Szemétgyűjtő algoritmusok ismertetése, gyakorlati szemétgyűjtési feladat megoldása

diplomamunka

az Eötvös Loránd Tudományegyetem
Természettudományi Karának
programtervező-matematikus szakára
2003

Témavezető:

Dr. Horváth Zoltán
hz@inf.elte.hu

Készítette:

Ivicsics Mátyás
mathiasr@inf.elte.hu

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

A Nijmegeni Egyetemen, Hollandiában fejlesztik a Clean funkcionális programozási nyelvet. A fejlesztést végzők egy csoportja a dinamikus programszerkesztés megteremtésén dolgozik, és ösztöndíjas hallgatóként ebbe a munkába volt alkalmam bekapcsolódni. A fejlesztés során szemétgyűjtési kérdésbe ütköztek, ennek megoldása volt fő feladatom. A probléma megoldása megfelelően sikerült, az általam elkészített program feladatának megfelelően működik.

Diplomamunkámban az elvégzett munka kapcsán szemétgyűjtő algoritmusokkal foglalkozom, és ismertetem a felmerült feladat megoldásának részleteit.

Bevezetés

A dinamikus programszerkesztés lényege, hogy a programok megírt és lefordított kódot tudnak valamiféle fájlként a háttértárra menteni, majd az így elmentett mobil kódot később más programok futásidőben betölthetik és végre hajthatják. A Clean csoport ennek a nyelvi lehetőségnek a megteremtését is megcélozta munkájával. A cél megvalósítása közben felmerült fájlok közötti szemétgyűjtési problémára kiegészítő program elkészítése nyújtott megoldást.

Az első fejezetben a szemétgyűjtés problémakörének miértjével, és fontosságának megvilágításával foglalkozom, valamint ismertetem a kínálkozó megoldások alapötleteit. A második fejezet a probléma megoldására kidolgozott alapvető algoritmusokat tárgyalja, a harmadik és negyedik fejezet pedig ezen algoritmusok további ötletek és elvárások alapján való optimalizációját keresi. Az ötödik fejezetben tárgyalom a Clean-ben megvalósított dinamikus programszerkesztés elveit és működésének részleteit, a hatodik fejezetben pedig rávilágítok, hogy ennek kapcsán hogyan került elő a kérdéses szemétgyűjtési probléma, és milyen megoldás született.

1. A szemétgyűjtés

1.1. Bekódolt szemétgyűjtés hátrányai

Az korai imperatív programozási nyelvek többségében a programozó saját felelősségeként jelentkezik a feleslegessé vált tárterület felszabadítása a memóriában. Az ilyen nyelvekben egy-egy elmaradt „free” vagy „dispose” utasítás idővel okozhatja a program számára lefoglalt memóriaterület beteltét, a kelleténél korábban való tárterület felszabadítás pedig igen kényelmetlenül felderíthető futási idejű hibákhoz vezethet. Az ilyen jellegű mulasztások vagy átgondolatlanságok felderítésének nehézsége abban rejlik, hogy az észlelt hibák nem determinisztikusan jelentkeznek, hanem esetenként mutatkoznak, máskor pedig nem. Gyakran előfordul, hogy a hiba a program tesztelése során egyáltalán nem okoz problémát a tesztelés sajátságos körülményei miatt. Képzeljük el azt a helyzetet, amikor egy memóriaterületet korábban felszabadítunk a kelleténél és esetleg van máshonnan az ott tárolt objektumra vonatkozó használatos hivatkozás a programban. Még ha használjuk is a továbbiakban az objektumot –akár olvashatjuk vagy írhatjuk is– a hiba mindaddig nem okoz problémát amíg az illető tárterület újra lefoglalásra nem kerül más célra – ha viszont ez megtörténik, akkor az adott terület felhasználása *párhuzamosan* egész más célra is megkezdődik. Ezen fatális hiba jelentkezése például csak az aktuális futtatási környezettől függ.

További hátránya a moduláris programozásban a programból történő explicit tárkezelésnek az, hogy a program logikája által esetleg meg nem kívánt függőségek is összekötik a különböző modulokat. Ha egy függvény vagy eljárás felszabadít egy memóriaterületet, fontos az, hogy a programban a továbbiakban sehonnan senki ne használja az ott tárolt objektumot. Minthogy az objektumok felszabadíthatósága ezáltal lokálisan nem lekezelhető *globális* kérdéssé vált, valamiféle globális nyilvántartást kell karbantartani annak eldönthetőségére, hogy adott objektumokra kik tartanak még igényt. Ezen globális nyilvántartás vezetése logikailag független kódrészek összehangolását igényli, ezzel csökkentve a program karbantarthatóságát, bővítését. Ugyanezen probléma felmerül konkurens alkalmazások esetén is.

