EÖTVÖS LORÁND TUDOMÁNYEGYETEM
TERMÉSZETTUDOMÁNYI KAR

Funkcionális programok erőforrásigénye

Horváth Gyula
Témavezető: Horváth Zoltán

2003. június 19.

Készült az OTKA
TO37742 projekt keretében

Tartalomjegyzék

1. Történeti kitekintés **5**

2. A funkcionális számításmodell **7**

2.1. λ-kalkulus . . . 7

2.1.1. λ-kifejezések . . . 7

2.2. TRS rendszerek . . . 10

2.2.1. TRS rendszerek fogalma . . . 10

2.2.2. TRS rendszerek felépítése . . . 10

2.2.3. TRS rendszerek működése . . . 13

2.3. GRS rendszerek . . . 16

2.3.1. GRS rendszerek fogalma . . . 16

2.3.2. GRS rendszerek felépítése . . . 17

2.3.3. GRS rendszerek működése . . . 17

2.4. A Clean nyelv . . . 20

2.4.1. A Clean nyelvű kódok . . . 20

2.4.2. A Clean típusrendszere . . . 22

2.4.3. A kiértékelési rendszer . . . 23

2.5. Az ABC gép . . . 24

2.5.1. Az ABC gép karakterisztikája . . . 24

3. Erőforrások használata **29**

3.1. A programok által igénybe vett erőforrások osztályozása . . . 29

3.2. Erőforrásigény vizsgálata . . . 29

3.3. Kiértékelésszám meghatározása . . . 30

3.4. Konkrét esetosztályok elemzése . . . 30

3.4.1. Mérési környezet által vizsgált erőforrások . . . 30

4. Vizsgált speciális esetosztályok **33**

4.1. Lineáris struktúrák . . . 33

4.1.1. Vermek és sorok . . . 33

4.1.2. Rendező algoritmusok . . . 35

4.2. Fa jellegű struktúrák . . . 37

4.3. Számításigényes matematikai feladatok . . . 37

4.3.1. Clean környezet diagnosztikájával mért adatok . . . 37

4.3.2. Időprofil elemzés 38

5. Programkódok elemzése 41

5.1. Felhasznált erőforrástípusok egymásra hatása 41

5.1.1. Kiértékelésszám hatása a memória allokációra 41

5.1.2. Kiértékelésszám hatása a futási időre 42

5.1.3. Allokált memória mennyiségének hatása a futási időre . 43

5.2. Köztes kódok vizsgálata 43

5.2.1. A köztes kód (ABC) viszonya a forráskódhoz 44

5.2.2. Implementációs ”stílusok” hatása a köztes kódra 44

5.3. Várható erőforrásigény becslése 45

5.3.1. Becslések alapja 45

5.3.2. Becslések módszere 46

A. Forrásként használt kódrészletek 49

A.1. Rendező algoritmusok 49

A.2. ABC kódok 50

1. fejezet

A funkcionális nyelvek történeti fejlődése

Az 50-es évektől kezdve az informatikai iparban kettős forradalom zajlott le. A hardverek árának csökkenő tendenciája mellett, teljesitményük folyamatos növekedése volt megfigyelhető. Ez a folyamat a 60-as évek közepére azt eredményezte, hogy a számítógéppel megoldható feladatok halmaza jelentősen megnőtt. A feladatok sokféleségével párhuzamosan a feladatok átlagos bonyolultsága is rohamosan nőtt. A szoftver készítése, és különösen a megbízható szoftverek iránti igény, új tervezési, reprezentációs és implementációs technikák kifejlesztését indukálta.

Ennek megfelelően az új technológiák fejlesztésekor, két fő irány különböztethető meg.

- Hogyan lehet olcsón, nagy méretű szoftver rendszereket létrehozni, melyek ráadásul kellően megbízhatóak és felhasználóbarátok.
- Hogyan lehet a szoftverek hatékonyságát olcsón növelni.

A kutatók egy része az imperatív nyelvek továbbfejlesztésének szükségességével érvelt, és ennek az ágnak az eredményeképpen létejöttek a funkcionális stílusú, illetve az objektum orientált nyelvek. E nyelvek kifejlesztése sok eredményt hoztott, de a felhasználók gondolkodásmódja továbbra is, az imperatív mód maradt.

Más kutatók új programozási paradigmák kidolgozásán dolgoztak. 1978-ban John Backus a számítási problémák egy új megoldási módjával állt elő. Az általa javasolt funkcionális programokban a függvények visszatérési értéke determinisztikus, és csak a paraméterek aktuális értékétől függött. A függvényeknek nem voltak külső kapcsolataik, így nem lehettek mellékhatásaik sem. Azaz indiferens a függvény értékének szempontjából, hogy ki, mikor és milyen környezetben hívta meg. Végeredményében sokkal egyszerűbb egy mellékhatásoktól mentes függvény "jóság"-át definiálni és bizonyítani, mint egy klasszikus imperatív függvényét.

A korai funkcionális nyelvek mindkét – a programozási nyelvek elterjedését gátló – tényező (megfelelő fordítóprogramok és képzett szakemberek hiánya) következtében megmaradtak az egyetemi kutatóközpontok falain belül. Az utóbbi két évtized eredményeinek köszönhetően, napjainkra a funkcionális nyelvek fordítóprogramjai, a legtöbb feladatosztályban egyenértékű tárgykódot képesek előállítani az imperatív nyelvek fordítóival. Ebben a fejlődésben jelentős szerepe van a funkcionális programok kiértékelés-optimalizálásának, illetve a speciális köztes kódok használatának.

2. fejezet

A funkcionális számításmodell

A funkcionális programok működési elve több absztrakciós szintre épül. Az alábbi fejezet célja, hogy rövid áttekintést adjon a lehetőségekről, valamint az egyes szintek bevezetésének céljairól. A fejezet első három alfejezete az elméleti funkcionális modellel foglalkozik, az utolsó kettő pedig a Clean nyelven keresztül egy gyakorlati megvalósítást tár az olvasó elé.

2.1. λ-kalkulus

A λ-kalkulust 1932-33-ban Church definiálta azzal a céllal, hogy függvények kiszámíthatóságát vizsgálja. A kalkulus a korábbi halmazelméleti függvénymodellhez képest jelentős előrelépésnek tekinthető, mivel kezelhető vele a rekurzió problémája.

Bár vannak akik, a λ-kalkulusra [3], mint az első funkcionális nyelvre tekintenek, a mai funkcionális nyelvek mellett, inkább matematikai kalkulusként használják. Ezzel együtt igaz, hogy a modern funkcionális nyelven írt programok áttranszformálhatóak λ-kalkulusba. Lehetséges - bár nem egyszerűen - úgynevezett λ-gépet létrehozni.

2.1.1. λ-kifejezések

A λ-kalkulusban λ-kifejezéseket ***[λ-expressions]*** lehet leírni. A továbbiakban a kifejezések halmazát átíró, vagy redukciós szabályoknak ***[rewrite/reduction rules]*** nevezzük. Ezek a kifejezések azt írják le, hogy hogyan lehet λ-kifejezéseket más, velük ekvivalens λ-kifejezésekké átírni.

λ-kifejezés	=	változó | λ-absztrakció | applikáció | konstans
λ-absztrakció	=	'(' 'λ' változó '.' λ-kifejezés ')'
applikáció	=	'(' λ-kifejezés λ-kifejezés ')'

Megegyezés szerint a változók kis betűvel, míg a konstansok nagy betűvel, vagy valamilyen speciális jellel kezdődnek (pl.: számjegy, @, $*$, stb.). Az

absztrakciók jobb -, míg az applikációk bal asszociatívok. Az applikáció magasabb precedenciájú, mint az absztrakció.

λ-absztrakció

λ-absztrakcióval név nélküli függvényeket lehet lérehozni, melyben a változót formális argumentumnak, a λ-kifejezést pedig törzsnek nevezzük.

A λ-absztrakciókat a következő képpen olvassuk:

λ	v	.	kifejezés
↑	↑	↑	↑
Egy függvénye	a v változónak,	amelyik	ezzel az értékkel "tér vissza".

A λ-absztrakciókban csak egy-argumentumos függvénynek definiálhatóak. A több-argumentumos függvények, egymásba ágyazott (magasabb rendű ***[higher order]***) függvényekkel szimulálhatóak.

Függvény applikáció

A funkcionális nyelvekhez hasonlóan, az applikációnak alapvető szerepe van a λ-kalkulusban is. Jelölésére az egymás mellé írt kifejezéseket használjuk.

Az applikációt a következő képpen olvassuk:

f	a
↑	↑
Az f-re alkalmazzuk	a-t.

Ha f egy függvény (λ-absztrakció), akkor a-t a függvény aktuális argumentumának nevezzük.

Konstansok

Az eredeti λ-kalkulus modellben, a tiszta λ-kalkulusban [3], nem szerepelnek konstansok. A gyorsabb - kevesebb redukciós lépést igénylő - számítások érdekében, érdemes a modellt kibővíteni külsőleg definiált függvényekkel és konstansokkal, melyek az állandó értékeket (pl.: számok) reprezentálják. Az ilyen konstansokhoz tartozó külsőleg definiált műveleteket, δ-szabályoknak ***[δ-rules]*** nevezzük.

Kötött és szabad változók

A λ-kifejezések tartalmazhatnak változókat, melyeket általában egy-karakteres azonosítók (x, y, z, a, b, ...) jelölnek. Az egy kifejezéshez tartozó változók halmaza két diszjunkt halmazból áll, a kötött ***[bound]*** és a szabad ***[free]***

változók halmazából.

Szabad változók a következők:

- az x kifejezésben, az x változó
- az EF kifejezésben az x változó, ha x szabad változó E-, vagy F-részkifejezésben
- a $\lambda y.E$ kifejezésben az x változó, ha $x \neq y$ és x szabad változó E részkifejezésben

Kötött változók - pedig - a következők:

- az EF kifejezésben az x változó, ha x kötött változó E-, és F-részkifejezésben
- a $\lambda y.E$ kifejezésben az x változó, ha $x = y$ és x szabad változó E-ben, vagy, ha x kötött változó E részkifejezésben

Átalakítások

A λ-kifejezéseken három különböző átalakítást lehet tenni. Ezek az következők:

- α-konverzió
- β-redukció
- δ-redukció

Az α-**konverzió**val a λ-kifejezés változóit nevezhetjük át, a következő módon:

$$\lambda x.E =_\alpha \lambda y.E[x := y], \quad \text{ahol}$$

az $E[x := y]$ jelöli, az E-ben szabad x változók y-ra cserélését.

A β-**redukció**n a λ-kifejezés egy függvényének actuális argumentummal történő felhasználását értjük, a következő módon:

$$(\lambda x.E)A \to_\beta E[x := A], \quad \text{ahol}$$

az $E[x := A]$ jelöli, az E-ben szabad x változók A-ra cserélését. A kötött és szabad változók között létrejöhet a névkonfliktus.

A δ-**redukció**nak a λ-szabályok végrehajtását nevezzük.

2.2. TRS rendszerek

A kifejezés átíró rendszererek (röviden: TRS) ***[Term rewriting system]*** [10], mint programozási paradigma elméleti áttekintése Huet és Oppen (1980), valamint Klop (1992) munkáiból ismerhető meg.

2.2.1. TRS rendszerek fogalma

A TRS rendszeren az átalakító szabályok ***[rewrite rules]*** halmazát értjük. Az átíró szabályok sok hasonlóságot mutatnak a függvényekkel, viszont a funkcionális nyelvekkel ellentétben, a szabályok - itt - csak globálisan definiálhatók. A TRS beli szabályok rendeltetése, hogy átalakítsanak különböző kifejezéseket ***[term]***, más kifejezésekké. Maguk a funkcionális nyelven írt programok is ilyen kifejezések, melyek a program futása során kiértékelésre kerülnek, a kezdeti kifejezés sorozatos átalakításával. A TRS rendszerekben a kulcselemek szintaktikai egyezése (minta illesztéssel ellenőrizve) az alapja a redukciónak. Ha a szabály bal oldala megfelel a kifejezésnek, akkor a kifejezés átalakítható a jobb oldalnak megfelelően.

Egyesek a mintaillesztéses kombinotarikus logika kiterjesztéseként tekintenek a TRS rendszerekre. Továbbá kapcsolatba hozhatók a λ-kalkulussal. A TRS rendszerek kifejezőereje nagyobb mint a λ-kalkulusnak. Például a TRS rendszerek nem feltétlenül determinisztikusak, ami sem a λ-kalkulus, sem a kombinotarikus logika keretei között nem megengedett. A TRS rendszerek egy részhalmazát alkotják az úgynevezett gráf átíró rendszerereknek (röviden: GRS) ***[Graph rewriting system]***, amelyről a következő fejezetben lesz szó.

