyben leírjuk, hogy tetszőleges `x` bemenetből hogyan számolhatjuk ki az eredményt.

A típus megadása nem kötelező egyetlen függvény esetében sem, mindaddig amíg a definícióból a fordító le tudja azt vezetni – ebben az esetben például látszik, hogy a függvény paraméterét szeretnénk a `-1` `Int` értékkel szorozni, ezért a paraméter típusa csakis `Int` lehet, a számítás eredménye pedig (ami a visszatérési érték) szintén `Int` lesz.

Amint láthatjuk a Clean függvénydefiníciói nagyon hasonlítanak a matematikai függvénydefiníciókhoz, és meglehetősen elütnek az imperatív programozási nyelvek világában megszokottaktól. Az egyszerű és kifejező szintaxis a Clean egyik nagy erőssége.

2.3. Mintaillesztés

A következő példa azt mutatja, hogy egy függvény definiálásakor hogyan lehet *mintaillesztés*es módszerrel dolgozni. A faktoriális számítás alapesete a nulla. Tetszőleges pozitív egész faktoriálisát rekurzívan értelmezzük (a továbbiakban a modulnévtől, az `import` parancsoktól és a `Start` függvény definiálásától eltekintek):

```
fac :: Int -> Int
fac 0= 1
fac n= n * fac(n-1)
```

Mintaillesztéses függvény kiértékelése esetén az első illeszthető ág kerül behelyettesítésre.

A mintaillesztésben „joker” mintát is használhatunk, ez a „_” jel, erre a mintára minden kifejezés illeszkedik.

2.4. Feltételek

Fontos észrevennünk, hogy az előbbi függvény negatív számmal meghívva végtelen rekurzióhoz vezet. Olyan feltételt a mintaillesztéssel nem tudunk kifejezni, hogy „pozitív argumentum esetén”. A kapcsos zárójeles függvények definiálásához *feltétel*eket használunk. Klasszikus példaként vegyük a maximum függvényt, ami két egész szám paramétert vár, és azok közül kiválasztja a nagyobbikat. A típusmegadásnál a két paraméter típusát egyszerű egymás-melléírással jelöljük.

```
maximum :: Int Int -> Int
maximum n m
   | n<m  = m
   | n>=m = n
```

A feltétel nem más mint egy logikai kifejezés, melyet közvetlen a minta után írhatunk egy ‘|’ jel és az egyenlőségjel közé. A fenti példában a második ág azonnal láthatóan az első komplementere. Ilyen esetekben az utolsó feltételt helyettesíthetjük az otherwise kulcsszóval, vagy egyszerűen el is hagyhatjuk:

```
maximum :: Int Int -> Int
maximum n m
   | n<m   = m
   = n
```

Végül egészítsük ki feltétellel a faktoriális számító függvényünket:

```
fac :: Int -> Int
fac 0= 1
fac n | 0<n = n * fac(n-1)
```

Az így kapott függvényünk parciális, vagyis negatív paraméterre nem tartalmaz definíciót. Ha mégis meghívjuk negatív paraméterrel, a kiértékelés hibaüzenettel leáll: nem talált megfelelő ágat.

2.5. Infix operátorok

Definiáljuk most azt a hatványozás operátort ami egy egész számot pozitív egész kitevőjű hatványra emel. Ezt '^' jellel szoktuk jelölni és szeretjük a két operandus közé írni. A Clean lehetőséget ad ilyen operátorok definiálására. A definíció megadásakor az operátor nevét még mindig a sor elejére írjuk, viszont zárójelbe tesszük a következő képpen:

```
(^) infixr 8 :: Int Int -> Int
(^) x 0 = 1
(^) x n = x * x ^ (n-1)
```

A típusleírásban található infix kulcsszó jelzi a fordítónak, hogy infix módon szeretnénk használni a függvényt, a hozzáragasztott 'r' pedig az operátor jobbra asszociatívságát jelenti. Balra asszociatív tulajdonságot az l' betű hozzáragasztásával adhatunk meg.

A 8-as szám az operátor precedencia szintjét jelenti. A Clean összesen 12 precedencia szintet tart nyilván (0-11). A 11-es szint tömbindexelésnek illetve rekord mező kiválasztásnak van lefoglalva, a 10-es függvényalkalmazásnak. A programozó által definiált operátorok maximum 9-es precedenciát kaphatnak. (A '+', '-' operátorok 6-os, a '*', '/' operátorok 7-es precedencia szintűek.)

2.6. Magasabb rendű függvények

Mivel Clean-ben egy `Int` érték sem más mint egy konstans függvény, nem jelenthet problémát a magasabbrendű függvények definiálása. Magasabb rendű függvények azok, amelyek nem konstans függvényt (is) kapnak paraméterül illetve adnak vissza eredményül. Példaként tekintsük a `twice` függvényt, amely a paraméterül kapott *érték*ből kiindulva kétszer alkalmazza a paraméterül kapott *függvény*t:

```
twice f x = f (f x)
```

Ennek definiálása után például a `twice neg -5` hívással -5-öt kapunk eredményül, a `twice fac 3` pedig 720-at ad.

2.7. Típusváltozók - Polimorfizmus

A típusváltozókat arra használjuk, hogy különböző típusokra azonos szignatúrával értelmezett függvényeket csupán egyszer, altalánosan kelljen definiálni. Vannak például olyan függvények, melyeknél az argumentum típusa

érdektelen. Ilyen az identitás, mely visszaadja az argumentumként kapott értéket:

```
Id x= x
```

Ennek a függvénynek a típusát típusváltozó segítségével adhatjuk meg:
`Id :: a -> a`.

Szűkíthetjük a típust azzal, hogy feltételül szabjuk bizonyos műveletek meglétét az adott típusra a '|' jel mögé írva. Ezzel a módszerrel definiálhatjuk például a négyzetre emelést egyszerre minden olyan típusra amelyre a szorzás művelet értelmezve van:

```
square :: a -> a | * a
square n= n*n
```

Definiálhatjuk a fenti maximum függvényt is álatalánosabban:

```
maximum :: a a -> a | < a
maximum n m
   | n<m  = m
   | n>=m = n
```

itt az összehasonlítás művelet meglétét kötöttük ki az általános típusra.

Típusváltozó segítségével megadhatjuk a twice függvény típusát is:
`twice :: (a -> a) a -> a`.

A típusváltozó használatával definiált függvények polimorfikusak, aktuális típusuk az éppen paraméterül kapott értékek típusától függ.

2.8. Rendezett *n*-es

Lehetőség van rendezett párok, hármasok, négyesek stb. létrehozására is Clean-ben. Az rendezett *n*-es jele a gömbölyű zárójel, ez fogja egységbe az *n*-es elemeit, melyeket ',' választ el egymástól. Például definiáljuk azt a négyzetre emelés függvényt, ami a paraméterül kapott `t` típusú értéknek nem csak a négyzetét adja vissza, hanem az eredeti számot is. Hasznos lehet ez például akkor amikor négyzetre emelés után is tudni szeretnénk a négyzetre emelt eredeti szám előjelét:

```
square :: a -> (a,a) | * a
square n= (n*n,n)
```

Ezen függvény definiálása után a `square -3` hívás például a `(9,-3)` párt adja vissza.

A rendezett n-es konstrukció ad lehetőséget arra, hogy összetartozó adatokat együtt kezeljünk illetve hogy olyan függvényeket definiáljunk melyektől egynél több értéket szeretnénk visszakapni.