1.2. Megoldási lehetőségek

A fenti problémák megoldásának többféle megközelítése létezik. Ha egy explicit tárterület felszabadítással elkészített program elfogyasztja a memóriát mert valahol rendszeresen elmarad a felszabadító utasítás, megpróbálkozhat a programozó olyan eszközökkel, melyeket pontosan ilyen hibák felderítésére készítettek. Ha megtalálta, hogy mely adatstruktúrák felszabadítása nem történik meg, ismét elgondolkodhat, hogy hol lenne a programban a megfelelő hely a felszabadító utasítás kiadására. Az is elképzelhető, hogy a program struktúrájából adódóan ilyen hely nem létezik, ekkor át kell strukturálni a programot. Ezen megközelítésnek és ilyen hibakereső eszközök alkalmazásának legnagyobb problémája az, hogy konkrét futás közbeni hibákat derít fel, így esetleges tervezési hibák amik nem jelentkeznek a tesztelés közben, rejtve maradnak.

Egyes programokhoz saját szemétgyűjtő modul készül. Ez a megoldás tulajdonképpen már nem sokban különbözik a programozási nyelvbe integrált szemétgyűjtőtől, de még mindig fennállhat a probléma, hogy mivel egyetlen alkalmazáshoz készül általában hibás és nem teljes, valamint nehezen használható.

A legelterjedtebb megoldás a nyelvbe integrált automatikus szemétgyűjtés. Ekkor a memóriaterület foglaló eljárás az, amelyik nem kielégíthető memóriakérés esetén speciális utasításokat hajt végre terület felszabadítás céljából. Felesleges tehát a továbbiakban memóriaterület felszabadítást kódolni a programba, ezt a foglaló eljárásunk szükség esetén elvégzi – rendszerint a szemétgyűjtő eljárás meghívásával.

1.3. Automatikus szemétgyűjtés

Az automatikus szemétgyűjtés lényege az, hogy a futó program által korábban lefoglalt, de a továbbiakban használni nem kívánt tárterület újra szabaddá váljon – hogy ismét felhasználható legyen. A szemétgyűjtő algoritmus felelőssége tehát kétoldalú: gondoskodnia kell arról, hogy a szemét által lefoglalt terület minél hamarabb szabaddá váljon, ugyanakkor biztosítania kell, hogy a még felhasználni kívánt objektumok által lefoglalt terület védve legyen.

A szemétgyűjtés megléte nagyban elősegíti a programozók munkáját a modern nyelvekben. Minthogy a programok a továbbiakban mentesek a tárterület felszabadító kódtól, absztraktabbak, könnyebben karbantarthatóak, bővíthetőek és nem utolsó sorban kevesebb a hibalehetőség.

Működési alapötlet

Nehéz ugyan automatikusan kideríteni azt, hogy egy memóriában tárolt adatobjektumra mikor történt az utolsó hivatkozás vagy hogy lesz-e még a jövőben rá hivatkozás, de azt biztosan tudhatjuk, hogy ha az illető objektumot a programból már semmilyen mutatóbejárással nem érjük el akkor a további hivatkozások elvi lehetősége is ki van zárva. Ezen az alapelven működnek az automatikus szemétgyűjtők: azokat az objektumokat tekintik szükségesnek, amik elérhetőek a futó alkalmazásból, minden mást pedig feleslegesnek nyilvánítanak.

Egy szemétgyűjtő eljárás feladata alapvetően két részre osztható. Először a szemét felderítését, majd a tárterület felszabadítását kell valahogy megoldani. A két feladatrész a futás során nem biztos, hogy elkülönítve hajtódik végre – átlapolhatják egymást.

Egy szemétgyűjtő tulajdonképpen rendszerint azt deríti fel, hogy mi az ami *szükséges*, majd az ezen kritériumot nem teljesítő objektumokat tekinti szemétnek. A szükségesség kritériumának definíciója általában a következő: tekintsük gyökérnek a statikusan lefoglalt globális változókat, a program vermén aktuálisan tárolt változókat és a regiszterekben tárolt változókat. Azok a memóriában tárolt objektumok melyek ezekből a változókból elérhetőek, szükségesek, valamit egy szükséges objektumból tetszőleges irányított úton elérhető objektumok szintén szükségesek. Azon objektumok melyek nem felelnek meg a definíciónak, a program számára a továbbiakban semmilyen (legális) úton nem elérhetők sem írásra, sem olvasásra – tehát szemétnek tekinthetők.