Az a legfontosabb különbség a TRS rendszerek és a funkcionális nyelvek között, hogy a TRS alap szemantikájában nincs rendezés definiálva az újraírási szabályok között, azaz minden szabály aktivizálódásának egyforma a valószínűsége. Továbbá nincs semmiféle stratégia, amely meghatározná a redex-ek (átírható része kifejezésnek) átalakításának sorrendjét.

A funkcionális nyelvek könnyen áttranszformálhatóak a TRS rendszerek különböző változataiba, implementációi általában sokkal közelebb állnak a TRS rendszerekhez, mint a λ-kalkulushoz. A TRS rendszeren alapuló implementációk kevesebb problémát vetnek fel, mint a λ-kalkuluson alapulók. Ezen kívül a fenti implementációk kellően hatékonyak lehetnek.

2.2.2. TRS rendszerek felépítése

A kifejezés átíró rendszerek az átíró szabályok halmazából állnak. Ezek az átíró szabályok meghatározzák azt, hogy miként lehet a kifejezéseket átalakítani (redukálni, átírni) más kifejezésekké.

Kifejezések

A kifejezések ***[terms]*** [10] halmaza a következő képpen definiálható:

$$Term \quad = \quad Variable \quad | \quad Constant \quad | \quad '('Constant \ \{Term\}')';$$

Ahol a { } jelepáron jelentése: nulla vagy több. Azaz $\{Term\}$ tetszőleges számú kifesezést ***[Term]***-t jelent. A változók ***[Variables]*** kötelezően kis betűvel kezdődnek, a konstansok ***[Constants]*** pedig nagy betűvel, vagy valamilyen speciális karakterrel (pl.: számjegy, @, $*$, stb.). A legkülső { } párt általában elhagyjuk. A konstans szimbólumot követő - nulla vagy több - kifejezést, összefoglaló néven argumentumoknak, számukat, aritásnak ***[arity]*** nevezzük. Azokat a kifejezéseket, melyek nem tartalmaznak vátozókat, zárt kifejezésnek ***[closed term]***, a többit parametrikus kifejezésnek ***[open term]*** nevezzük.

Példák kifejezésekre:

x	$Reverse\ (Cons\ 2\ Nil)$
1000	$Select\ 3\ (Array\ 1\ 2\ 3\ 4\ 5\ 6)$
$Fac\ x$	$Sqrt\ (-\ (*\ b\ b)\ (*\ 4\ (*\ a\ c)))$
Fac 1000	$Ap\ (Ap\ x\ z)\ (Ap\ y\ z)$

Egy τ' akkor és csak akkor részkifejezése ***[subterm]*** a τ kifejezésnek, ha τ lexikálisan tartalmazza a τ'-t, és τ' a τ konstrukciójában, mint kifejezés megjelenik. A τ'-t valódi részkifejezésnek nevezzük, ha τ' részkifejezése τ-nak, de nem egyezik meg vele. A TRS beli részkifejezés ***[subterm]*** nem azonos a Miranda nyelv, illetve a λ-kalkulus beli részkifejezéssel ***[subexpression]***. Ez utóbbi a hiányzó implicit applikációra vonatkozik. Az implicit applikációk megengedik bármely két kifejezés ***[expression]*** egymás mellé helyezéses összekötését, és így újabb kifejezés kialakítását.

Példák az $Ap\ (Ap\ x\ K)\ K$ kifejezés részkifejezéseire:

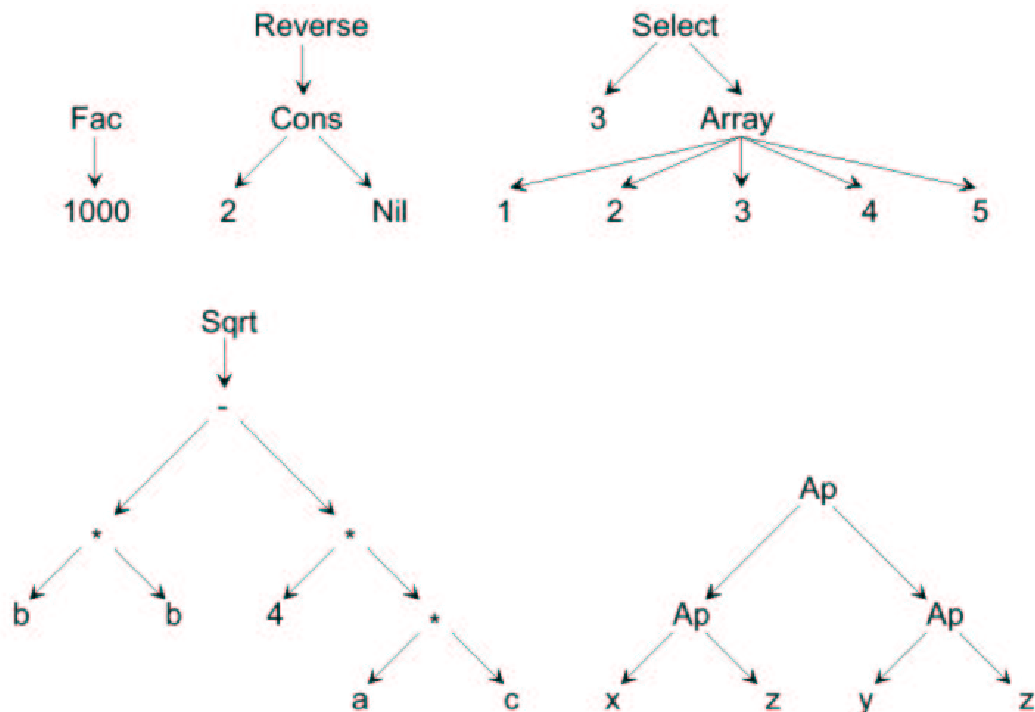
$Ap\ (Ap\ x\ K)\ K$	x
$Ap\ x\ K$	K (kétszeresen is)

Továbbá néhány ellenpélda, melyek $Ap\ (Ap\ x\ K)\ K$ kifejezésnek nem részkifejezései:

Ap	$x\ K$
$Ap\ (Ap\ x\ K)$	$(Ap\ x\ K)\ K$
$Ap\ x$	

Kifejezések reprezentálása fával

A kifejezések reprezentálhatók fákkal [10].



Ezekkel a fákkal a legtöbb esetben érthetőbben reprezentálhatóak a különböző kifejezések, mint a definicójuk szerinti alakkal. A fa csúcsai a konstans szimbólumokat jelenítik meg, míg az adott csúcsból kiinduló irányított szakaszok, az ehhez a csúcshoz tartozó konstans argumentumainak felelnek meg. Azaz a konstansok aritása megegyezik a hozzá tartozó csúcsból kiinduló élek számával. A fa egy részfája, ami tartalmazza az – éleken keresztül – általa elérhető egyéb csúcsokat, megfelel a részkifejezésnek.

Kifejezések átírásának rendszere

A TRS rendszerekben $\alpha_i \rightarrow \beta_i$ alakú átírási (redukciós, átalakítási) szabályok találhatók, ahol α_i-k,β_i-k kifejezések, továbbá minden β_i-beli változó megtalálható α_i-ben is, és egyetlen α_i sem áll, egyetlen változóból.

TRS rendszer szintaxisa:

TRS	=	{TRS-szabály}
TRS-szabály	=	Szabály-bal-oldala $' \rightarrow '$ Szabály-jobb-oldala
Szabály-bal-oldala	=	Konstans {Kifejezés}
Szabály-jobb-oldala	=	Kifejezés

A TRS rendszer beli konstansok két csoportba sorolhatók. Minden konstanst, amely előfordul valamely szabály bal oldalának legbaloldalibb pozí-

cióján, függvénynek nevezünk. Ezek alapján a fenti szabályt a függvény szabályának, továbbá a megfelelő szimbólumot definiciós előfordulásnak nevezhetjük. A többi konstanst összefoglaló néven konstruktoroknak hívjuk. A szabály bal oldala helyett a továbbiakban a minta ***[pattern]*** kifejezés is megfelelő. A minta részkifejezése a részminta ***[(sub)pattern]***.

Példák szabályos TRS beli kifejezésekre:

$$
\begin{array}{lll}
Fac\ 0 & \rightarrow & 1 \\
Fac\ n & \rightarrow & *\ n(Fac\ (-\ n\ 1)) \\
 & & \\
Cond\ True\ x\ y & \rightarrow & x \\
Cond\ False\ x\ y & \rightarrow & y
\end{array}
$$

Továbbá néhány ellenpélda, melyek szintaktikailag hibás szabályok:

x	$\rightarrow$	1	*ugyanis α_i (a szabály baloldala) nem állhat csupán egyetlen változóból.*
$Illegal\ x$	$\rightarrow$	y	*ugyanis β_i (a szabály jobboldala) nem tartalmazhat olyan változót, mely nem szerepel a baloldalon.*

2.2.3. TRS rendszerek működése

A TRS rendszerek képesek átalakítani a τ zárt kifejezéseket ***[closed term]***, azaz azokat, amelyek csak konstansokból állnak. A Clean és Miranda nyelvek kezdőszimbóluma is ilyen zárt kifejezés, melynek kiértékelése - átalakítások sorozata - a program futása.

Redex-ek

A (rész)kifejezést redex-nek [10] nevezzük - akkor és csak akkor -, ha létezik olyan szabály melynek bal oldala ***[pattern]*** megfelel a kifejezésnek. A szabály bal oldala megfelel a (rész)kifejezésnek, ha szintaktikusan megegyeznek a megfelelő konstansokkal a mintában. Azaz a redex egy olyan τ' részkifejezése a τ zárt kifejezésnek, hogy τ' szintaktikusan ekvivalens az egyik szabály α_i bal oldalának egy példányával ***[instance]***. Ahol az α_i kifejezés egy példányán, azt a $\sigma(\alpha_i)$ kifejezést érjük, ahol σ - az α_i változóiba - a zárt kifejezések helyettesítése. E feltételeknek való megfelelés esetén mondjuk, hogy τ' illeszkedik ***[match]*** α_i-re.

Az illesztendő zárt kifejezés:

$$Ap\ (Ap\ K\ K)\ (Ap\ S\ K)$$

Példa a kifejezések illesztésére:

$$\begin{array}{lll} Ap\ (Ap\ (Ap\ S\ x)\ y)\ z & \rightarrow & Ap\ (Ap\ x\ z)\ (Ap\ y\ z) \\ Ap\ (Ap\ K\ x)\ y & \rightarrow & x \end{array}$$

Ez a kifejezés illeszkedik a $Ap\ (Ap\ K\ x)\ y$ mintára. A szükséges helyettesítés a következő:

$$\sigma(x) = K, \sigma(y) = Ap\ S\ K$$

azaz,

$$\sigma(Ap\ (Ap\ K\ x)\ y) = Ap\ (Ap\ K\ K)\ (Ap\ S\ K)$$

Néha viszont nyilvánvalónak tűnő esetekben sem jön létre az illeszkedés. Legyen a TRS az alábbi:

$$Eq\ x\ x \rightarrow True$$

Zárt kifejezésnek a $Eq\ (+\ 0\ 1)$ 1-et választva, nem tudunk illeszteni, ugyanis nem létezik olyan helyettesítése x-nek, amely a szabály bal oldalát adná. Ellenben a $Eq\ (+\ 0\ 1)\ (+\ 0\ 1)$ illeszkedik a szabályra.

Következésképpen az, hogy egy részkifejezés redex, a TRS rendszerben található szabályok szerkezetén múlik, mivel egy részkifejezés akkor és csak akkor redex, ha létezik olyan szabály a TRS rendszerben, amelynek a mintájára (bal oldalára) illeszkedik a részkifejezés. Ez ellentétben áll a λ-kalkulussal, ahol minden esetben külön redex-ről beszélünk, valahányszor egy függvény van az argumentumban.

Kifejezések átírása

Legyen a TRS az alábbi:

$$Hd\ (Cons\ a\ b) \rightarrow a$$

A kifejezésnek, válasszuk a $Hd\ (Hd\ (Cons\ (Cons\ 0\ Nil)\ Nil))$, és lássuk milyen helyettesítés lehetséges:

$$\begin{array}{ll} Hd\ (Hd\ (Cons\ (Cons\ 0\ Nil)\ Nil)) & \|\ \sigma(a) = Cons\ 0\ Nil, \quad \sigma(b) = Nil \\ \rightarrow_{Hd}\ Hd\ (Cons\ 0\ Nil) & \|\ \sigma(a) = 0, \quad \sigma(b) = Nil \\ \rightarrow_{Hd}\ 0 & \end{array}$$

Egy τ kifejezésen végrehajtott átírási ***[rewrite]*** [10] (vagy redukciós) lépésnek nevezzük a τ- beli redex kicserélését τ'-re, ami azon szabály ***[rule]***

jobb oldala, amelyre a kifejezés (a megfelelő σ helyettesítéssel) illeszkedett. A τ' redex amely megegyezett a $\delta(\alpha_i)$-vel lecserélődik $\delta(\beta_i)$-re, ami a megfelelő szabály jobb oldalának a helyetesített változata.