2.9. Listák

A funkcionális programozásban az egyik leggyakrabban használt összetett típus a lista. Egy lista tetszőleges számú, tetszőleges –de azonos– típusú elemből állhat. A listát mindig szögletes zárójelek között jelöljük. Néhány példa:

`[]`	üres lista
`[5]`	egyetlen elemet tartalmazó lista
`[5,8,3]`	három elemű lista
`[1..5]`	1-től 5-ig egyesével
`[1,3..10]`	1-től 10-ig kettesével (a 10 nem eleme)
`[1..]`	az összes természetes számot tartalmazó lista
`[x:y]`	ennek a listának `x` az első *eleme* (head) amit az `y` *lista* követ (tail)

Tekintsük példaként azt a függvényt, amely a neki paraméterül adott lista elemeinek szorzatát számolja ki. A típusmegadásnál igyekszünk a lehető legáltalánosabban eljárni, a különböző eseteket pedig mintaillesztéssel különböztetjük meg:

```
product :: [a] -> a | one, *a
product []    = one
product [x:r]= x * product r
```

Az `[a]` `a` típusú elemekből álló lista típust jelöl, a `one` pedig az egységelem meglétét szabja feltételül az `a` típusra. Minthogy az utolsó ág `r`-je szintén `[a]` típusú (tehát lista), alkalmazhatunk rá rekurzív hívást.

Listák segítségével a faktoriális függvény is egyszerűbb alakra hozható:

```
fac :: Int -> Int
fac n | 0<n = product [1..n]
```

2.10. Lista generátorok

Kényelmesebbé teszi a programozó munkáját a listagenerálás lehetősége. Ez egy speciális szintaxissal lehetséges. Példaként generáljuk azt a listát, amely az egy és tíz közötti néggyel nem osztható számokat és azok faktoriálisát tartalmazza rendezett párok formájában:

```
[(x,fac x) \\ x<-[1..10] | x rem 4 <> 0]
```

A `\\` és `|` szimbólumok közötti rész tartalmazza a kiinduló listát, a `|` utáni rész egy szűrő a lista elemeire, a legelső rész pedig az eredménylista elemeinek konstrukcióját.

Listagenerátorok alkalmazásával létrehozhatjuk a prímszámok listáját is amennyiben az alábbi módon definiáljuk a `sieve` függvényt és meghívjuk az `[2..]` paraméterrel:

```
sieve [p:xs] = [p: sieve [ i \\ i <- xs | i mod p <> 0]]
```

Első ránézésre szemet szúr, hogy a `sieve [2..]` kifejezés kiértékelése nem terminál. Ezen segít a 1.2 alfejezetben leírt *lusta* kiértékelési stratégia:
Ha elő szeretnénk állítani az első száz prímszámot, ahhoz venni kell a prímszámok végtelen listájának első száz elemét:

```
Start= take 100 (sieve [2..])
```

ahol a `take` függvény az <első paraméter> darab elemet veszi a <második paraméter>-ként adott listából a következőképpen:

```
take :: Int [a] -> [a]
take n [] = []
take n [a:x] = [a:take (n-1) x]
take 0 _ = []
```

A kiértékelés nem a prímszámok előállításával kezdődik, hogy aztán majd ha az összeset előállítottuk akkor kiválasszuk az első százat, hanem a `take 100` függvény behelyettesítésével aminek szüksége van az első prímre. Az első prím előállítása után látjuk, hogy a rekurzív hívás miatt a `take 99` függvényt kell végrehajtanunk a prímek hátralévő részére, ezért előállítjuk a prímek listájának második elemét... és így tovább. Így jutunk el a `take 0` függvényhez, aminek bármi is már a további paramétere, az értéke definíció szerint az üres lista: `[]`, tehát a prímek listájának további elemeit felesleges előállítani – a program az első száz prím meghatározása után szabályosan végetér.

2.11. Algebrai adattípusok

Az algebrai adattípusok definiálásának legegyszerűbb formája az, amikor felsoroljuk a típus konstruktorait. Példaként tekintsük az *évszak* típus definícióját, és hozzuk létre a *tavasz*nak megfelelő konstans függvényt:

```
:: Evszak = Tavasz | Nyar | Osz | Tel
tavasz :: Evszak
tavasz= Tavasz
```

Algebrai adattípusok konstruktorainak lehetnek különböző típusú argumentumaik is. Ha az évszak tavasz, kiváncsiak lehetnénk, hogy hanyadik napján van Húsvét hétfő, ha pedig tél, mellette lehetne, hogy esett-e a hó:

```
:: Evszak = Tavasz Int | Nyar | Osz | Tel Bool
tavasz= Tavasz 32
tel= Tel True
```

Mivel az argumentum típusa akár az épp definiálandó típus is lehet, rekurzívan is készíthetünk típusokat. A következő sor a lista típust definiálja. Az tartalmazott elemek típusát általánosan –típusváltozóval– adjuk meg:

```
:: List a = Nil | Cons a (List a)
egylista :: List Int
egylista= Cons 3 (Cons 4 (Cons 6 Nil))
```

Természetesen a Clean nyevben lista alatt sohasem az előbb definiált algebrai adattípust értjük, hiszen a lista előre definiált típus amint azt a 2.9. alfejezetben leírtam.

Végül az algebrai adattípus gyakori alkalmazásaként tekintsük egy tetszőleges típusú elemeket tartalmazó fa definícióját, és írjuk meg azt a függvényt ami a fát postorder módon bejárja és listát készít belőle:

```
:: Tree a = Empty | Node (Tree a) a (Tree a)

toList :: (Tree t) -> [t]
toList Empty = []
toList (Node left elem right) = toList left ++ [elem] ++ toList right
```

A `toList` definíciójában használt „++" előre definiált művelet a listákra: konkatenációt jelent.

2.12. Láthatóság, struktúráltság

A Clean nyelvű forráskód láthatósági szabályainak alapvető meghatározására a baloldali margó szolgál. Egy függvénydefinícióban azokat a függvényeket *látjuk* (használhatjuk), amelyeknek a definícióját legalább annyival beljebb kezdtük a margótól, mint az adott definíciót.

A *where* kulcsszó mögött megadhatunk lokális definíciókat is. Ezeket csak annak a függvénynek a törzsében használhatjuk fel, amihez a where tartozik, a lokális definíciók pedig használhatják a függvény paramétereit. Tekintsük példaként azt a függvényt, amelyik két paraméterként kapott lista közül kiválasztja a hosszabikat:

```
maxLength :: [a] [a] -> [a]
maxLength list1 list2
    | a>=b = list1
    = list2
where
    a= length list1
    b= length list2
```

A *length* a lista típusra előre definiált függvény, a lista elemeinek számát adja meg.

Lokális definíciókat úgy is írhatunk, hogy a sort a '#' karakterrel kezdjük. Az előbbi függvénydefiníció ebben az esetben így néz ki:

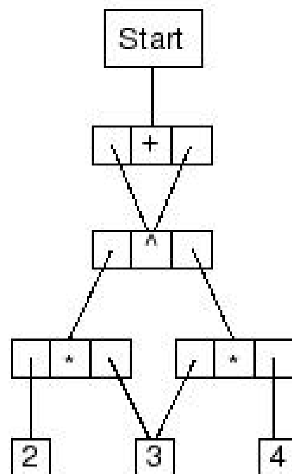
```
maxLength :: [a] [a] -> [a]
maxLength list1 list2
    #a= length list1
    #b= length list2
    | a>=b = list1
    = list2
```

2.13. A Unique tulajdonság

A funkcionális programozási stílus egyik alapvető előnyeként említettem, hogy nem kell mellékhatásokra számítani. Ez azt jelenti, hogy egy kifejezés kiértékelése mindig ugyanazt az eredményt fogja adni, akárhányszor és akármilyen körülmények között hajtjuk is végre. Ezt a tulajdonságot ki is

használhatja futtatáskor a rendszer, hiszen ha egy részkifejezés többször előfordul egy kifejezésben, lehet, hogy csupán egyszer értékelődik ki. Például a `2*3^3*4 + 2*3^3*4` kifejezés kiértékelése esetén a `2*3^3*4` kifejezés kiértékelése csak egyszer hajtódik végre, felesleges lenne kétszer kiszámolni.

Technikailag ennek megvalósítása úgy történik, hogy a kifejezés kiértékelési fájában a többször előforduló részkifejezés részfája csupán egyszer szerepel, de két csúcsból is hivatkozunk rá. A kiértékelése akkor történik meg amikor a két csúcs valamelyikének értékét szeretnénk meghatározni, megtörténte után pedig a részfát a számítás eredményét tartalmazó csúccsal helyettesítjük. Amikor a második rá hivatkozó csúcs értékét szeretnénk meghatározni, a kifejezés helyén már kész értéket találunk, így a számítás nem hajtódik végre mégegyszer[2]. A fenti példa esetén a kiértékelést az alábbi fával ábrázolhatjuk:



Mit lehet kezdeni ilyen körülmények között egy olyan programmal, aminek kapcsolata van a külvilággal? Ha például egyik függvényünk hozzáfűz egy szövegfájlhoz egy sort, minden olyan kifejezésünk értéke megváltozik amiben az adott fájl szerepel. Nem mindegy többé, hogy a kifejezéseket milyen sorrendben értékeli ki a program. Ha előbb az író függvény értékelődik ki majd utána egy olyan függvény ami kiolvassa a tartalmát és kiírja, más jelenik meg a képernyőn mintha előbb az olvasó majd az író függvényt hajtjuk végre.

A probléma megoldása az, hogy nem engedjük meg azon csúcsok értékének megváltoztatását a kiértékelési fában, amelyekre több csúcsból is van

hivatkozás. Amely kifejezésre viszont csupán egyetlen csúcs hivatkozik, azt (annak a csúcsnak a kiértékelése során) meg is változtathatjuk. Ha olyan függvényt szeretnénk írni, amelyik a paraméterül kapott objektumot meg szerené változtatni, jeleznünk kell a szignatúrában a „*" karakter segítségével, hogy elvárjuk a paraméterként kapott objektumtól, hogy egyszeresen hivatkozott legyen. Ezek után lehetünk csak ugyanis biztosak abban, hogy a változás nem érinti a programban szereplő más függvények kiértékelését. Ilyen szignatúrával rendelkező függvények esetén a fordító ellenőrzi, hogy a függvénynek alkalmazáskor átadott aktuális paraméterek valóban egyszeresen hivatkozottak-e. A kifejezések azon tulajdonságát, hogy legfeljebb egyetlen csúcsból hivatkozzák őket *unique* tulajdonságnak nevezzük.

Unique world

A külvilágot a Cleanben egy *unique world* vagyis `*World` típusú objektummal reprezentálhatjuk. Az interaktív programok `Start` függvényének paramétere a külvilág. Egy külvilággal bánó függvény ezt kapja paraméterül, és végrehajtása után az általa megváltoztatott világ értékét visszatérési értékként viszszaadja. Minden további függvény már csupán ezt kaphatja paraméterül... és így tovább. Ezt a módszert *environment passing*nak vagyis a *környezet továbbadásá*nak hívják, alkalmazása a külvilággal bánó függvények végrehajtásának sorrendjét is meghatározza.

A `*World` típus használatának megértéséhez tekintsük azt az egyszerű programot, ami a felhasználó által begépelt sort megfordítva kiírja:

```
Start :: *World -> *World
Start world
    #(console, world)= stdio world
    #console= fwrites "Enter a text\n" console
    #(text, console)= freadline console
    #console= fwrites ((toString (reverse (list text)))+++"\n") console
    #(ok,world)= fclose console world
    |ok= world
    = abort "Failure while closing console"
    where
        list :: String -> [Char]
        list str= fromString str
```

A '+++' művelet előredefiniált a `String` típus konkatenációs műveleteként, a `toString` és `fromString` műveletek pedig azért kellenek, hogy a `reverse` –lista típusra előredefiniált megfordító– függvényt alkalmazhassuk.

A program írásakor felhasználtam a lokális definíciók készítésének mindkét módszerét. A '#' használatának előnyös tulajdonsága, hogy a definiált objektum nevével eltakarhatjuk a definícióban felhasznált objektumok nevét. Így tettünk, valahányszor a `console` és `world` *unique* tulajdonságú objektumokat megváltoztattuk. A visszakapott `world` illetve `console` természetesen minden esetben teljesen új, csupán a nevét használtuk fel újra a megszüntetett réginek.

3. A dinamikus szerkesztés

A fejlesztés során az az ötlet merült fel, hogy lehetne valami olyan eszközt a programozó kezébe adni, amivel mindenféle nyelvi egységet –egy `Int` értéket, egy algebrai típusdefiníciót vagy akár egy függvényt– elmenthet a háttértárolóra. Az eltárolt kódot az alkalmazások futás közben *dinamikusan* beszerkeszthetnék és használhatnák.

Természetesen ennek a megvalósítása sok problémát felvet, a megléte viszont sok lehetőséget rejt.

3.1. A dinamikus szerkesztésben rejlő lehetőségek közül néhány

1. Ha egy számítógépen fenn lenne például a StandardIO könyvtár eltárolt kód formájában, nem kellene azt minden egyes programba az „import" kulcsszóval statikusan belefordítani és feleslegesen sok példányban tárolni, hanem a programok egyszerűen a megadott lelőhelyről futási időben beszerkesztenék és kiértékelhetnék a meghívott függvényeket, használnák az ott definiált típusokat.

2. A programok tudnának a *kódállományok*on[1] keresztül információt cserélni. Ez elvileg lehetséges lenne „sima" fájlokon keresztül is egyszerű írás olvasással, csakhogy veszélyt rejtene magában, hogy a beolvasó program egyszerűen elhiggye, hogy a kiíró a megfelelő adatot megfelelő formátumban és nem valami értelmetlenséget írt ki a közösen használt fájlba. Ha a kiíró program az átadni kívánt `Int` értéket kódállományon keresztül adja át, biztosak lehetünk benne, hogy a megfelelő típusellenőrzés beszerkesztés előtt végrehajtódik.

3. Lehetségessé válna a (csupán elvben létező) Clean nyelven megírt internet böngészőhöz „plug-in"-t készíteni - vagyis interneten keresztül elküldeni kódállomány formátumban a végrehajtani kívánt kódot és beszerkeszteni az épp futó böngészőbe.

4. Megvalósítható lenne a „típusos operációs rendszer", amelyben minden egyes fájl alapvetően kódállomány. Ebben a rendszerben a begépelt pa-

[1] Így nevezem a továbbiakban az eltárolt kódot tartalmazó és dinamikusan beszerkeszthető fájlokat

rancsok mindegyikének pontosan meghatározható típusa lenne amelynek a megfelelő típusú paraméterekre lenne szüksége. Ez a gondolatkör messze vezet, de a lényege az, hogy az így elkészült operációs rendszer rendkívül biztonságosan bánna a tárolt adatokkal és működésében igen megbízható lenne[5].

A Nijmegeni csoport az ötletet nagyrészt valóra váltotta 2002 januárjára. A rendszer működésének elméleti megalapozása megtörtént[3], korlátozott keretek között már az implementációt is tanulmányozni, tesztelni lehetett.

3.2. A `Dynamic` típus

A megvalósítás alapötlete a `Dynamic` típus. Ez egy új típus ami bekerült a statikus típusrendszerbe. Arra jó hogy az elmenteni kívánt nyelvi egységeket egy általános típusú elemként kezelhessük. Egyszerű „csomagolás"-sal bármely típusú nyelvi egységet `Dynamic` típusúvá tehetünk, kicsomagoláskor pedig dinamikusan derül ki, hogy az adott érték milyen típushoz tartozik és ennek megfelelően dinamikusan választjuk ki a kiértékelésre kerülő függvényt (dinamikus típusrendszer). A Dynamic típusra elkészült a fájlba kiíró és fájlból beolvasó művelet, ez ad lehetőséget kódállományok létrehozására és beszerkesztésére.

3.2.1. Elvárások a típussal és az általa nyújtott lehetőségekkel szemben

1. Semmiféle csomagolási vagy fájlkezelési művelet nem változtathatja meg a tárgykifejezés értékét, sem a kiértékeltségi állapotát.

2. Miután egy futó alkalmazásba beszerkesztődtek a külső fájlokból várt nyelvi egységek, a program végrehajtásának éppoly hatékonyan kell lezajlania, mintha statikusan szerkesztett programról lenne szó.

3. A mobil kód eltárolása és beszerkeszése a lehető leghatékonyabb legyen.

4. A nyelvbe integrálható könnyen érthető szintaxis és szemantika.

3.2.2. A `Dynamic` típus műveletei

1. **Becsomagolás**
   Tetszőleges típusú elemet becsomagolással `Dynamic` típusúvá tehetünk.