A kifejezések TRS beli átírása ekvivalens a λ-kalkulusbeli β-redukcióval. A legfőbb különbség az, hogy a λ-kifejezés (pl: $(\lambda x.M)\ N$) mindig redex, mely köthető egy konkrét változóhoz. TRS rendszereknél csak a TRS beli szabályoktól függ, hogy egy (rész)kifejezés redex -e, vagy sem. Ráadásul a TRS-beli szabályoknak (függvényeknek), egynél több argumentumot is lehet definiálni. Természetesen csak azok a szabályok használhatóak fel, amelyeknek valamennyi argumentumuk létezik, és megfelel a mintának. Ezért a többargumentumos függvények argumentumait sem szükséges Curry-transzformációval szimulálni.

Példák szabályos többargumentumos függvényekre:

$$\begin{array}{lcl} G\ x\ y & \rightarrow & x \\ H\ x\ y\ z & \rightarrow & G\ x\ y \end{array}$$

Azaz, a $G\ 2\ 3\ \rightarrow_G\ 2$ átírás létrejön, viszont a $G\ 2$ kifejezés nem lesz redukálható, ugyanis nincs benne redex, mivel csak egyetlen argumentuma van, a mintának pedig kettő.

A névütközés ***[name confilct]***, amely a λ-kalkulusban előfordul, a TRS rendszerekben nem merülhet fel. A kezdeti kifejezés ***[initial term]*** egy zárt kifejezés, és szabad változók nem keletkezhetnek egy átírási (redukciós) lépés során. Tehát a teljes kiértékelés során sem jelenhetnek meg a kifejezésben változók, ezért nem lehet névütközés sem.

δ-szabályok

A TRS rendszerekben, a λ-kalkulushoz hasonlóan, létez(het)nek külsőleg definiált függvények és konstansok. Ezeket a külsőleg definiált függvényeket röviden δ-szabályoknak ***[δ-rule]*** [10], vagy δ-függvényeknek nevezzük. Az ilyen δ-függvény aktiválható, ha a szükséges argumentumai megvannak. Ekkor az illeszkedő részkifejezést δ-redexnek hívjuk. Ha egy ilyen külsőleg definiált szabály kerül alkalmazásra, akkor δ-redukciós lépésről beszélünk.

Íme egy δ-redukciós lépés, amelyet a $+$ δ-szabályon és a 2 valamint a 3 külsőleg definiált konstansokkal mutatok be:

$$+\ 2\ 3\ \rightarrow_{\delta+}\ 5$$

Noha lehetséges ezekkel azonos működésű szabályokat közvetlenül a TRS rendszerben definiálni, de a legtöbb esetben sokkal olvashatóbb és a lényeget jobban szemlélteti, ha ezekre külsőleg definiált függvényeket vezetünk be.

A külsőleg definiált + szabály aktiválódásának előfeltétele, hogy a argumentumok valóban a megfelelő típusból való konstansok legyenek. Ebben az esetben a paramétereknek kötelezően konstans számoknak kell lenniük, különben a kifejezés nem δ-redex és nem jöhet létre δ-redukció.

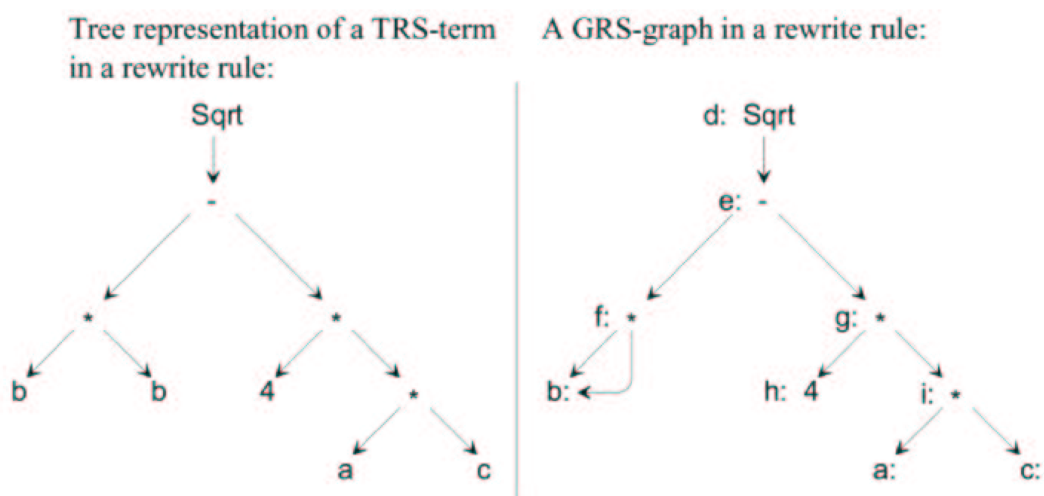
2.3. GRS rendszerek

A gráf átíró rendszerek (röviden GRS) ***[Graph Rewriting System]*** [10] munkából részletesen megismerhetőek. A GRS rendszerek fogalmát 1987-ben vezzeték be Barendregt, Eekelen, Glauert, Kennaway, Plasmeijer és Sleep. A gráf átíró rendszerek, a kifejezés átíró rendszerek azon hibáját hivatottak orvosolni, hogy a legtöbb feladatban többször van szükség bizonyos részeredményre a további számításokhoz. A kifejezés kiértékelő rendszerek, az ilyen esetekben, az érintett részkifejezést többször számolják ki, ami nagy mértékben rontja a hatékonyságot.

2.3.1. GRS rendszerek fogalma

A GRS rendszerek alapgondolata az, hogy ha a kifejezéseket irányított gráfokra cseréljük, aminek köszönhetően lehetőség van a gráf pontjaira való többszörös hivatkozásnak, ezzel a többszörös kiértékelések elkerülésének.

Szemléletes reprezentációja a TRS rendszer beli kifejezésnek a kitüntetett csúcsú fa volt. A GRS rendszerben ennek a cimkézett körmentes irányitott gráf felel meg. A gráf csúcsait egyedi cimkékel látjuk el, melyeket ***[node-id]***-eknek nevezzük, és a részgráfok azonosítására használjuk. Íme egy példa, mely a TRS és GRS rendszerek közötti különbséget szemlélteti [10]:



Az átíró szabályok fogalma is áthozható a gráf átíró rendszerekbe. A zárt és nyitott kifejezések, a TRS rendszerbeli átírások fontos elemei, fogalmának

is megfeleltethető valami hasonló a GRS rendszerben. A GRS rendszer átíró szabályaiban szereplő gráfok, nyitott gráfok ***[open graphs]***, mert egyedi cimkéik változók. Az átírásban szereplő gráfok, zárt gráfok***[closed graphs]***, mert cimkéi konstansok. Ez különbség a TRS és a GRS-beli változók ***[variable]*** között.

A legfontosabb különbség a TRS és GRS rendszerek között a működésükben van. A GRS rendszereknek a gráf struktúra miatt a

$$Double\ x \quad \longrightarrow \quad +\ x\ x$$

alakú kifejezések esetében, az x értéke - a kiértékelése után -, mind a két hivatkozott helyen használható, ellentétben a TRS rendszerekkel, ahol a fenti kifejezéskiértékeléséhez kétszer kell x értékét kiszámítani. Az ilyen többszörös kiértékelés különösen hátrányos, ha a paraméterként átadott x kiértékelése nem triviális.

2.3.2. GRS rendszerek felépítése

A GRS rendszer az átíró szabályok összessége. Ezen szabályok határozzák meg, miként lehet a gráfokat más gráfokká átírni ***[rewriten]***, (redukálni ***[reduced graph]***). A gráfok leírására két forma terjedt el a rövid alak ***[shorthand form]*** és a kanonikus alak ***[canonical form]***. A rövid alak esetén a gráf szerkezetét nem kell teljesen leírni, ha egyértelműen meghatározható a jelölésből. Ezzel a jelöléssel, a TRS fa stuktúrájához hasonló gráfok esetében a TRS beli szabályokhoz igen hasonló alak írható fel. A kanonikus formával, mindent expliciten definiálni kell. Ennek megfelelően a rövid alakot használjuk a számítások leírására, míg a kanonikus alakot például a GRS szemantikájával foglalkozó feladatokban. A két alaknak kölcsönösen megfeleltehetőnek kell lennie. Az egyszerűség kedvéért, egyenlőre használjuk a rövid alakot.

A gráf

A gráfok halmazát a konstans szimbólumok és a csúcsokhoz használt azonosítók (Csúcs azonosító) ***[node-id]*** halmaza felett a következőképpen lehet definiálni:

Az általános jelölés szerint a konstansok nagy betűvel, számjeggyel vagy valamilyen speciális karakterrel (például: $\star$) kezdődnek.

A "Csúcs-azon-változó"-k viszont kis betűvel, a "Csúcs-azon-konstans"-ok pedik a "@" szimbólummal kell kezdődjenek.

2.3.3. GRS rendszerek működése

A GRS rendszer működtetésének célja, hogy a kezdeti gráfból eljussunk a redukciók alkalmazásával egy tovább már nem redukálható gráfhoz. A kezdeti

Gráf a GRS rendszerben:

Gráf	=	CsúcsDefinició{',' CsúcsDefinició};	
CsúcsDefinició	=	Csúcs azonosító ':' Csúcs;	
Csúcs	=	Szimbólum{Argumentum}	| Üres csúcs;
Szimbólum	=	Konstans;	
Argumentum	=	Csúcs azonosító;	
Csúcs azonosító	=	Csúcs-azon-változó	| Csúcs-azon-konstans;
Üres csúcs	=	'⊥';	

Szintaktikailag helyes kanonikus formában leírt gráfok:

```
@1:  Hd @2,
@2:  Cons @3 @4,
@3:  0,
@4:  Nil

@1:  Add @2 @2,
@2:  Fac @3,
@3:  1000

@1:  Cons @2 @1
@2:  1

@1:  Tuple @2 @3 @4 @5 @6,
@2:  1,
@3:  −3,
@4:  5,
@5:  −7,
@6:  11

x:  Add y z,
y:  Succ z,
z:  Zero
```

gráfot ***[The initial graph]*** a következőképpen szokás definiálni:

A kezdeti gráf:

```
@DataRoot:   Graph @StartNode,
@StartNode:  Start
```

A GRS rendszert a *Start* kifejezéstől, kezdve szokás felépíteni.

Redex-ek

Nevezzük a konstans szimbólumokból és csúcs-azonosító változókból ***[node-id]*** álló kifejezéseket redex mintának ***[redex pattern]*** [10]. A minta egy

példányának nevezzük az adat gráf egy részgráfját, melynek szimbólumai és a csúcsainak szerkezete megfelelnek ***[match]*** a mintának. A fenti részgráfot röviden hívjuk redex-nek ***[reducable expression]***.

Az adatgráf átírása

Az adatgráf átírásának [10] módja kissé eltér a kifejezések átírásának módjától. A illeszkedő redex megtalálása után az adott szabálynak megfelelően előállnak új csúcsok. Ezeket az új csúcsokat be kell illeszteni az adatgráfba. Be kell állítani a rá mutató, valamint az általuk mutatott referenciákat.

Az új csúcs beillesztése után lesznek olyan csúcsok, melyek nem érhetőek el a kitüntetett @DataRoot csúcsból. Ezek a gráf további kiértékelésében nem játszanak szerepet, tehát fel lehet szabadítani az általuk foglalt memóriaterületet. A fenti mechanizmus a szemétgyüjtés tipikus modellje.

A kifejezéseknél tárgyaltuk az úgynevezett δ-kifejezéseket. Ezek kezelése megegyezik a TRS és GRS rendszerekben.

Tekintsük a következő rövid példát példát [11]:
Legyen az átíró szabályok halmaza a következő:

```
x =: Add y z
y =: Zero     => z              (1)

x =: Add y z
y =: Succ a   => m =: Succ n
                 n =: Add a z   (2)
x =: Start    => m =: Add n o
                 n =: Succ o
                 o =: Zero      (3)
```

Az adatgráf kezdeti értéke legyen a szokásos:

```
@DataRoot  =: Graph @StartNode
@StartNode =: Start
```

Az egyetlen redexre a *Start* csúcsra vonatkozó szabály eredménye:

```
@A         =: Add @B @C
@B         =: Succ @C
@C         =: Zero
```

Belillesztve ezt az adatgráfba:

```
@DataRoot  =: Graph @A
@StartNode =: Start
@A         =: Add @B @C
@B         =: Succ @C
@C         =: Zero
```

Mivel a *@StartNode* szimbólum a *@DataRoot*-ból nem elérhető, ezt a szemétgyűjtő felszabadíthatja:

```
@DataRoot  =: Graph @D
@D         =: Succ @C
@C         =: Zero
```

Mivel nem lehet több redukciót végrehajtani, ezért a fenti adatgráf a függvény kiértékelésének eredménye.

2.4. A Clean nyelv

A Clean nyelv [11] egy tisztán funkcionális nyelv, melyet a Nijmengeni Katolikus Egyetem oktatói és hallgatói fejlesztenek. A Clean nyelv fejlesztésének célja kettős, egyrészt a funkcionális technikák fejlesztéséhez, illetve kutatási területként használják, másrészt HILT cég révén piaci forgalomba is kerül. A két felhasználási terület egy közel optimális lehetőség a nyelv, illetve a hozzá tartozó fordítóprogram fejlesztői számára, mivel helyet kapnak az legfrissebb elméleti kutatásokból származó eredmények és megakadályozza, hogy a nyelv *l'art pour l'art*, azaz öncélú nyelvvé váljon.

A Clean egy lusta kiértékelésű magasabb rendű funkcionális nyelv, amelynek alapja egy gráf-átíró rendszer. Ebben a rendszerben a programozó átíró-szabályokat (függvényeket) definiál. A program futása pedig, a *Start* függvény kiértékelésének folyamata. Mivel a felhasználóknak egy valós környezetben futó teljes értékű programnyelvre volt szükségük, a Clean nyelv fejlesztői kibővítették a GRS rendszer működését néhány elemmel. Ilyen például a futtató operációs rendszer szolgáltatásainak ***[interface]*** elérhetővé tétele, amellyel a program külső forrásból (háttértárolóról, konzolról, grafikus felhasználói felületről ***[GUI]***, hálózatról, ...) is szerezhet adatokat. Bár ez a szemlélet különbözik a funkcionális programozás alapgondolatától - "kölcsönhatás, nem kívánt mellékhatás-mentes programozás"-, a fejlesztőknek sikerült megfelelően beépíteni a modellbe.

2.4.1. A Clean nyelvű kódok

A Clean nyelvű programok átíró szabályokból (függvényekből) állnak. Minden szabálynak van típusa, melyet vagy jelöl a programozó a szabály deklarációjában, vagy a fordító automatikusan határozza meg, a szabály definiciója alapján.

Modulok

A fordítóprogram a nyelv definiciója alapján megköveteli a moduláris programozást. A Clean nyelv három féle modult használ:

- implementációs modul *[implementation module]*
- definiciós modul *[definition module]*
- rendszer modul *[system module]*

A definiciós modul célja, hogy megvalósítsa az információrejtést a különböző modulok között. Az implementációs modulban, a modul összes függvénye látható, míg más modulokból csak azok, melyeket a definiciós modul is tartalmaz.

A rendszer modul egy speciális definiciós modul, amely azt jelzi, hogy a hozzá tartozó implementáció nem Clean nyelven íródott, hanem egy ABC kódnak nevezett köztes kódban, amelyről a következő fejezetben lesz szó. Az ilyen rendszer-implementációs modulpárok általában a δ-redukciók megvalósítását tartalmazzák.

Vezérlési szerkezetek

A Clean függvények alapvető (elsődleges) "vezérlési szerkezete" a mintaillesztés. Mintaillesztésnek hívjuk azt a módszert, amikor egy függvénynévhez -formálisan - több definició is tartozik. Például:

```
fact 0 = 1
fact n = n * fact (n - 1)
```

A *fact* névhez tartozó definiciók közül, a Clean nyelv a mintaillesztésének szemantikája miatt, először az első sor kerül illesztésre. Ha nem illeszkedik, akkor a második sor kerül illesztésre. Ha az utolsó (esetünkben a második) elem illesztése sem sikeres, akkor a program futása hibaüzenet megjelenítésével befejeződik.

A Clean nyelv ezen kívül támogatja az *if-then-else* típusú feltételek megadását is:

```
Insert e [] = [e]
Insert e [x:xs]
   | e<=x      = [e,x:xs]
   | otherwise = [x: Insert e xs]
```

A feltételt a "|" szimbólum vezeti be, majd ezt követi az adott feltételhez tartozó definició.

Bizonyos esetekben, ha a definició túl nagy -, vagy nem elég átlátható lenne, akkor bevezethetőek lokális alkifejezések a *where* kulcsszó után. Például:

```
msort :: [a] -> [a] | Ord a
msort xs
  | len <=1   = xs
  | otherwise = Merge (msort ys) (msort zs)
 where
    ys = take half xs
    zs = drop half xs
    half = len/2
    len = length xs
```

Bizonyos esetekben a *where* kifejezéssel optimalizálni is lehet a generált kódot, segítve a fordítót a megegyező értékek közös csúccsá való összevonásában.

```
Start = (length (queens 12), first (queens 12))
```

helyett,

```
Start = (length result, first result)
  where result = queens 12
```

2.4.2. A Clean típusrendszere

A Clean egy erősen típusos programozási nyelv. A típus fogalom bevezetése a funkcionális nyelvekbe egyrészt a programozót segíti olyan programok tervezésében, melyek helyesek és könnyen áttekinthetőek. Másrészt a fordítóprogram számára lehetővé teszi hatékonyabb kód előállítását.

Előre deifiniált egyszerű típusok

A nyelv rendelkezik 6 előre definiált egyszerű típussal és a hozzájuk tartozó alapvető fontosságú függvénnyel:

Int	$+, -, ==, >=, \%, \ldots$
Real	$+, -, ==, >=, sin, exp, \ldots$
Char	$+, -, ==, >=, \ldots$
Bool	$==, not, \&\&, \|\|, \ldots$
Srting	$+, -, ==, >=, +++, \ldots$
File	$StdIO, FOpen, FWriteC, \ldots$

Előre deifiniált összetett típusok

Az előre definiált összetett típusok közül kettőt kell mindenképpen megemlíteni, a listákat és a rendezett n-es típusokat. A listákat ***[list]*** azonos típusú -, míg a rendezett n-est ***[tuple]*** különböző elemek tárolására használhatjuk. Míg rendezett n-esek pontosan a definiálásukkor megadott számú és típusú elemet tartalmaznak, addig a listákban az elemek száma dinamikusan változtatható.

Új típusok definiálása

A Clean nyelv lehetővé teszi a programozónak saját adattípusok definiálását. A legegyszerűbb lehetőség, valamilyen már létező típus átnevezésével létrehozni az új típust.

Hatékony algebrai adattípusok létrehozására is lehetőség van. Ez elsősorban matematikai eredetű feladatok megoldásánál lehet hasznos. Amennyiben a külvilág elől el kívánja rejteni az adattípus alkotója a konkrét megvalósítást, akkor absztrakt adattípust létrehozva a definiciós modulon keresztül, csak a típus neve és a rajta értelmezett függvények lesznek elérhetőek.

Függvény típusok

Függvények szignatúrája éppúgy használható, mint bármely beépített típus. Lehetőség van függvény paraméterű, sőt függvény visszatérési értékű, úgynevezett magasabb rendű függvények létrehozására.

2.4.3. A kiértékelési rendszer

A Clean nyelv több kiértékelési módszert támogat. A preferált kiértékelésforma függ a kiértékelendő elem típusától, és a kontextustól. A programozónak bizonyos esetekben be kell avatkoznia, például sebesség optimalizáció céljából. Például, az eredetileg lusta kiértékelésű részgráf, mohó kiértékelését kérve.

Lusta kiértékelés

Lusta kiértékelésnek nevezzük a funkcionális programok azon stratégiáját, hogy a kifejezéseket csak akkor számolják ki, ha már valóban szükség van az értékükre. Gyakran előfordulnak olyan problémák, hogy egy programban felesleges kódrészletek vannak. Ezek kiértékelését elkerülve, a program sebessége és memóriaigénye is csökkenthető.

Mohó kiértékelés

Mohó kiértékeléssel történik az egyszerű operátorok (pl: egész, valós típusokon végzett összeadás, kivonás, szorzás, osztás, stb.). Matematikai algoritmusokban, ahol ismert a feladat karakterisztikája érdemes a mohó kiértékelést "választani".

Curry kiértékelés

A nyelv lehetőséget biztosít félig kiértékelt függvények paraméterként való átadására. Az alábbi példában a *drop* kétparamétres függvényből név nélküli egyparaméteres függvényt "csinálunk", majd átadtuk a *twice* nevű függvénynek.

```
twice :: (x -> x) x -> x
twice f x = f (f x)

Start = twice (drop 1) [1..100]
```

2.5. Az ABC gép

Az ABC absztrakt gép, mely a Clean nyelv fordítóprogramjának köztes kódját az ABC-kódot képes futatni.

Az ABC név a gép három komponensének nevéből jön, ezek az A-, B-, és C-vermek.

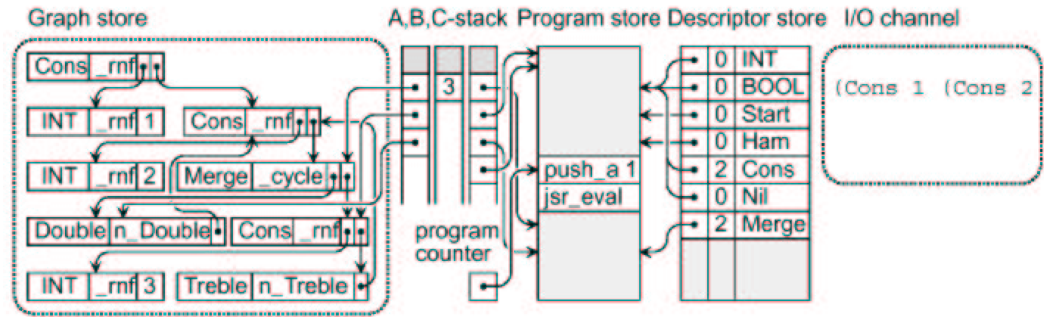
A gép bevezetésének célja a fordítóprogram egyszerűsítése, és hatékony kód generálása. A gép specifikációja [10] a Clean szintaxisának megfelelően íródott.

2.5.1. Az ABC gép karakterisztikája

Az ABC gép az idealizált gráfátíró rendszer, és a tradicionális verem alapú gép keveréke.

A gép a következő komponensekből áll:

- ***[graph store]***, mely az átírható gráfot tartalmazza.
- ***[program store]***, mely a végrehajtandó utasítás-szekvenciát tartalmazza.
- ***[program counter]***, mely a következőleg végrehajtandó utasítás azonosítója.
- ***[A(rgument)-stack]***, melynek elemei, a referenciák a gráf csúcsaira.
- ***[B(asic value)-stack]***, melynek elemei, primitív típusú elemek.
- ***[C(ontrol)-stack]***, melynek elemei, a visszatérési címeket tartalmazzák.
- ***[descriptor store]***, mely a deklarált szimbólumokról tárol információkat.
- ***[I(nput)/O(utput) channel]***, a program eredményének megjelenítéséhez.



Ennek megfelelően az absztrakt ABC gép a következő összetett állapottal bír:

$$state == (astack, bstack, cstack, graphstore, descrstore, pc, programstore, io)$$

A gép ezen alacsony szintű tárolóinak elérését a mikro-utasítások ***[micro-instruction]*** teszik lehetővé.

A gép utasításai megváltoztatják a fenti állapotot. Ennek megfelelően a gép utasításai a következő alakúak:

$$instruction == state \rightarrow state$$

Mivel az adattárolókat csak a mikró-utasításokon keresztül lehet elérni, ezért az ABC gép utasításai mikró-utasításokkal vannak leírva.

Mikró-utasítások

A mikro-utasítások a nevükben elkódolt különböző adattárolók primitív műveletei.

A legegyszerűbb a **program tároló**, melynek csupán két művelete van. A *ps_init*, amely utasításokból előállítja a tárolót, és a *ps_get*, amely egy azonosítóhoz (sorszámhoz) utasítást rendel.

$$\begin{array}{lll} ps_get & :: & instrid \rightarrow programstore \rightarrow instruction \\ ps_init & :: & [instruction] \rightarrow programstore \end{array}$$

Az utasítások közötti "bolyongást" az **utasítás számláló *[program counter]*** 6 mikró utasítása segíti. A 6 utasítás között megtalálható egy konstruktor *pc_init*, egy léptető *pc_next*, kettő melyekkel az utasítás-azonosítóhoz lehet hozzáférni *pc_get* és *pc_update*, valamint kettő a program befejezésének kezelésére *pc_halt* és *pc_end*.

$$\begin{array}{lll} pc_init & :: & pc \\ pc_next & :: & pc \rightarrow pc \\ pc_halt & :: & pc \rightarrow pc \\ pc_end & :: & pc \rightarrow bool \\ pc_get & :: & pc \rightarrow instrid \\ pc_update & :: & instrid \rightarrow pc \rightarrow pc \end{array}$$

Az ABC gép egyik legfontosabb része a **gráftér** ***[graph store]***. Ennek kezelésére több mint 20 mikró utasítást definiáltak, melyek a gráf inicializálását, a gráf topológiájához és a csúcsaihoz való hozzáférést kezelik. A gráf csúcsai nem tartalmaznak szimbólumokat, csak leírókat ***[descriptor identification]***, melyek a leíró tárolókban ***[descriptor store]*** kerülnek összekötésre a konkrét szimbólumokkal.