A létrejött elem statikus típusa `Dynamic`, dinamikus típusa a becsomagolt elem statikus típusával egyezik meg. A becsomagolás a `dynamic` kulcsszóval történik, paraméterül megadjuk a becsomagolni kívánt elemet, mellette pedig opcionálisan megadhatjuk még annak típusát is. Az alábbi példák tehát a legegyszerűbb `Dynamic` értékek:

`dynamic True`	a dinamikus típus `Bool`, az érték `True`
`dynamic True :: Bool`	ugyanaz, az opcionális típusmegadással
`dynamic fac :: Int -> Int`	a faktoriális függvény becsomagolása
`dynamic maximum 5 4`	egy `Int` típusú kifejezés becsomagolása

2. **Kicsomagolás**

   Egy `Dynamic` típusú elemből természetesen egyszer vissza szeretnénk nyerni a becsomagolt értéket, hogy ismét statikus típusú elemként lehessen használni. A kicsomagolás mintaillesztéssel működik. A mintaillesztés itt két fázisú: először végrehajtódik egy *típus* mintaillesztés, majd ha ez sikeresen végrehajódott, következik egy általános mintaillesztés. Ha a második mintaillesztés is sikeres, akkor következik a megfelelő ág végrehajtása. Tekintsük az alábbi példát, mely a faktoriális függvény `Dynamic` típusra:

   ```
   dynFac :: Dynamic -> Int
   dynFac (0 :: Int) = 1
   dynFac (n :: Int) | 0<n = n * dynFac (dynamic n-1)
   ```

   Az első mintaillesztés a függvény első ágában hajtódik végre a paraméter dinamikus típusára. Ha ez `Int`, akkor folytatódik a kiértékelés a `0` értékre való általános mintaillesztéssel. Ha ez is egyezik végrehajtódik a jobboldal, ha nem, a közvezkező ágra ugrunk. A következő ág mintaillesztése csupán a dinamikus típusra tartalmaz megkötést, ha ez `Int` a jobboldal végrehajtódik. A rekurzív hívás paraméteréül be kell csomagolnunk az `n-1` kifejezést.

   A fenti függvénydefiníciót a „`dynFac else = ...`” sorral bővíthetnénk még ki, amely a mintaillesztést (a típusét csakúgy mint az értékét) átugorja.

3. **`Dynamic` típusú elem kiírása fájlba**

   `Dynamic` típusú értéket a `writeDynamic` függvény segítségével lehet

fájlba írni. Ez ad lehetőséget a kódállományok létrehozására. A függvény típusa: `String Dynamic *World -> (Bool,*World)`. Az első paraméter a fájlnév, a második a menteni kívánt elem, a harmadik pedig a „Világ”. A visszaadott `Bool` érték a mentés sikeres végrehajtását jelzi. Egyszerű példaként tekintsük az alábbi függvényt mely a paraméterül kapott érték faktoriálisát menti el:

```
facToDisk :: Int *World -> (Bool,*World)
facToDisk v w =
   writeDynamic ("C:\\fac_of_"+++toString v) (dynamic fac v) w
```

4. **`Dynamic` típusú elem beolvasása fájlból**
   A kiírt `Dynamic`-okat az alkalmazások be is olvashatják a kövezkezőképpen definiált függvénnyel:
   `readDynamic :: String *World -> (Bool,Dynamic,*World)`.
   Így tudunk tehát kódállományt a programunkba beszerkeszteni. Példaként írjunk olyan függvényt, amelyik beolvassa az előzőleg definiált függvényünk által a `facToDisk 3 world` hívásra kiírt `Dynamic` típusú elemet, és ellenőrzi a számítást (a három faktoriálisa hat, tehát a kiírt `Dynamic` elem értéke `Int 6` kell hogy legyen):

```
checkFacThree :: *World -> (Bool, *World)
checkFacThree w= (isSix v, nw)
where
    (ok,v,nw)= readDynamic ("C:\\fac_of_3") w
    isSix :: Dynamic -> Bool
    isSix (6 :: Int)= True
    isSix else= False
```

Amint látható, a műveletek szintaktikusan illenek a nyelvbe, a Dynamic típus nyújtotta lehetőségek kihasználása programozói szinten kellőképpen egyszerű.

3.3. A rendszer működése

Amikor egy általános alkalmazást hozunk létre, a fordítás után statikus programszerkesztés és a kész program háttértárra mentése következik.

Egy `Dynamic` típust használó program készítésekor statikus szerkesztésre nem kerül sor. A program fordítása után az abban definiált függvények – még szimbolikus hivatkozásokat tartalmazó– nyers gépi kódja kerül mentésre egy *lib* kiterjesztésű fájlba, és a definiált típusok leírása egy *typ* kiterjeszésű fájlba.

A program futtatása egy *bat* kiterjesztésű fájl futtatását fogja jelenteni, ami parancssori hívást tartalmaz a *dinamikus szerkesztőprogram*ra a programhoz tartozó könyvtár-[2] illetve típusfájl[3] nevét argumentumként átadva. A dinamikus szerkesztőprogram szerkeszti és futtatja végrehajtáskor a programot.

Megjegyzendő, hogy a kódállományt nem beolvasó programok szerkesztése statikusan is történhetne, de a jelenlegi implementáció szerint a dinamikus szerkesztőprogram gondoskodik az összes `Dynamic` típussal kapcsolatos műveletről, és az ezeket használó programokat mind dinamikusan szerkeszti össze[4].

Amikor egy dinamikus alkalmazás kiír egy kifejezést kódállományba, a létrejövő fájl hivatkozást tartalmaz a programhoz tartozó könyvtár- illetve típusfájlra.

Ha egy másik dinamikus alkalmazás később beolvassa a kiírt kifejezést, akkor a dinamikus szerkesztőprogram először beolvassa a típusfájlból a megfelelő információkat, hogy a típus mintaillesztést el lehessen végezni – és el is végzi azt. Ha a típus mintaillesztése sikeres, és a kódállományban tárolt kifejezés kiértékelése valóban szükségessé válik (lusta a kiértékelés), akkor folytatódik az eljárás a kifejezést reprezentáló gráf felépítésével és a könyvtárfájl betöltésével, hogy a futó program memóriaterületén a kifejezés és a kiértékeléséhez szükséges függvények kódjai elérhetőek legyenek. A könyvtárfájl betöltése után a program a normál kiértékeléshez tér vissza.

A rendszer működésének hatékonysági szempontjai

Mivel nem magában a kódállományban tároljuk a kiértékeléséhez esetleg szükséges –az őt kiíró programban definiált– függvények kódját, kódmegosztás történik abban az értelemben, hogy az ugyanazon program által kiírt

---

[2] A „lib" kiterjesztésű fájlokat a továbbiakban *könyvtár*fájloknak nevezem.

[3] A „typ" kiterjesztésű fájlokat a továbbiakban *típus*fájloknak nevezem.

[4] Azokat a programokat amik a dinamikus szerkesztőprogram segítségével futási időben szerkesztődnek –minden a `Dynamic` típust használó program ilyen– alkalmanként *dinamikus alkalmazás*oknak nevezem.

kódállományok mind ugyanazokat a könyvtár- illetve típusfájlokat hivatkozzák. Így nem csak a tárolás hatékony, hanem a kódállományokat beolvasó programba sem kell kétszer beszerkeszteni a megfelelő függvénykódokat.

A könyvtár- illetve típusfájl fogalmának bevezetésével külön vannak választva az elmentett típusinformációk a függvénykódoktól, így amennyiben egy kifejezés fájlból történő beolvasásakor a típus mintaillesztése sikertelen, a kódállományhoz tartozó függvénykódok költséges beszerkesztése nem is történik meg.

Azzal, hogy a kódállományokhoz a könyvtár- illetve típusfájlokban már lefordított –gépi kódú– definíciók tartoznak, a beszerkesztés ezek egyszerű beolvasásával megtörténik, nincs szükség interpreter szerű fordításra.