A gráftér globális kezelését végző függvények:

$$\begin{aligned} &gs_get && :: && nodeid \rightarrow graphstore \rightarrow node \\ &gs_init && :: && graphstore \\ &gs_newnode && :: && graphstore \rightarrow (graphstore, nodeid) \\ &gs_update && :: && nodeid \rightarrow (node \rightarrow node) \rightarrow graphstore \rightarrow graphstore \end{aligned}$$

Mikró utasítások a gráf beli csúcsok adatainak lekérdezésére:

$$\begin{aligned} &n_arg && :: && node \rightarrow arg_nr \rightarrow arity \rightarrow nodeid \\ &n_args && :: && node \rightarrow arity \rightarrow nodeid_seq \\ &n_arity && :: && node \rightarrow arity \\ &n_B && :: && node \rightarrow bool \\ &n_descrid && :: && node \rightarrow descrid \\ &n_entry && :: && node \rightarrow instrid \\ &n_N && :: && node \rightarrow int \\ &n_nargs && :: && node \rightarrow nr_args \rightarrow arity \rightarrow nodeid_seq \end{aligned}$$

Gráf csúcsok vizsgálatát végző utasítások, melyek a mintaillesztés implementációjában játszanak fontos szerepet:

$$\begin{aligned} &n_eq_arity && :: && node \rightarrow arity \rightarrow bool \\ &n_eq_B && :: && node \rightarrow bool \rightarrow bool \\ &n_eq_descrid && :: && node \rightarrow descrid \rightarrow bool \\ &n_eq_I && :: && node \rightarrow int \rightarrow bool \\ &n_eq_symbol && :: && node \rightarrow node \rightarrow bool \end{aligned}$$

További függvények a csúcsok adattagjainak eléréséhez:

$$\begin{aligned} &n_copy && :: && node \rightarrow node \rightarrow node \\ &n_fill && :: && descrid \rightarrow instrid \rightarrow nodeid_seq \rightarrow node \rightarrow node \\ &n_fillB && :: && descrid \rightarrow instrid \rightarrow bool \rightarrow node \rightarrow node \\ &n_fillI && :: && descrid \rightarrow instrid \rightarrow int \rightarrow node \rightarrow node \\ &n_setentry && :: && instrid \rightarrow node \rightarrow node \end{aligned}$$

Az ABC gép a deklarált szimbólukat egy leíró táblán keresztül kezeli. Minden egyes szimbólumhoz és egyszerű típushoz létezik a **leíró táblában** ***[descriptor strore]*** egy egyedi bejegyzés, mely egy azonosítón ***[desc_id]*** keresztül érhető el.

ds_get	::	$descrid \to descrstore \to descr$
ds_init	::	$[descr] \to descrstore$
d_ap_entry	::	$descr \to instrid$
d_arity	::	$descr \to arity$
d_name	::	$descr \to string$

A gép a rendelkezésére álló három vermet alapvetően más-más feladatokra használja. Az **A-verem**ben a gráftér csúcsaira mutató referenciák találhatóak. Ezzel valósítható meg a függvények közötti paraméterátadás, hasonlóan az imperatív nyelv cím szerinti paraméterátadásához.

as_init	::	$astack$
as_get	::	$a_src \to astack \to nodeid$
as_topn	::	$nr_args \to astack \to nodeid_seq$
as_popn	::	$nr_args \to astack \to astack$
as_push	::	$nodeid \to astack \to astack$
as_pushn	::	$nodeid_seq \to astack \to astack$
as_update	::	$a_dst \to nodeid \to astack \to astack$

A **B-verem** primitív típusú elemeket (mint: egészek, valósak, logikai értékek) tartalmaz. Ezzel valósítható meg a függvények közötti paraméterátadás, hasonlóan az imperatív nyelv érték szerinti paraméterátadásához.

bs_init	::	$bstack$
bs_get	::	$b_src \to bstack \to basic$
bs_getB	::	$b_src \to bstack \to bool$
bs_getI	::	$b_src \to bstack \to int$
bs_copy	::	$b_src \to bstack \to bstack$
bs_popn	::	$nr_args \to bstack \to bstack$
bs_push	::	$basic \to bstack \to bstack$
bs_pushB	::	$bool \to bstack \to bstack$
bs_pushI	::	$int \to bstack \to bstack$
bs_update	::	$b_dst \to basic \to bstack \to bstack$

A különböző primitív típusokhoz tartozó specifikus operátorok is a mikróutasítások között szerepelnek. Az ilyen műveletek általában a verem tetején levő elem(ek) alapján számolnak és a verem tetejére teszik vissza a végeredményt. *(A gyorsabb kód érdekében érdemes az értékek egy részét regiszterben tárolni)* Ezek közül néhány egész operátor:

bs_addI	::	$bstack \to bstack$
bs_eqI	::	$bstack \to bstack$
bs_eqIi	::	$int \to b_src \to bstack \to bstack$
bs_gtI	::	$bstack \to bstack$

A **C-verem** az egymásba ágyazódó redukciós lépések implementációjának adatait tartalmazza. Egészen pontosan egy beágyazódó lépés kezdetekor bekerül az utasításszámláló aktuális értéke, amely az adott redukció befejeztével

visszaállításra kerül.

$$\begin{array}{lll} cs_init & :: & cstack \\ cs_get & :: & c_src \rightarrow cstack \rightarrow instrid \\ cs_popn & :: & nr_args \rightarrow cstack \rightarrow cstack \\ cs_push & :: & instrid \rightarrow cstack \rightarrow cstack \end{array}$$

Az **I/O csatorna** lehetővé teszi a program eredményének egy szekvenciális konzolon való megjelenítését. Az ennél finomabb I/O műveletekre is szüksége van egy teljes értékű programozási nyelvnek, de ezekre a műveletekre, már nem igaz a mellékhatás-mentességi tulajdonság.

Az ABC gép utasításkészletét a fenti mikró-utasításokkal definiálták, ennek megfelelően elemi állapotátmenetek sorozatával lehet, az ABC nyelvű kód hatásrelációját leírni.

Az utasítások egyszerűsége miatt az ABC gép hatékonyan és könnyen implementálható a ma elterjedt hardware-szoftware architektúrákban.

3. fejezet

Erőforrások használata

3.1. A programok által igénybe vett erőforrások osztályozása

A programok, programkomponsek erőforrásigénye a következő alaposztályokba csoportosíthatóak:

Processzor használata.

- A program futási ideje.
- A központi egység által a program futtatására szánt idő.

Háttértár használata.

- A program futtatásához szükséges minimális memória.
- Az esetleges szemétgyűjtő által kezelendő objektumok száma.
- A program működésével összefüggő file műveletek száma.

Hálózati erőforrások használata.

Egy program működését, illetve a felhasználónak az erről alkotott benyomását, alapvetően a futási idő és a maximálisan használt memória mennyisége határozza meg. Speciális feladatosztályok esetében - mint az elosztott számítási feladatok-, az összesített futási időre való jelentős hatása miatt, a hálózati erőforrások használatát is érdemes vizsgálat tárgyává tenni.

3.2. Erőforrásigény vizsgálata

A programok, programkomponensek viszgálatának egyik legelegánsabb, de a legnehezebb (legtöbb hibalehetőséget tartalmazó) módja, a program kódjának statikus vizsgálata. Mivel az általánosan használt funkcionális nyelvek

kifejezőereje igen nagy, ezért ennek becslésének várható hibája is, meghaladja az elfogadható értéket.

A fenti becsléseket egy egyszerűbb nyelv esetében pontosabban, és kisebb erőforrásigénnyel el lehetne végezni. Megfelelhetne például egy köztes nyelv, mint a Clean esetében az ABC. Ugyanis az ABC gép specialitása, hogy nagyon szabályozott a működése, azaz 3 vermet,és egy kiemelt csúcssal bíró gráfteret használ a program által igénybevett adatok tárolására. A kiemelt csúcsból elérhető csúcsok halmaza határozza meg a program által aktuálisan használt memóriát, mivel ez a szemétgyüjtő algoritmus indulópontja is. Ezen az úton a memória használata feltérképezhetővé válik, feltéve, hogy ismert az egyes függvények kiértékelésének száma (röviden: kiértékelésszám).

A kiértékelésszám meghatározása a processzor használat aproximációjában is szerepet kap. Ha ismert a kiértékelések mikéntje, lehet a függvényekre - a paraméter(ek)től függő -, végrehajtási idő becslést adni.

3.3. Kiértékelésszám meghatározása

Az egyes függvények kiértékelés-számának vizsgálata egy speciális részfeladata az erőforrásigény vizsgálatának. Ezen részfeladat megoldása megfelelően köthető pl: a Clean forráskódhoz, a *RWSDebug* könyvtári csomag segítségével. A funkcionális programok esetében - a kiértékelésszám tekintetében -, a rekurzív függvények kiértékelésszáma a mérvadó.

3.4. Konkrét esetosztályok elemzése

Az általános becslési eljárásokon kívül, illetve ezen eljárások pontosítása céljából, szükséges néhány konkrét esetosztály felállítása, és ezek elemeinek szisztematikus elemzése. A konkrét mérésekhez speciális mérési környezetet kell kialakítani.

3.4.1. Mérési környezet által vizsgált erőforrások

Ebben a fejezetben ismertetésre kerül az általam választott mérési környezet, amelynek egyes elemei manuálisak, míg mások félig automatikusak.

A mérési környezet a Clean környezethez készült.

Processzorhasználat

A processzorhasználat mérésére a következő három módszer valamelyike jöhet szóba:

- A forráskódban a *RWSDebug* csomag segítségével, időmérést végezni az egyes függvényekre.

- A lefordított összeszerkesztett programot egy külső rendszerből futtatni.
- Clean project fordítási opcióval.

A processzorhasználat mérését, általában nem érdemes magában a Clean nyelvű tárgykódban megtenni, mivel a mérés befolyásolná magát a mérendő folyamatot. Egyrészt a mérési eredményekben is szerepelnek a mérést végző kódrészletek, másrészt a mérésnek nem szabad a kiértékelés sorrendjét, illetve a kiértékelendő komponensek halmazát módosítaniuk.

Az általam választott megoldás a következő. Futtassuk le az eredeti függvényt többször. Majd az összesített futási idő alapján becsüljük meg, a függvény futási idejét ($\hat{T_{run}}$).

$$T_{total} = n * T_{init} + n * \hat{T_{run}} + (n-1) * T_{destroy}$$

$$\hat{T_{run}} = \frac{T_{total} - (n-1) * T_{destroy}}{n} - T_{init}$$

A Clean project fordítási opcióval történő vizsgálatának legnagyobb előnye, hogy egyszerűsíti a számítási képletet. Két használható fordítási opciót támogat a jelenleg hivatalosan kiadott változat:

Globális futási idő elemzés. Ez az elemzésforma a legegyszerűbb időmérésen alapul, vagyis a program, azaz a *Start* függvény kiértékelése előtti és utáni időből megállapítja az eltelt időt.
Főbb tulajdonságai:

+ Nem befolyásolja a függvények kiértékelésének menetét.
+ Viszonylag kis mértékben van hatással az összesített futási időre.
– Az általa adott időadatok csak 1/100 másodperc pontosságúak.
– Meglehetősen kevés adatot szolgáltat:
  - Futási idő.
  - Szemétgyűjtésre fordított idő.
  - Összesített idő. (A fenti kettő összege)
– Korlátozza hatékonyságát, hogy csak nagyobb függvényekre használható.