3.4. Alkalmazási példa

Az egyszerű szemléltetés kedvéért tekintsük az alábbi –a fejlesztők által előszeretettel használt– példát!

3.4.1. A példa felépítése

Az **f** nevű program feladata, hogy kódállományként a háttértárra mentsen *function* néven egy függvényt, mely megszámolja egy „`:: Tree b = Node b (Tree b) (Tree b) | Leaf`” típusú fa leveleit és `Int` típusú értékkel tér vissza. (A fa típusleírásának jelentése: egy `Tree b` típusú fa levelei nem hordoznak értéket, belső csúcsai pedig egy `b` típusú elemet viselnek cimkeként és két leágazásuk van). A program kódja:

```
module f
import StdDynamic, StdEnv, StdDynamicFileIO

:: Tree b = Node b (Tree b) (Tree b) | Leaf

Start world
    #! (ok,world)= writeDynamic ("C:\\function") dt world
    | not ok= abort "could not write dynamic"
    = (dt,world)
where
    dt = (dynamic count_leafs)
```

```
count_leafs :: (Tree Int) -> Int;
count_leafs tree= count tree 0;
where
    count :: (Tree Int) Int -> Int
    count Leaf n_leafs= n_leafs + 1
    count (Node _ left right) n_leafs
          = count left (count right n_leafs)
```

A **v** program feladata, hogy elmentsen *value* néven, egy `Tree Int` típusú fát:

```
module v
import StdDynamic, StdEnv, StdDynamicFileIO

:: Tree a = Node a (Tree a) (Tree a) | Leaf

Start world
    #! (ok,world)= writeDynamic ("C:\\value") dt world
    | not ok= abort "could not write dynamic"
    = (dt,world)
where
    dt = dynamic Node 98 (Node 2 (Node 1 Leaf Leaf) Leaf)
                         (Node 2 (Node 1 Leaf Leaf) Leaf)
```

Az elmentett fának hat levelét számolhatjuk össze.

A kritikus szemlélő észreveheti, hogy a `Tree` típusdefiníciójában típusváltozóként az `f` modulban `b`-t míg a `v` modulban `a`-t használtunk. Természetesen típusegyeztetéskor a típusdefiníciók szemantikája és nem az elnevezések számítanak a dinamikus szerkesztőprogramnak.

Az **apply** program feladata lesz az, hogy a kódállományként a háttértáron található *function* függvény segítségével megszámolja az ugyancsak ilyen formában tárolt *value* fa leveleit, és az `Int` típusú eredményt elmentse *result* néven.

```
module apply
import StdDynamic, StdEnv, StdDynamicFileIO
```

```
:: Tree a = Node a (Tree a) (Tree a) | Leaf

Start world
    # (ok,f,world)= readDynamic ("C:\\function") world
    | not ok= abort " could not read function"

    # (ok,v,world)= readDynamic ("C:\\value") world
    | not ok= abort " could not read value"

    # applied_dynamic= apply f v

    #! (ok2,world)= writeDynamic ("C:\\result") applied_dynamic world
    | not ok2= abort "could not write dynamic"

    = (world);
where
    apply (f :: a -> b) (v :: a)= dynamic f v
    apply _ _= abort "unmatched"
```

A **read_dynamic_apply** feladata csupán annyi, hogy a *result* kódállományban tárolt értéket beolvassa és kiírja a felhasználónak a képernyőre.

```
module read_dynamic_apply

import StdDynamic, StdEnv, StdDynamicFileIO

Start world
    # (ok,d,world)= readDynamic ("C:\\result") world
    | not ok= abort " could not read";

    = d
```

3.4.2. Fordítás, futtatás

A **v** szintaktikus ellenőrzése és fordítása után a szerkesztés nem történik meg, hanem létrejön a könyvtár- illetve típusfájl, és a program futtatásához szükséges parancssori hívást tartalmazó „bat” fájl. A futtatható fájl természetesen a felhasználó saját könyvtárában jelenik meg, a könyvtár- illetve típusfájlok

viszot egy külön rendszer könyvtárban. A létrejövő könyvtár- és típusfájl neve azonos, csupán kiterjesztésükben különböznek.

A „bat" fájl futtatásakor a dinamikus szerkesztőprogram –a korábban részletezett módon– gondoskodik a szerkesztésről és a futtatásról egyaránt.

A futtatás eredményeként kiíródik a háttértárra a program tulajdonképpeni célja a *value* érték (egy fa) kódállomány formájában a felhasználó által kívánt könyvtárba. A kódállomány *hivatkozás*t tartalmaz a programhoz tartozó könyvtár- illetve típusfájlra, hogy ha később ezt az értéket szeretnénk beolvasni és használni egy programban, akkor a szerkesztőprogram információt nyerjen belőlük.

Az **f** fordítása és futtatása hasonlóképpen működik.

Az **apply** fordítása is ugyanilyen, futtatásánál pedig a dinamikus szerkesztőnek be kell szerkesztenie a programba a *function* és a *value* kódállományokat. Ezt úgy teszi meg, hogy a rendszer tulajdonképpeni fő célját a típusellenőrzést elvégzi a hozzájuk tartozó típusfájlok segítségével, vagyis a *function* paraméterének és a *value* értéknek típusazonossága feltétele a folytatásnak.

A levélszámoló függvénynek a fára történő alkalmazása után a *result* kódállomány íródik ki, amely (mint mindig) hivatkozást tartalmaz az `apply` fordításakor létrejött könyvtár- és típusfájlokra.

A lustaság szerepe a példában

Mivel a fa levélszámának értéke nem kerül közvetlen felhasználásra az `apply` program folyamán –csupán el szeretnénk tárolni–, nem is értékelődik ki! A dinamikus szerkesztőprogram nem építi fel a *value*t és a *function*t reprezentáló gráfokat, és nem tölti be a hozzájuk tartozó könyvtárfájlokat sem. A típusfájljaikat természetesen betölti, hiszen ellenőriznie kell, hogy alkalmazható-e egyáltalán a függvény az értékre. A létrejött *result* tehát nem tartalmazhatja a *function* végrehajtásának eredményét, hiszen nem is tudjuk még az eredményt. Az elkészült kódállomány tehát mindössze a leírását tartalmazza annak, hogy ha később ki akarjuk értékelni, azt hogyan tehetjük meg. Márpedig a későbbi kiértékeléshez szükség lesz még a *function* és a *value* kódállományokra is, ezért a *result*-ban jelezzük ezeknek az elérési útját és nevét is. Az ilyen hivatkozástípust *lusta kódállomány hivatkozás*nak nevezzük.

Ebből a példából jól látszik, hogy a dinamikus szerkesztőprogram felépítése

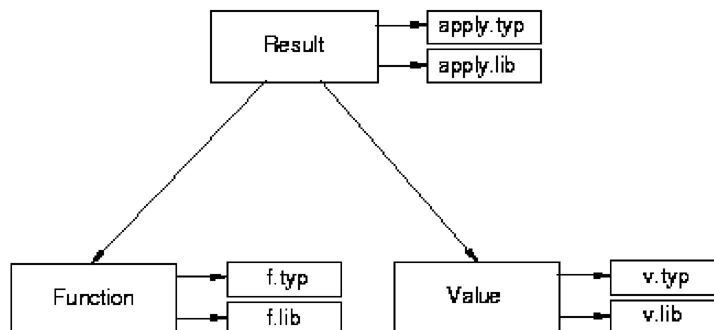
a fájlok szintjén is támogatja a nyelv alapvető tulajdonságát a lusta kiértékelést.

Amikor a **read_dynamic_apply** kerül fordításra, ugyanúgy működik, mint az eddigiek. Futtatásnál ki akarjuk írni a képernyőre a *result* kódállomány értékét, ehhez beszerkesztjük, (ellenőrzés történik a *result* és a `Start` függvény kimeneti paraméterének típusegyezőségéről) majd mivel a *result* nincs kiértékelve, be kell szerkesztenünk a *function*t és a *value*t is az értékének meghatározásához - méghozzá könyvtárfájlokkal együtt. Valójában itt a `read_dynamic_apply`-ban történik meg a függvény alkalmazása - a kifejezés kiértékelése, majd mindezek után a képernyőre való kiirás.

3.5. A továbbfejlesztéshez vezető kérdések, feladatok

Függőségek

Amint a példából látszik, a kódállományok mindegyike hivatkozik a hozzá tartozó típus- és könyvtárfájlokra, illetve amennyiben nem teljesen kiértékelt „lusta" kódállományról van szó –mint például a fenti példában szereplő *result*– hivatkozik még a kiértékeléséhez szükséges további kódállományokra. Ez a hivatkozási rendszer egy fa struktúrát alkot. A fenti példa *result* kódállományának hivatkozás-fája például az alábbi:



A könyvtár- és típusfájlok mindig a hivatkozásfa levelei –hiszek ők már nem hivatkoznak semmire– a hivatkozásfa gyökere és belső csúcsai pedig mindig kódállományok. Ciklikus hivatkozás sohasem alakulhat ki.

3.5.1. Nehézkes kezelhetőség

Mivel a kódállományokban és a „bat” fájlokban egyaránt hivatkozásként a hivatkozott fájlok neve és teljes elérési útja szerepel, rendkívül nehézkes illetve lehetetlen a felhasználónak lehetővé tenni, hogy kódállományokat más könyvtárba áthelyezzen, vagy akárcsak a kódállományt illetve az azt tartalmazó bármely szülőkönyvtárat átnevezzen. Ha töröl egy típus- vagy könyvtárfájlt esetleg kódállományt, az véglegesen használhatatlanná tehet egyéb kódállományokat, vagy dinamikus alkalmazásokat.

3.5.2. Szemétgyűjtés

A hivatkozásfa a felhasználónak csupán egy logikai struktúra, ezt a struktúrát nem látja, nehezen deríti ki, hogy melyik fájl melyik másikra hivatkozik – a könyvtár- és típusfájlok neve például teljesen automatikus, csupán az a lényeg, hogy az ilyen jellegű fájlok tárolásála szolgáló központi könyvtárban egyediek legyenek legyen pl.: „v_0c.lib” vagy „f_1c.typ”. Problémás azonban így a felesleges fájlok törlése, hiszen nem tudhatjuk, melyek valóban feleslegesek márpedig ezek terjedelmes méretűek lehetnek – gondoljunk csak a gépi kódot tartalmazó könyvtárfájlokra. Szükségessé válik tehát a szemétgyűjtő program megvalósítása.

3.5.3. Kódállományok másolása, küldése más számítógépre

Ugyancsak a függőségek miatt merül fel a kódállományok számítógépek közötti másolásának bonyolultsága is: ahhoz, hogy a célszámítógépen használható legyen a kódállomány, jelen kell lennie a belőle gyökerező függőségi fa összes fájljának, tehát azokat is át kellene másolni. Ennek elvégzéséhez egy úgynevezett „küldő” program elkészítése lenne célszerű, amelyik a kívánt kódállományt –összes függőségével együtt– átmásolja.

3.6. A továbbfejlesztés közben felmerült kérdések

3.6.1. Körülhatárolható „véges rendszer” problémája

Ahhoz, hogy a szemétgyűjtő eljárás működhessen, szükséges az, hogy valamiféle körülhatárolható célterülete legyen. Felmerült például korábban, hogy én mint felhasználó a saját számítógépemen jelen levő kódállományokat felajánlhassam interneten elérhetőként, azaz távoli gépen futó programok elér-

hessék és használhassák azt. A probléma azonban akkor merül fel, amikor a szemétgyűjtő eljárásom nem tudja így eldönteni, hogy egy adott kódállomány –habár én szükségtelennek tartom és a számítógépemen lévő más kódállományok nem hivatkoznak rá– törölhető-e vagy nem. Hiszen azt lehetetlen ellenőrizni, hogy van-e rá az internetről valahonnan hivatkozás.
A véges rendszer tehát nem más, mint egy olyan terület, amit ha a szemétgyűjtő átfésül és megvizsgálja az ott található hivatkozásokat, felelősségteljesen meg tudja állapítani valamiről, hogy már felesleges.

Ahhoz, hogy valamiféle tárolási hatékonyságot elérhessünk, bővíthetjük véges rendszerünk méretét. Négy-öt vagy akár száz –mindenesetre véges számú– gép megoszthatná és együtt tárolhatná kódállományait, a hozzájuk tartozó szemétgyűjtő pedig minden futtatásnál a mindezen gépekről hivatkozott fájlokat gyűjthetné össze és tekinthetné szükségesnek. Ezzel a megoldással viszont a futási idő hatékonyságát veszítjük el, hiszen egyrészt a szemétgyűjtő eljárás igényel több futási időt a hálózati forgalom miatt, másrészt minden egyes dinamikusan szerkesztett alkalmazás is, mivel le kell tölteni a távoli gépről a megfelelő kódállományokat. A sérülékenység is ezen megoldás ellen szól, hiszen a hálózati adatforgalom sérülése, vagy egy számítógép időszakos leállása lehetetlenné teheti nem csak a hulladékgyűjtést, hanem az onnan igényelt kódállományok beszerkesztését is az épp futtatni kívánt alkalmazásba.
Ilyen lehetőség elméleti megalapozásához és az ezt lehetővé tevő programok elkészítéséhez további kutatásra és programozási munkára lenne szükség.

A kutatás egyik eredményeként megvalósított szemétgyűjtő eljárás véges rendszerként egy számítógépet véve dolgozik. Hivatkozások más számítógépekről nem megengedettek.

3.6.2. Egyedi fájlnevek problémája

1. **Bonyolult küldés**

   A küldő program elkészítésénél merült fel a probléma, hogy a másik számítógépre küldendő kódállomány célgépen való zökkenőmentes működéséhez el kell helyezni az általa hivatkozott könyvtár- és típusfájlokat a megfelelő könyvtárba, hiszen az előző pontban leírt érvek miatt másik számítógépre való hivatkozások nem megengedettek. Ezeket a fájlokat nyilván olyan néven kell elhelyezni, ami az adott könyvtárra nézve egyedi.

   Tekintsük azt az esetet, amikor a fenti példában szereplő *value* kódál-

lományt egy távoli gépre szeretnénk küldeni. Ekkor vele kell küldenünk a v_0c.lib és v_0c.typ fájlokat is, és át kell neveznünk őket célgépen egyedire. A *value* kódállományban a hivatkozások azonban a régi fájlnevekre sőt könyvtárstruktúrára vonatkoznak, tehát azokat is át kell írni.

   Természetesen ez a bonyolultság nem a fenti egyszerű példa esetén számottevő, hanem akkor növekszik azzá, amikor igen komplex hivatkozási fát kell átküldeni.

2. **A küldés hatékonysága**
   A küldő program sokkal hatékonyabb lehetne, ha nem másolná át a már úgyis jelenlévő fájlokat. Ha a fenti példában szereplő *value* kódállomány átküldése után szeretném az *apply*t elküldeni (aminek a hivatkozási fájában a *value* megtalálható), felesleges mégegyszer elküldenem a *value*t és a hozzátartozó könyvtár- illetve típusfájlt, márpedig annak kiderítése, hogy már jelen vannak az adott gépen -csupán lokálisan egyedi fájlnevekkel- nem lehetséges.

   Ez különösen akkor számítana, ha sikerülne például a 3.1 alatt említett első lehetőséget kihasználni, vagyis nagyobb könyvtári egységeket (pl. StdEnv) csupán egyszer –kódállomány formájában– tárolni a gépeken, –a minden alkalmazásba való statikus idejű beszerkesztés helyett– ezzel növelve az adott gép tárolási hatékonyságát. Ilyen esetben nyilván egyik kódállománnyal sem kéne átküldeni előre definiált könyvtári egységeket.