Időprofil elemzés. Ebben az esetben az elemzés két lépésből áll. Az első lépés az időprofil elkészítése, a program segítségével. Majd a második lépésként lehet az összegyűjtött adatokat egy segédprogrammal megjeleníteni. Hasonló megoldás szerepel [12] cikkben is.
Főbb tulajdonságai:

+ Nem befolyásolja a függvények kiértékelésének menetét.
+ A kapott időadatok matematikailag pontosak (1/1000000 másodperc pontosságúak).
+ Meglehetősen sok adatot szolgáltat, függvényenként:
  - Futási idő.
  - Futási idő százalékos aránya.
  - A gráf számára lefoglalt byte-k száma.
  - A gráf számára lefoglalt byte-k százalékos aránya.
  - Kiértékelésének száma.
    - "Strict"
    - "Lazy"
    - "Curried"
- A kiértékelés számlálók nem kezelik a közvetlen rekurziót, illetve a lusta kiértékeléseket.
- Jelentős mértékben hatással van az összesített futási időre. (Akár hatszorosára is nőhet.)
- Az általa adott időadatok esetenként ellentmondásosak (Pl: Előfordulhatnak negatív időadatok).
- Nem, illetve csak kis mértékben paraméterezhető.

Minimálisan szükséges memória

A Clean környezet támogatja a programonkénti memóriahasználati profilokat. Az egyes profilok meghatározzák a program futása során az ABC gép implementációjára maximálisan használható memória nagyságát.

Ezen ismeret alapján meghatározható, hogy egy adott függvény kiértékeléséhez, egy bizonyos paraméterhalmaz mellett, mi a minimálisan szükséges memória, amit rendelkezésre kell bocsátani.

Például, egy rendezési algoritmus esetében - amely egydimenziós számvektorokat képes rendezni -, a lényeges paraméter, a rendezendő vektor elemeinek száma.

Szemétgyüjtő működése

A Clean környezet a memória kezelésére szemétgyűjtő algoritmust használ. Az ehhez szükséges kódot a fordítóprogram automatikusan belefordítja az összes programba. A használt szemétgyűjtő paraméterezheztő a Clean keretrendszeren keresztül. A legfontosabb paraméter a diagnosztikai kiíratás engedélyezése, amely a program futása során a szemétgyűjtés folyamatáról add tájékoztatást.

4. fejezet

Vizsgált speciális esetosztályok

Ebben a fejezetben, néhány általánosan ismert függvényosztály viselkedését mutatom be. A vizsgálat egyrészt kiterjed a különböző implementációk közötti, valamint az azonos implementációk, különböző paramétereinek erőforrásigénnyel kapcsolatos hatásaira.

4.1. Lineáris struktúrák

4.1.1. Vermek és sorok

A két adattípus a vizsgálat szempontjából közösnek tekinthető, ugyanis csak az elemeikhez való hozzáférés mikéntje bír jelentőséggel, az erőforrásigény vizsgálatának szempontjától.

A vizsgálat a következő reprezentációkra történt meg:

- Absztract algebrai típussal (konstruktorokkal reprezentálva)
- Lusta kiértékelésű listával ***[Lazy list]***
- Szigorú kiértékelésű listával ***[Strict list]***

Első példának tekintsük a legegyszerűbb műveletekkel rendelkező lineáris adatszerkezetet, a vermet. A verem viszonylag kevés függvénye, és ezek egyszerűsége lehetővé teszi, a lusta kiértékelésű lista és az algebrai reprezentáció (lásd a 34. oldalon) összehasonlítását.

Rinus Plasmeijer és Marko van Eekelen [10] felhívják az olvasó figyelmét a lusta kiértékelésű listák és a konstuktoros reprezentáció közös gyökerére. Ez a közös gyökér, a Clean fordítóprogramja által rendelkezésünkre bocsátott köztes kód (ABC kód) alapján, a lusta lista konstruktorainak *_Nil*, *_Cons* megfeleltetése, az absztrakt algebrai reprezentációban általunk definiált *Nil*, *Cons* konstuktoroknak. Azaz, innentől kezdve az absztrakt algebrai típus konstrutorral történő reprezentációját, és a lusta kiértékelésű listát ekvivalensnek tekintjük.

```
implementation module stack1

:: Stack1 a :== [a]


Push :: a (Stack1 a) -> Stack1 a
Push e s = [e:s]

Pop :: (Stack1 a) -> Stack1 a
Pop [e:s] = s

top :: (Stack1 a) -> a
top [e:s] = e

Empty :: Stack1 a
Empty = []
```

stack1.icl

```
implementation module stack2

:: Stack2 a = Nil
            | Cons a (Stack2 a)

Push :: a (Stack2 a) -> Stack2 a
Push e s = Cons e s

Pop :: (Stack2 a) -> Stack2 a
Pop (Cons e s) = s

top :: (Stack2 a) -> a
top (Cons e s) = e

Empty :: Stack2 a
Empty = Nil
```

stack2.icl

Hasonló állítás belátható a rekord szerkezetek és a absztrakt algebrai reprezentáció konstruktorai között is.

A második példa célja a lusta-,***[Lazy list]*** és a szigorú kiértékelésű lista ***[Strict list]*** kezelésének összehasonlítása. Erre a célra készült két sort megvalósító modul. Az első szigorú kiértékeléssel oldja meg a feladatát, addig a második (lásd a 35. oldalon) lusta kiértékelésű listával. A tesztkörnyezet mindkét esetben abból állt, hogy elhelyeztünk 25000 elemet a listában, az *InitQueue* függvény segítségével, és kiolvastuk a legelső elemet.

module	function	Strict (n)	Lazy (n)	Curry (n)	Alloc (byte)	Time (s)
_SystemEnum-Strict	_from_to_sts-;102	25001	0	0	500000	0.002709
_SystemEnum-Strict	_from_to_sts-;37	0	1	0	0	0.000001
queue1	InitQueue	0	1	0	0	0.000001
queue1	Touch	1	0	0	0	0.001528
qTest1	Start	0	1	0	32	0.000002
System	start	0	0	0	12	0.000004

module	function	Strict (n)	Lazy (n)	Curry (n)	Alloc (byte)	Time (s)
_SystemEnum	_f8._from-_to;17._from-_to;17.+-;6.one;11	0	25000	0	0	0.000296
_SystemEnum	_from_to;6	0	1	0	0	0.000010
_SystemEnum	_from_to;17	25001	0	0	500000	-0.000716 !!!
queue2	InitQueue	0	1	0	0	0.000000
queue2	Touch	1	0	0	0	0.002226
qTest2	Start	0	1	0	32	0.000002
System	start	0	0	0	12	0.000004

```
implementation module queue1

:: Queue1 a :== [!a!]

Put :: a (Queue1 a) -> Queue1 a
Put e q = [!e:q!]

Get :: (Queue1 a) -> Queue1 a
Get [!e:[!!]!] = [!e!]
Get [!e:q!] = [!e:Get q!]

Touch :: (Queue1 a) -> a
Touch [!e:[!!]!] = e
Touch [!e:q!] = Touch q

Empty :: Queue1 a
Empty = [!!]

InitQueue :: [!a!] -> Queue1 a
InitQueue q = q
```

queue1.icl

```
implementation module queue2

:: Queue2 a :== [a]

Put :: a (Queue2 a) -> Queue2 a
Put e q = [e:q]

Get :: (Queue2 a) -> Queue2 a
Get [e:[]] = [e]
Get [e:q] = [e:Get q]

Touch :: (Queue2 a) -> a
Touch [e:[]] = e
Touch [e:q] = Touch q

Empty :: Queue2 a
Empty = []

InitQueue :: [a] -> Queue2 a
InitQueue q = q
```

queue2.icl

Konklúzió

A primitív típusok kezelése minimális erőforrásigénnyel történik, viszont mind a két esetben a létrejövő lista-szerkezet felépítése jelentős mennyiségű memória allokációjával jár. A *_SystemEnumStrict* és a *_SystemEnum* csomagok esetében kétféle elemből képes a *_from_to* függvény listát előállítani, a karakterből és az egész számból. A mérések tanulsága alapján, a lefoglalt memória mennyisége a $(20 * hossz)$ **byte** képlettel írható fel, a fenti csomaggal előállított szigorúan monoton növekvő lista esetében. Ugyanez az arány szigorúan monoton csökkenő listánál $(24 * hossz)$ **byte**-ra módosul.

A nem primitív típusok esetében a szigorú kiértékelésű listák használatakor a lista kiértékelése, az elemek kiértésének idejével hosszabb, mint a lusta kiértékelésű listáknál.

4.1.2. Rendező algoritmusok

A listák összehasonlítás alapú, rendező algoritmusok lépésszámának elméleti optimuma az $O(n * lg\ n)$. Az ilyen összehasonlításon alapuló algoritmusok erőforrásigényét tehát elsődlegesen a memóriaigényre, és a szemétgyűjtő algoritmus futására érdemes vizsgálni.

Három, a [7] előadásból származó, algoritmus került górcső alá, ezek az összefésüléses rendező (lásd **MergeSort.icl** a 49. oldalon), a beszúrásos

rendező (lásd **InsertSort.icl** a 49. oldalon) és a gyorsrendező (lásd **QuickSort.icl** a 50. oldalon).

Konklúzió

Az algoritmusok legrosszabb eseteit vizsgálva a következő felső becsléseket lehet tenni. A három algoritmus közül kettő -a gyorsrendező és a beszúrásos rendező -, $O(n^2)$ memóriát igényel. Ennél jobb eredményei vannak az összefésüléses rendezőnek, mivel memóriaigénye $O(n * log_2 n)$ képlettel felülről becsülhető.

kódrész	allokált memória	kiértékés száma
Merge;4	$12 * n \ log_2 n$	$n-1$ strict, $\frac{n \ log_2 n}{2}$ lazy
msort;3	$56 * n$	
drop	$6 * n \ log_2 n$	$n - 1$ strict
take	$6 * n \ log_2 n$	$\frac{n \ log_2 n}{2} + (n-1)$ strict
Összesen	$24 * n \ log_2 n$ + 56 * n	

MergeSort.icl

kódrész	allokált memória	kiértékés száma
Insert;4	$6 * n(n-1)$	$\frac{n*(n-1)}{2}$ lazy, n strict
isort;3	$12 * n$	
Összesen	$6 * n(n-1) + 12 * n$	

InsertSort.icl

kódrész	allokált memória	kiértékés száma
g_c1;8;33;9	$6 * n(n-1)$	$\frac{n*(n+1)}{2}$ strict
qsort;4	$72 * n$	
++	$12 * n$	
Összesen	$6 * n(n-1) + 84 * n$	

QuickSort.icl

A nagyságrendek meghatározásán túl a hozzájuk tartozó konstansok, és a memóriaallokációért felelős kódrészletek is azonosításra kerültek. Ennek megfelelően megállapítható, hogy memória-allokáció tekintetében $n \in [4..10]$

esetén a QuickSort ideálisabb, mint a MergSort. Az InsertSort pedig $n <= 27$ esetén igényel kevesebb memóriát, mint a MergSort.

A legtöbb esetben a nagyságrendet meghatározó tényezők, a kódrészletek kiértékelésszámában is megjelennek, ami szintén szerepet játszik az összesített futási időben.

4.2. Fa jellegű struktúrák

A fa jellegű struktúrák tulajdonságai alapvetően hasonlítanak a lista szerkezetek tulajdonságaihoz. A fa stuktúrák esetében egy node, a kötelező konstans szimbólumon kívül nem egy, hanem r csúcs azonosítót tartalmaz. Ennek megfelelően beszélünk r-áris fákról. Az 1-áris fákat röviden listáknak nevezzük.

Ennek megfelelően az ilyen adattípusok esetében a mérések eredménye - az adott algoritmus erőforrásigénye -, nem csak a tartalmazott elemek számától függ, hanem a fa szerkezetétől, általánosságában a fa mélységétől. Elméleti mukákból (pl: [2]) ismerjük, hogy a fa struktúrákon dolgozó algoritmusok - elméleti idő és tárbonyolultság legrosszabb esetének - becslésében, gyakran szerepet játszik az adott fa mélysége.

4.3. Számításigényes matematikai feladatok

A számításigényes metematikai feladatok közül az n-kiránynő problémáját választottam. A feladat az általánosan ismert 8-királynő probléma kiterjesztése, $n \times n$-es táblára. Formálisan, adott $n \times n$-es táblára helyezzünk el n darab figurát úgy, hogy sem egy sorban, sem egy oszolopban, sem valamely átló mentén nem lehet két figura.

4.3.1. Clean környezet diagnosztikájával mért adatok

A munka során a Clean project diagnosztikai lehetőségei közül, a legegyszerűbbel történtek a mérések, mely módszer csak az összesített futási időadatokat adja meg, és jár a legkevesebb beavatkozással a véglegesen generált kódhoz képest.

A feladat megoldására [8] könyv 307. oldalán nyolc különböző funkcionális megoldást ír le. Ezen megoldások mind helyesek, viszont igen nagy mértékben különböznek az erőforrásigényeik. A [8]-ben szereplő adatok a Macintosh Performa 630 gépen, Clean 1.1 környezetre vonatkoznak.

A Clean 2.0.2 fordítóprogram és egy 733MHz-es PC használatával, a következőképpen változtak az egyes megoldások erőforrásigényei:

Execution time in seconds of various versions of the queens program	Old value ($n = 10$) execution	garbage collection	total	New value ($n = 12$) execution	garbage collection	total
1: list comprehensions	39.66	0.47	40.13	10.07	1.13	11.20
2: list comprehensions using safe from 3	9.75	0.05	9.80	10.17	1.16	11.33
3: direct recursion	10.78	0.05	10.83	6.55	0.61	7.16
4: direct recursion using next‘	9.71	0.04	9.75	6.04	0.31	6.35
5: toolbox functions	67.45	1.80	69.25	18.90	7.42	26.32
6: toolbox functions using safe from 3	17.65	0.16	17.81	18.63	6.95	25.58
7: continuations	10.20	0.03	10.23	6.12	0.44	6.56
8: continuations using next‘	9.63	0.02	9.65	5.85	0.37	6.22

A konklúzió levonása előtt meg kell jegyeznem, hogy kettővel meg kellett növelnem a probléma bemeneti paraméterét, mert ($n = 10$) esetén nem futott elég ideig az algoritmus ahhoz, hogy az adatok szemmel jól olvashatóak legyenek. (Az eredeti mérések is elkészültek és a későbbiekben, használom is ezeket az adatokat a statisztikai becsléseknél.) Az n-királynő probléma karakterisztikája miatt, a bemenő paraméter 20%-os növelése a problémateret 1861%-al növelte meg.

A következtetés tehát a különböző sorok arányának változásából vonható le. Nem elfelejtve a [8] cikben leírtakat, miszerint mind a nyolc implementáció helyes, sőt egyformán értékes, mert különböző szemszögből közelítik meg a ugyanazt a problémát.

Konklúzió

Az első és legszembeötlőbb különbség, az egyes-kettes és az ötös-hatos esetekben bekövetkezett arányváltozás. Míg a korábbi verzióban a ***[safe]*** függvény kezelésének optimalizálásával négyszeresére volt gyorsítható a kód, addig az új fordító esetében a különbség elhanyagolható. Ezt valószínűleg, a fordító optimalizációjának jelentős javulása okozza.

4.3.2. Időprofil elemzés

Az időprofil elemzés alapja, hogy a különböző függvényekhez adattárolókat rendelünk, és ezekben kumulálódnak a különböző mérési eredmények. A módszer használhatóságát korlátozza, hogy ($n = 12$) esetén a számlálók túlcsordulnak, ezért a mérések ($n = 10$)-re készültek.

Execution time in seconds of various rious versions of the queens program	Alloc (bytes)	Strict kiértékelés (db)	Lazy	Curry	GC
1: list comprehensions	79’832’636	5’107’028	3’687’738	0	5’205
2: list comprehensions using safe from 3	79’823’468	5’106’206	3’687’312	0	5’193
3: direct recursion	29’334’484	2’090’847	1’823’737	0	1’930
4: direct recursion using next‘	20’978’884	1’393’985	1’475’587	0	1’368
5: toolbox functions	225’244’048	2’939’880	8’725’817	417’779	15’405
6: toolbox functions using safe from 3	217’842’068	2’520’789	8’377’020	417’576	14’764
7: continuations	25’278’592	1’707’597	1’752’661	0	1’645
8: continuations using next‘	20’276’800	1’359’118	1’440’049	0	1’316

Konklúzió

Az ötös-hatos pár esetében, a futási idő növekedést egyrészt a [8] alapján, a Curry féle kiértékelések okozzák, de feltehető egy második, szintén jelentős tényező az adott implementáció memóriaigénye Átlagos esethez képest majd 20-szoros memóriát allokál, ez egyrészt a memória lefoglalásánál jelent komoly többletidőt, másrészt ezen memória felszabadításánál. A szemétgyűjtő algoritmusnak, a megnövekedett memóriaigény kielégítésére, a rendelkezésre álló korlátos (400 Kb) memóriát, 15000-szer kellett végigolvasnia.

5. fejezet

Programkódok és erőforrásigényük összefüggései

Ennek a fejezetnek a célja, hogy összefoglalja és rendszerezze a erőforrásigények vizsgálata közben szerzett tapasztalatokat. Az összefoglalás során egyre általánosabb szabályszerűségek kerülnek ismertetésre.

5.1. Felhasznált erőforrástípusok egymásra hatása

A konkrét példákon elvégzett mérések közben egyre világosabbá vált az a tény, hogy a funkcionális programok által igényelt, különböző típusú erőforrások mennyiségei nem függetlenek. Tekintsünk egy problémaosztályt, és nézzük meg a lehetséges becsléseket. Az előző fejezetben tárgyalt n-királynő probléma legyen a becslésekhez használt adathalmaz.

5.1.1. Kiértékelésszám hatása a memória allokációra

A kiértékelésszám memória allokációra való hatását két aspektusból lehet vizsgálni. Az első durvább becslést a funkcionális program összesített kiértékelésszámából és az összesen allokált memória mennyiségéből kaphatjuk.

A mérési eredmények alapján indokoltnak tekinthető, a kiértékelésszám és a memória allokáció között lineáris összefüggés feltételezése. A feltételezés helytállóságáról lineáris regresszióval [1] lehet megerősítést szerezni.

Vezessük be a queenMEM, queenSTRICT, queenLAZY, queenCURRY kifejezéseket rendre következő mennyiségekre, memória allokációra, valamint a Clean nyelv három féle kiértékelés fajtájára.

Lássuk a regresszió eredményének kivonatát:

```
glm(formula = queenMEM ~ queenSTRICT + queenCURRY)
```

[1] A statisztikai problémák megoldásához az "R" nevű programcsomagot vettem igénybe.

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.207e+06  8.274e+05  -2.668   0.0445 *
queenSTRICT  1.603e+01  2.547e-01  62.946 1.92e-08 ***
queenCURRY   4.309e+02  2.021e+00 213.169 4.31e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 1.068863e+12)

    Null deviance: 5.2279e+16  on 7  degrees of freedom
Residual deviance: 5.3443e+12  on 5  degrees of freedom
```

A regresszió készítésekor kiderült, hogy a memória allokáció kérdésében a lusta kiértékelések száma nem szignifikáns. Ennek megfelelően a várható memóriaigény becslésére a következőt érdemes választani:

$$\hat{T}_{queenMEM} = 16.03 * T_{queenSTRICT} + 430.9 * T_{queenCURRY} - 2207000$$

Sikeresnek nevezhető ez a lineáris regresszióval előállított eredmény, mivel a queenSTRICT és queenCURRY kifejezések szignifikanciája kicsi valamint, hogy a queenMEM szórása a lineáris becslés után 4 nagyságrendel csökkent, az eredeti eloszlásás szórásához képest.

Ezt a módszert durva becslésnek kell azonban tekinteni, mivel a funkcionális kifejezés különböző részkifejezései jelentős különbségeket mutat(hat)nak a kiértékelésszám, és az allokált memória mennyisége között.

5.1.2. Kiértékelésszám hatása a futási időre

A kiértékelésszám nyilvánvalóan szerepet játszik a függvény kiértékeléséhez szükséges időben (röviden: futási idő). Az allokált memória becsléséhez hasonlóan elkészült a következő regressziós becslés:

A korábbi queenMEM, queenSTRICT, queenLAZY, queenCURRY kifejezések mellé, vegyük föl a queenTIME kifejezést, mely az n-királynő problémát megoldó megoldások összesített futási idejét tartalmazza. A mérés során keletkezett mérési eredmények közül, véletlenszerűen kerültek ki a becsléshez használt adatok (A mérések egy előre nem definiált számszor kerültek végrehajtásra).

Lássuk a regresszió eredményének kivonatát:

```
glm(formula = quuenTIME ~ queenSTRICT + queenCURRY)

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.327e+02  4.621e+00   28.73 9.58e-07 ***
queenSTRICT 4.417e-05  1.423e-06   31.05 6.50e-07 ***
queenCURRY  1.070e-03  1.129e-05   94.76 2.48e-09 ***
---
```

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 33.34410)

    Null deviance: 328050.53  on 7  degrees of freedom
Residual deviance:    166.72  on 5  degrees of freedom
```

A becslés elkészítésekor megismétlődött a memória allokáció vizsgálatakor tapasztalt jelenség, miszerint az összesített futási idő meghatározásában nem játszik jelentős szerepet a lusta kiértékelések száma. Ami viszont meglepetésnek tekinthető, hogy az allokált memória mennyisége sem, legalább is a szigorú és a curry kiértékelésekhez képest. Ennek megfelelően az összesített futási idő a következő képlettel becsülhető:

$$\hat{T}_{quuenTIME} = 2.503 * 10^{-6} * T_{queenSTRICT} + 1.070 * 10^{-3} * T_{queenCURRY} + 148.6$$

Sikeresnek nevezhető ez a lineáris regresszióval előállított eredmény, mivel a queenSTRICT és queenCURRY kiffejezések szignifikanciája kicsi valamint, hogy a quuenTIME szórása a lineáris becslés után 4 nagyságrendel csökkent, az eredeti eloszlásás szórásához képest. Meg kell jegyezni azonban, hogy a quuenTIME kezdeti szórása messze elmarad a queenMEM kezdeti szórásától.

5.1.3. Allokált memória mennyiségének hatása a futási időre

A fenti két becslésen kívül elkészült a futási idő memória allokációval történő becslése is. A becslés nem tartalmaz túl sok ujdonságot, ezért csak az eredményét közlöm:

$$\hat{T}_{quuenTIME} = 4.417 * 10^{-5} * T_{queenMEM} + 132.7$$

A quuenTIME ilyen alakú becslésének szórása 504.63 magasabb az előző becslésben szereplő értékénél.

5.2. Köztes kódok vizsgálata

Egy funkcionális nyelven írt programkód által igényelt erőforrásigényről túl nehéz az eredeti kód alapján megfelelő becslést adni, mivel a következő tényezők nehezítik a feladatot:

- Igen nagy kifejezőerő
- Bonyolult nyelvi szerkezetek

- Lusta és szigorú kiértékelés lehetősége
- Curry-függvények

A köztes kódok, ezzel szemben, rendelkeznek néhány az elemzést megkönnyítő, ezáltal számunkra hasznos tulajdonsággal:

- Eredeti forráskóddal való ekvivalencia (*feltételezhető*)
- Egyszerű nyelvi elemek
- Egyszerűbb nyelvi szerkezetek
  - Szekvencia
  - (Feltételes) ugrások

5.2.1. A köztes kód (ABC) viszonya a forráskódhoz

A köztes kódok vizsgálata során fény derült a fordítóprogram azon kellemes tulajdonságára, hogy az általa generált köztes kódok "folytonosnak" tekinthetők. Ezt a folytonossági tulajdonságot úgy lehet a legszemléletesebben körülírni, ha azt mondjuk, hogy a forrás kód kis változása a köztes kódban csak kis változást indukál. A változások az ABC kódban általában szemmel nem láthatóak, de a különböző verziók, programmal (pl: verzió kezelő segédprogrammal) való összehasonlítása feltárja a különbségeket.

Vannak olyan elemek, amelyek minden köztes kódban változnak. Ilyen a modul verziószámát helyetesítő dátum mező. Más elemek, például konstansok értékének változtatása az ABC kódban is csak egy konstans értéket változtat. Egy újabb könyvtári függvény aktivizálása ***[import]***, a köztes kódban néhány újabb cimke megjelenésével jár. Egy függvény lecserélése ennél már több változást eredményez, de ezek a változtatások lokálisak, mivel csak a meghívott függvény neve (az ugrás cimkéje), és a paraméterátadás változik, de ez utóbbi csak akkor, ha nem ugyanaz a eredeti és az új függvény szignatúrája.

A kiértékelés sorrendjét megváltoztató függvénycserék (például más precedencia szintű operátorok használata miatt), ennél is komolyabb változásokat idéznek elő a köztes kódban. A változások, kódrészletek felcserélődése, új kódrészletek mejelenése, régebbiek kimaradása lehetnek. Ebben az esetben is igaz, hogy általában a változások az ABC kód csak egy-egy szakaszát érintik.

5.2.2. Implementációs "stílusok" hatása a köztes kódra

Az implementációs stílusok bemutatása az n-királynő problémán [8] keresztül történt. Az implementációs stílusok négy nagyobb csoportba sorolhatóak, melyekben két-két egymásra erősen hasonlító megoldás található. Az egy

csoporton belüli megoldások tulajdonképpen egy további érték vizsgálatával optimalizálják a program tulajdonságait. A köztes kódban ez mint feltételvizsgálat és egy a speciális esetet kezelő új kódrészlet jelenik meg.