3. **Manipulált (sérült) fájlok problémája**
   A felhasználó rossz szándékkal megteheti, hogy a hivatkozott fájlok tartalmát manipulálja, például a típusfájlokban található típusinformációkat módosítja. Ez ahhoz vezethet, hogy a dinamikus szerkesztőprogram megpróbál nem megfelelő típusú elemet beszerkeszteni, mert ahol a típusinformációt ellenőrzi ott mindent rendben talál. Az ilyen manipuláció következménye előre megjósolhatatlan.

Az első két problémára közös megoldásként kínálkozik a következő: valamiképpen olyan fájlneveket kellene mentéskor adni a típus- és könyvtárfájloknak illetve a kódállományoknak, hogy azok az *egész világon* egyediek legyenek. A megoldás első hangzásra utópisztikusnak tűnik, de komolyan elgondolkodtunk, hogyan is lehetne ezt megvalósítani:

Kínálkozó megoldások

1. **Központi szerver**
   Elvileg lehetséges az, hogy amikor a fordító létre szeretne hozni egy egyedi nevű típus- illetve könyvtárfájlt, egy központi szervertől kérjen egy új „azonosítót”, ami aztán az egész világon egyedi lesz.
   előnye:

   (a) 100%-ig egyedi lesz az előállított azonosító.

   hátrányai:

   (a) Nem minden gép rendelkezik internet csatlakozással, hogy kapcsolatot létesítsen a központi géppel.
   (b) Igen érzékeny a rendszer az adatforgalom vagy a központi szerver sérülésére.

2. **Hely és idő**
   Tudjuk, hogy a hely és az idő pontosan meghatároz egy eseményt. Belefűzhetnénk tehát ezen analógia szerint a fájlnévbe a létrehozó számítógép IP címét és a létrehozás dátumát illetve óraidejét.
   előnyei:

   (a) Közel 100%-ig egyedi lesz az előállított azonosító.
   (b) Nem érzékeny a rendszer hálózati problémákra.

   hátrányai:

   (a) Könnyen manipulálható az előállított azonosító (pl. óraidő).
   (b) Még mindig elvárja az internet csatlakozást.

3. **Message digest**
   Egyre elterjedtebbek a világon a „message digest” vagyis *üzenet hitelesítő* algoritmusok. Ezek arra jók, hogy hash függvények segítségével nehezen manipulálható módon ellenőrző összegeket készítsenek szövegekhez illetve fájlokhoz. Erősségük az, hogy rendkívül bonyolult előre meghatározott azonosítóra illeszkedő fájlt készíteni, illetve két ugyanolyan azonosítójú fájlt előállítani.

   Általában digitális hitelesítésre használják ezeket az algoritmusokat, de lehetőségünk lenne ilyen algoritmusokat felhasználni a jelen probléma

megoldására is, méghozzá úgy, hogy a fájlnevekbe belefűzzük a fájl tartalmából kiszámolt ellenőrző összeget.
előnyei:

(a) Nem szükséges internet csatlakozás az azonosító létrehozásához.
(b) Az előállított azonosító igen nehezen manipulálható
(c) Az adott azonosító információt hordoz a fájl belső tartalmáról is, tehát pl. a különböző helyen és időben fordított de azonos verziójú StdEnv kódállomány azonosítója ugyanaz lesz, ami a küldés és tárolás hatékonyságának fejleszésében nagy szerepet játszhat, a verzióhűség megtartása mellett.
(d) A fájlnevek „message digest" alapon történő előállítása lehetővé tenné a dinamikus szerkesztőprogramnak, hogy azt is ellenőrizze, vajon sérült-e a fájl tartalma a létrehozása óta, tehát megoldaná a manipulált fájlok problémáját.

hátránya:

(a) Bármikor megvan az elvi lehetősége nem egyedi azonosító létrejöttének, tehát ez csupán tapasztalati úton igazolható, „nagy valószínűséggel" egyedi azonosító.

Konklúzió

Az egyedi fájlnevek megvalósítására szükség van, az egyszerű és tárolási hatékonyságot szem előtt tartó küldő program elkészíthetőségéhez. A megoldások közül a harmadik a legelőnyösebb, hiszen lehetővé válik a belső tartalmat is tükröző, nehezen manipulálható azonosító, és a két különböző helyen előállított de máskülönben azonos kódállományok azonosságának felismerése amiket a másik két megoldás nem tesz lehetővé. Hátránya: a nem bizonyíthatóan egyediség kétségtelenül fennáll. A döntés előtt teszteltem jópár lehetőséget (pl. 1 bit-ben különböző fájlok esete) és másokkal egyetértésben mégis ezen megoldás mellett döntöttünk: választásunk az *MD5* nevű Ronald L. Rivest által kifejlesztett algoritmusra esett[6].

3.7. A rendszeren végrehajtott módosítások

A rendszer működésében változtatásokat hatjottunk végre, ezzel a megvalósítás szintjén is biztonságosabbá vált az elméletileg igen átgondolt és nagy meg-

bízhatóságot ígérő elképzelés.

3.7.1. Új fájlnévmeghatározás

Az *MD5* algoritmusnak elkészítettem egy hatékony Clean nyelvű implementációját, melynek segítségével lehetséges az egyedi nevek „message digest” alapon történő meghatározása. Ennek használatával átírtam azt a programkódot, mely a dinamikusan szerkesztendő program fordítása után a létrejövő könyvtár- és típusfájlokat elmenti: jelenleg úgy ad nevet nekik, hogy kiszámítja mindkét fájl tartalmának az *MD5* ellenőrzőösszegét és –minthogy szorosan összetartoznak– a két fájlnév továbbra is ugyanaz: <könyvtárfájl ellenőrzőösszege>_<típusfájl ellenőrző összege> és az ehhez járuló „lib” illetve „typ” kiterjesztés.

Az *MD5* azonosítók kiszámítása még elmentés előtt történik, hogy létrehozhassuk a megfelelő nevű fájlokat. Ennek a megoldásnak igen nagy a memóriaigénye – hiszen a leendő fájlok tartalmát egy az egyben tárolnunk kell a memóriában mielőtt a kiírást megkezdhetnénk.
Kisebb memóriaigényű lenne az ideiglenes fájlnéven való tárolás és a később kiszámolt ellenőrzőösszegek szerinti fájlátnevezés, ez viszont könnyebb manipulálhatósághoz vezetne.

3.7.2. A fájlok elrejtése

A könnyebb kezelhetőség és a megbízhatóság növelése érdekében létrehoztunk két „adatbázis” könyvtárat a dinamikus szerkesztőprogram számára, amelyek egyikében tárolhatja az összes létrehozott kódállományt, a másikban pedig a dinamikus alkalmazásokhoz tartozó könyvtár- és típusfájlokat, mégpedig olyan módon, hogy azokhoz a felhasználó garantáltan nem nyúl hozzá. Ezekbe a könyvárakba fájlokat másolni illetve belőlük törölni csak magának a szerkesztőprogramnak, a szemétgyűjtő- valamint a küldő-programnak van joga.

Az dinamikus alkalmazások fordítása után könyvtár- és típusfájljaik (az előbb taglalt néven) az arra szolgáló könyvtárba kerülnek. A felhasználó könyvárába jön létre továbbra is a „bat” kiterjesztésű fájl a szerkesztő meghívásával történő programfuttatásához, de nem tartalmaz többé teljes elérési utat hivatkozásként, csupán az ellenőrzőösszegből létrejött fájlnevet.

Amikor egy program kódállományt szeretne kiírni a háttértárra (például a v a *value*t), a kódállomány egy arra szolgáló központi könyvtárban jön létre

a felhasználó számára láthatatlan módon, neve az *MD5* algoritmusból nyert ellenőrzőösszeg lesz „*dynsys*" kiterjesztéssel[5], a felhasználó könyvtárába pedig létrejön egy mutató fájl[6] *value.dyn* néven. Amikor később az apply program be szeretné szerkeszteni a *value* kódállományt, a szerkesztő program feloldja ezt a hivatkozást és az „adatbázis" könyvtárból veszi a megfelelő rendszerhez tartozó kódállományt.

A kódállományokban található hivatkozások (akár a könyvtár- és típusfájlokra akár más kódállományokra) nem tartalmazák már az elérési utakat, csupán az MD5 alapú *fájlneveket*.