A csoportok között viszont jelentős különbségek fedezhetők fel. Mivel a lexikális hasonlóság a különböző forráskódok között sem túl nagy, ezért a köztes kódok esetében is csak kisebb mértékű a hasonlóság. Vannak olyan változatok, például az első, melyek esetében a forráskód megírásakor is cél volt a maximális érthetőség, ennek a változatnak a köztes kódjára is azt lehet mondani, hogy világos szép szerkezetű. Az ellentétes véglet az ötödik változat, ahol a tervezés célja a lehető legáltalánosabb kód megírása volt. Az ebből a változatból létrejövő ABC kód, ennek megfelelően, meglehetősen bonyolult és csak maradványaiban emlékeztet a más változatokban látottakra.

5.3. Várható erőforrásigény kód alapján történő becslése

5.3.1. Becslések alapja

A becslés alapját az az általánosítás szolgáltatja, hogy egy bizonyos kódrészlet ugyanolyan körülmények között, mindig ugyanannyi erőforrást igényel. Ezen megállapítás szerint a kódrészletekre meg lehet határozni egy futási körülményektől (továbbiakban környezettől) függő függvényt, amely az adott kódrészlet erőforrásigényét írja le.

A becslés ezek szerint kódrészletekre vonatkozik, tehát előállítása után a kódrészletre hivatkozó egyéb kódok erőforrásigényének kiszámításakor már felhasználható. Meg kell említeni a rekurzív függvények kérdését. Ebben az esetben a becslés meghatározása csak rekurzívan, általános esetben, a becslés egy más környezetbeli értékének kiszámítása után, lehetséges. A becslések rekurziója az eredeti kódrészlet rekurziójának megfelelően értékelhető ki.

Például tekintsük a következő rekurzív függvényt:

```
module fact

import StdInt

fact 0 = 1
fact n = n * fact (n - 1)

Start = fact 3
```

A program erőforrásigényét (R) a következő elemekből lehet kiszámolni.

$$\begin{aligned} R &= R_{fact}(n=3) \\ &= R_* + R_- + R_{fact}(n=2) \\ &= R_* + R_- + (R_* + R_- + R_{fact}(n=1)) \\ &= R_* + R_- + (R_* + R_- + (R_* + R_- + R_{fact}(n=0))) \\ &= R_* + R_- + (R_* + R_- + (R_* + R_- + 0)) \qquad mivel, (R_1 \sim 0) \end{aligned}$$

A számításban kétféle összetevő szerepel. Az egyszerűbbek, a R_* (amely $*$ operátor erőforrásigénye) alakú összetevők , melyek paramétereiktől nem függő erőforrásigényük van, röviden paraméterfüggetlen erőforrás-komponensek. Jellegzetesen ilyenek a δ-redukciók, melyek primitív típusokon végeznek műveleteket. A másik csoportba tartoznak a $R_{fact}(n=3)$ alakú összetevők, melyek a ***[fact]*** függvény erőforrásigényét jelölik $(n=3)$, peremfeltétel mellett. Ezek a paraméterfüggő erőforrásigény-komponensek.

Ennek megfelelően a ***[fact]*** függvényre a következő általános képlet adható:

$$R_{fact}(n) = \begin{cases} R_* + R_- + R_{fact}(n-1), & \text{ha } n > 0 \\ 0, & \text{különben} \end{cases}$$

Ezt zárt alakra hozva:

$$R_{fact}(n) = n * (R_* + R_-)$$

5.3.2. Becslések módszere

A becslés fenti módszere direkten nem alkalmazható, mivel ebben az esetben a függvény kiértékelését, minden bemenő paraméterre meg kellene tenni, ami persze nem elfogadható. A problémát kicsit jobban megfigyelve látható, hogy a ***[fact]*** függvény mintaillesztése okozza a felesleges izgalmakat, ugyanis a ***[fact]*** kiértékelésekor, a két különböző mintának megfelelően, két különböző erőforrásigényt lehet tapasztalni.

Amikor a továbbiakban a várható erőforrásigény becsléséről lesz szó, a ”kód”-on a köztes ABC kódot kell érteni, az ettől való esetleges eltérések külön jelzésre kerülnek.

A fenn található ***[fact]*** függvény ABC kódjának részletét (lásd a 50. oldalon) /az ABC kód eleje hiányzik/ megvizsgálva, a következő megállapítások tehetők.

Az elemzés céljából a legmegfelelőbb a kódot kódszekvenciákra kell bontani. Ezt a felbontást az ABC nyelv definiciója segítségével tehetjük meg. A

felbontás alapjai legyenek, a cimkék és a vezérlési szerkezetek által létesített határok.

Az egyes kódszekvenciákban az erőforrásigény felírható rekurzív formában a következőképpen. Minden ABC nyelvbeli utasításra, kivéve a vezérlésátadási utasításokat, meghatározható az általa igényelt erőforrásigény. Ez utóbbiaknak viszont saját erőforrásigényükön kívül, a vezérlést megkapó kódrészlet erőforrásigényével is kell számolniuk.

A **fact.abc** kódot a következő kódrészletekre lehet bontani (kezdetüket jelző cimkék szerint):

- __fact_Start
- n2
- ea2
- s2
- s1
- case.1
- case.2

Az absztrakt ABC gép [10] modellje szerint, a követktkező összetett állapottal bír:

$$state == (astack, bstack, cstack, graphstore, descrstore, pc, programstore, io)$$

Vegyük az első, *__fact_Start* névvel jelzett, kódrészletet. Ennek erőforrásigénye:

$$R_{__fact_Start} = R_{[build]} + R_{n2} + R_{[jmp]} + R_{_driver}$$

$$R_{n2} = R_{[push_node]} + R_{[jsr]} + R_{ea2} + R_{[fillI_b]} + R_{[pop_b]}$$

$$R_{ea2} = R_{s2}$$

$$R_{s2} = R_{[pushI]} + R_{[jmp]} + R_{s1}$$

Az első érdekes pontja az erőforrásigények meghatározásának, az *s1* nevű kódrészlet, mivel ennek két féle erőforrásigénye lehet, az *equI_b* utasítás eredményének megfelelően.

$$R_{s1} = R_{[equI_b]} + R_{[jmp_true]} + R_{case.1}$$

$$R_{s1} = R_{[equI_b]} + R_{[jmp_true]} + R_{[jmp]} + R_{case.2}$$

A fenti probléma megoldásához szükséges volt az ABC állapotának ***[state]***, bevezetésére. Ugyanis a kódrészletek ezt az állapotot változtatják meg, illetve ennek függvényében változik a tevékenységük (erőforrásigényük). Az egyes kódrészletek környezethez (***[state]***)-hez való viszonyát a következőképpen lehet osztályozni. A köztük levő viszonyt, egyszerűsítve függőségnek hívjuk:

- B (argumentum) veremtől függő kódrészeletek
  - Csak a verem legfelső elemétől függőek
  - A verem legfelső legfeljebb n elemétől függőek
- A gráftól függő kódrészeletek.

Tisztán az argumentum veremtől függő kódrészletekből áll a fenti *fact* függvény, míg tisztán a gráftól függő kódrészletek találhatóak a *Clean* beépített *last* függvényében. Általános esetben az egy függvényhez tartozó kódrészletek, mind a veremtől, mind a gráftól függenek.

Ezen összefüggések ismeretét a tételbizonyítók is igénylik. Ennek megfelelően az erőforrásigény becslését egy kiterjesztett verifikációs eszköz keretén belül érdemes megvalósítani. Az erőforrásbecslés és a tételbizonyító integrációjának eredményeképpen, az erőforrásigény becslése, a tételbizonyító futási idejét (erőforrás igényét) csak a kódrészletek számának megfelelően, lineárisan növeli.

Egyes kódrészletek erőforrásigényének rekurzív becslése alapján, viszonylag hatékonyan kiszámolható konkrét esetek erőforrásigénye. Például a vizsgált függvény kiértékelése elött, a függvény kiértékeléséhez képest elhanyagolható erőforrásigényű rekurziós képlettel, megbecsülve a várható erőforrásigényt, a futtató rendszer dönthet az adott függvény kiértékelése mellett, vagy elutasíthatja azt.

Egy, az elöbbi módszernél szebb megoldást kapunk, ha a keletkezett rekurzív képletet zárt alakra tudjuk hozni. Az ilyen rekurzív függvények megoldására a [4] és [5] alapján a készlet-módszerrel (lineáris-rekurzió esetében), matematikai programcsomaggal, illetve valamilyen matematikai adatbázisban való kereséssel kerülhet sor. Természetesen elképzelhető olyan függvény, amelyre az alábbi módszer nem jár eredménnyel, ekkor az erőforrásigény felső becslése lehet a megoldás.

A. Függelék

Forrásként használt kódrészletek

Az alábbi fejezet olyan teljes forrásálományokat és kódtöredékeket tartalmaz melyekről úgy éreztem, hogy megkönnyíthetik az egyes részek megértését. A forrásként használt kódrészletekben a függvények valamint a minták nevét angol nyelven adom meg. Ez remélhetőleg nem befolyásolja az érthetőséget, viszont javítja majd az olvashatóságot.

A.1. Rendező algoritmusok

```
Merge :: [a] [a] -> [a] | Ord a
Merge [] ys = ys
Merge xs [] = xs
Merge [x:xs] [y:ys]
  | x <= y     = [x: Merge xs [y:ys]]
  | otherwise  = [y: Merge [x:xs] ys]

msort :: [a] -> [a] | Ord a
msort xs
  | len <=1   = xs
  | otherwise = Merge (msort ys) (msort zs)
 where
    ys = take half xs
    zs = drop half xs
    half = len/2
    len = length xs
```

MergeSort.icl

```
Insert :: a [a] -> [a] | Ord a
Insert e [] = [e]
Insert e [x:xs]
   | e<=x      = [e,x:xs]
   | otherwise = [x: Insert e xs]

isort :: [a] -> [a] | Ord a
```

```
isort [] = []
isort [a:x] = Insert a (isort x)
```

InsertSort.icl

```
qsort :: [a] -> [a] | Ord a
qsort [] = []
qsort [a:xs] = qsort [x \\ x <- xs | x < a] ++ [a]
               ++ qsort [x \\ x <-xs | x > a]
```

QuickSort.icl

A.2. ABC kódok

```
__fact_Start
      build _ 0 n2
.d 1 0
     jmp _driver
.n 0 _
.o 1 0
n2
     push_node _cycle_in_spine 0
.d 0 0
     jsr ea2
.o 0 1 i
     fillI_b 0 0
     pop_b 1
.d 1 0
     rtn
.o 0 0
ea2
.o 0 0
s2
     pushI 3
.d 0 1 i
     jmp s1
.o 0 1 i
s1
     eqI_b 0 0
     jmp_true case.1
     jmp case.2
case.1
     pop_b 1
     pushI 1
.d 0 1 i
     rtn
case.2
     pushI 1
     push_b 1
     subI
```

```
.d 0 1 i
     jsr s1
.o 0 1 i
     push_b 1
     update_b 1 2
     updatepop_b 0 1
     mulI
.d 0 1 i
     rtn
```

fact.abc

Irodalomjegyzék

[1] Chris Clack, Stuart Clayman, David Parott, *Lexical Profiling: Theory and Practice*, Cambridge University Press 1993

[2] Thomas H. Cormen, Charles E. Leierson, Ronald L. Rivest, *Algoritmusok*, Műszaki Könyvkiadó 1997

[3] Zoltán Csörnyei, *Funckionális programnyelvek implementációja, 1. rész A λ-kalkulus*, Budapest 2003

[4] Ronald L. Graham, Donald E. Knuth, Oren Patashnik, *Konkrét matematika, 1. rész Rekurziós problémák*, Műszaki könyvkiadó 1998

[5] Ronald L. Graham, Donald E. Knuth, Oren Patashnik, *Konkrét matematika, 2. rész Összegek*, Műszaki könyvkiadó 1998

[6] John van Groningen, Rinus Plasmeijer *Strict and Unboxed Lists using Type Constructor Classes in a Lazy Functional Language*, 2001

[7] Zoltán Horváth, *Concepts of Modern Functional Programming Languages*, 2002 Draft

[8] Pieter Koopman, Rinus Plasmeijer, Marko van Eekelen, Sjaak Smetsers, *Functional Programming in CLEAN*, 2001 Draft

[9] Lovász László, *Algoritmusok bonyolultsága*, Budapest 1992

[10] Rinus Plasmeijer, Marko van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley 1993

[11] Rinus Plasmeijer, Marko van Eekelen, *Clean Version 2.0 Language Report*, Draft 2001

[12] Patrick M. Sansom, Simon L. Peyton Jones, *Profiling Lazy Functional Programs*, 1992

[13] Patrick M. Sansom, Simon L. Peyton Jones, *Formally Based Profiling for Higher-Order Functional Languages*, 1997

[14] Ian Sommerville, *Szoftver rendszerek fejlesztése*, Panem 2002