3.7.3. Szemétgyűjtő

Sikerült megvalósítanom a rendszer szemétgyűjtő programját. A program első lépésként egy megadott gyökérkönyvtárból[7] kiindulva összegyűjti a *dyn* és a *bat* kiterjesztésű fájlokat – az utóbbiak esetén természetesen ellenőrzi, hogy valóban dinamikus alkalmazásról van-e szó. Az összegyűjtött fájloknak felépíti a hivatkozási-fáját, így kapja meg a rendszer biztonságos működéséhez az adatbázis könyvtárakban megtartandó fájlok listáját. A harmadik ütemben összehasonlítja a kapott listát a tárolt fájlok listájával, majd a nem hivatkozott könyvtár- és típusfájlokat illetve rendszerhez tartozó kódállományokat törli.

A gyökérkönyvtár megadásának lehetősége alapvetően hatékonysági okokból merült fel. Természetesen ez alapértelmezésben lehet „C:\", a futási idő azonban csökkenthető ha például tudjuk, hogy minden ilyen dinamikus alkalmazásnak és kódállomány mutatónak valahol a „C:\Clean" alatt kell lennie mert csak ott foglalkozunk Clean fejleszésekkel.

Meggondolandó tényező még az egyes operációs rendszerekben megvalósított szeméttároló kérdése. Erre azért kell külön figyelni, mert ha a felhasználó letöröl egy kódállomány mutatót vagy egy dinamikus alkalmazást, az magával vonhatja –természetesen, hiszen ez a cél,– hogy a szemétgyűjtő letöröljön hozzá tartozó fájlokat az adatbázis könyvtárakból. Márpedig a felhasználó által törölt fájl még a szeméttárolóban lehet. Ha aztán a felhasználó a letörölt fájt visszahelyezi a helyére, a rendszer megbízhatósága felborul. A

[5] Ezt a fájlt a továbbiakban *rendszerhez tartozó kódállomány*nak nevezem

[6] A mutató fájl itt nem valódi *link fájl*ként értendő, csupán egy szövegfájlról van szó, amely tartalmazza a valódi kódállomány nevét. A fájlt a továbbiakban *kódállomány mutató*ként emlegetem.

[7] A gyökérkönyvtár nevét egy erre szolgáló konfigurációs fájlból olvassa ki.

dolog megoldása egyrészről az lehet, hogy a szemétgyűjtőnek elérhetővé teszszük a szeméttároló könyvtárát is, hogy onnan is gyűjtsön hivatkozásokat, másrészről kibővíthetjük a szemétgyűjtő program funkcióját azzal a feladattal, hogy a szeméttárolóban talált dinamikus alkalmazásokat és kódállomány mutatókat *véglegesen* letörölje.

3.7.4. Küldő

Várnagy Zoltán valósította meg a rendszerhez tartozó másik kisegítő programot, amely a kódállományok távoli gépre történő átküldését oldja meg hálózaton keresztül.

Mivel a globálisan egyedi nevek módot biztosítanak a fájlok egyértelmű azonosítására, a két gép előzetes kommunikációval kiválasztja az átküldendő kódállomány hivatkozási fájából a célgépen már tárolt illetve nem tárolt fájlokat, majd az utóbbiakat átküldi.

Mivel a fájlnevek egyedisége miatt változtatásukra nincs szükség és a könyvtárstruktúra sincs eltárolva a hivatkozásokban, nem is kell módosítani ezeket a hivatkozásokat.

3.8. A változtatásokkal elért eredmények

Az új működési elv a dinamikus rendszer és a felhasználó háttértárolón való munkaterületének különválasztásával egyszerre teszi lehetővé a felhasználónak, hogy úgy bánjon a fájljaival ahogyan szeretne –lehetővé válik számára a kódállományok biztonságos módon való törlése, átnevezése, áthelyezése [8]– és a dinamikus rendszernek, hogy a megbízható működést az adatbázis könyvtárak folyamatos karbantartásával biztosítsa.

Az egyedi fájlnevek bevezetése a küldő eljárás hatékony és egyszerű megvalósítására adott módot — ahol a hatékonyság tárolási –a célgépen már meglévő fájlokat nem duplikáljuk [9]– és futási idejű –csak a hiányzó fájlok vesznek részt a hálózati forgalomban– értelemben egyaránt értendő, az egyszerűség pedig a hivatkozásátírások szükségtelenségére utal.

A fájlnevek konstrukciójából adódóan lehetővé vált a dinamikus szerkesztőprogramnak a beszerkesztendő fájlok eredetiségének ellenőrzése, amely ere-

---

[8] természetesen a felhasználó számára a kódállományok a továbbiakban csupán a kódállomány mutatókat jelentik

[9] például azonos verziójú *StdEnv*

detiség a fájl tartalmának manipulálásától és sérülésétől egyaránt megszűnhet.

A szemétgyűjtő program megírásával megoldódott a feleslegesen tárolt fájlok kérdése, a küldő program létrejötte pedig biztosítja, hogy még kódállományok más gépre történő másoláskor se kelljen a felhasználónak tisztában lennie a rendszer belső működésével.

Konklúzió

A továbbfejlesztés eredményeképpen a programozó számára a mobil kódot tartalmazó fájlok korábbi nehézkes kezelhetősége megszűnt, ezen fájlokat bizonyos absztrakciós szintről látja. Az elmentett állományok felépítése és további implementációs kérdések elrejtésre kerültek, a rendszer biztonságos működéséről pedig programok gondoskodnak.

Ezzel a fejlesztéssel a dinamikus típusrendszer előrelépett abba az irányba, hogy a jelenlegi kísérleti verzióból kényelmesen használható és biztonságos működésű programozási eszköz válhassson.

A dolgozat létrejötte nemcsak az én érdemem. A szemétgyűjtő eljárás elkészítése és a dinamikus típusrendszer implementációjának módosítása közben folyamatosan egyeztettünk Rinus Plasmeijer ottani professzor úrral aki a Clean csoport vezetője, Martijn Vervoort-tal aki a dinamikus szerkesztőprogram megalkotója és folyamatos fejlesztője, valamint Várnagy Zoltánnal aki Erasmus hallgatóként –csakúgy mint jómagam– kint töltötte a szemesztert és megalkotta a kódállomány küldőprogramot. A közös munka eredményeiből a Nijmegeni Egyetemen is készült cikk, mely az irodalomjegyzékben is megtatlálható[4].

Irodalomjegyzék

[1] P. Koopman, R. Plasmeijer, M. van Eekelen, S. Smetsers: *Functional Programming in Clean (Draft)*; September 10, 2001. http://www.cs.kun.nl/~clean/contents/Clean_Book/clean_book.html

[2] M. van Eekelen, S. Smetsers, R. Plasmeijer: *Graph Rewriting Semantics for Functional Programming Languages*; In Proc. of CSL '96, Fifth Annual conference of the European Association for Computer Science Logic (EACSL), Utrecht, Springer Verlag, LNCS 1258, pp. 106-128.

[3] M. Pil: *Dynamic Types and Type Dependent Functions*; in Proc. of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, 1998, pp. 65-84.

[4] M. Vervoort, R. Plasmeijer: *Lazy Dynamic Input/Output in the lazy functional language Clean*; Post-workshop submission: 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Madrid, September 16-18, 2002. ftp://ftp.cs.kun.nl/pub/Clean/papers/2002/verm2002-LazyDynamicIO2.pdf

[5] A. van Weelden, R. Plasmeijer: *Towards a Strongly Typed Functional Operating System*; 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Madrid, September 16-18, 2002.

[6] R. L. Rivest: *The Md5 Message-Digest Algorithm*; MIT Laboratory for Computer Science and RSA Data Security, Inc. April, 1992 (RFC 1321)

[7] Nyékyné G. J. szerk., Nyékyné G.J.-Horváth Z. és mások: *Programozási nyelvek összehasonlító elemzése*; Kiskapu Kiadó, Budapest, 2002.
Fejezet (Horváth Z.):*Funkcionális programozási nyelvek elemei*, 56 oldal, megjelenés: 2002. dec.