Vegyük észre, hogy a feltétel tág. Attól, hogy a program számára még elérhető egy objektum, nem biztos, hogy azt használni is fogja még a futás során – elképzelhető, hogy nincs is olyan végrehajtási útja, amiben még használhatná. A szükségesség a szemétgyűjtő algoritmus számára információ hiányában mégsem definiálható szűkebben, mivel a program végrehajtási útjairól a memóriakezelés szintjén nem tudhatunk.

A következő fejezetekben nagyjából áttekintjük az automatikus szemétgyűjtés elvégzésére kidolgozott algoritmusokat. Aki esetleg az itt tárgyaltnál részletesebben kíváncsi a témára, annak az irodalomjegyzékben szereplő doktori munkának [1] első és harmadik fejezetét, aki pedig az általam egyáltalán nem ismertetett osztott szemétgyűjtés problémájának megoldására kifejlesztett algoritmusok iránt érdeklődik, annak a [2] irodalmat ajánlom figyelmébe.

2. Alapvető szemétgyűjtő algoritmusok

A szükséges objektumok felderítése –vagyis a szemétgyűjtés első fázisa– alapvetően két technikával végezhető el. Az egyik módszer a *hivatkozás számlálás*. Lényege az, hogy a program futása során folyamatosan karbantartunk minden objektumhoz egy számláló értéket arról, hogy hányan hivatkoznak rá a programból. A másik: a *bejárás* módszere a szükségesség definíciója szerint halad, vagyis bejárja mindazokat a hivatkozási utakat melyeken a program végigmehet, és minden elért objektumot szükségesnek tekint.

2.1. Hivatkozás számlálás

Az algoritmus párhuzamosan működik a program futásával. Minden objektumhoz karban kell tartania egy számlálót. Ha egy mutatót hoznak létre létező objektumra akkor növeli eggyel az adott objektum hivatkozási számlálóját, ha törölnek egy mutatót akkor pedig csökkenti. Mindazok az adatobjektumok váltak szükségtelenné, amelyek hivatkozási számlálója nulla – ezeket a program a továbbiakban képtelen elérni, mert egyetlen változója sem hivatkozik rájuk. Amikor egy objektumot megszűntetünk –mert a számlálója elérte a nullát– mindazon objektumok számlálóját is csökkentjük eggyel, amelyekre a megszüntetendő valamiféleképpen hivatkozott.

A számlálót rendszerint az objektumhoz rendelt *header* információk között tárolják. Ezek az információk csak rendszer szinten láthatóak, programozás közben nem.

A módszer előnye, hogy a program végrehajtásával együtt működik, vagyis nincsen szükség a program futásának alkalmankénti hosszú megszakítására ahhoz, hogy szemétgyűjtéseket végezzünk, hanem az elérhetetlenné vált objektumok által foglalt területet tulajdonképpen azonnal felszabadítjuk. Érdemes azonban látni, hogy ez az objektumok bonyolultságától is függ, hiszen ha egy fa struktúra gyökerének hivatkozási számlálója eléri a nullát akkor egymás után az egész fában csökkenteni kell a számlálókat, majd felszabadítani a területeket – és így persze már tekintélyes mértékben megakadhat a program futása. A probléma orvosolásaként létrejöttek olyan hivatkozás számlálási elven működő szemétgyűjtők, melyek nagyobb adatstruktúrák felszabadítását nem azonnal, hanem adagonként végzik el. Ezek számon tartanak egy plusz listát arról, hogy mely objektumok azok, amelyeknek a hivatkozási számlálója már nulla, de tárterületük még nem került felszabadításra. Az ilyen módon kiegészített algoritmus már akár valós idejű követelményeket is kielégíthet, vagyis biztosíthatjuk, hogy bizonyos időperiódus alatt a program bizonyos számú utasítását a processzor garantáltan elvégezze, akárhogy is áll a szemétgyűjtés folyamata.

Jelentős probléma azonban a módszer elvében az, hogy a ciklikusan egymásra hivatkozó ám a program számára már elérhetetlen adatstruktúrákat nem ismeri fel szemétként, hiszen a hivatkozási számlálójuk egymás révén nem nulla, és nem is válik már soha nullává. A hiba elvi gyökere az, hogy amíg ha egy objektumra senki nem hivatkozik az nyilván szemét, a fordítottja ennek nem igaz.

A probléma fatálisnak tűnik, mégis vannak olyan rendszerek melyek a hivatkozás számláló algoritmust használják. A hiba kijavítására vagy egy háttérben működő bejárásos elven működő szemétgyűjtőt alkalmaznak –amelyik felfedezi és összegyűjti az ilyen struktúrákat is–, vagy a programozási stílusból próbálják a ciklikus adatstruktúrákat kiszorítani – esetleg szemétté válásuk előtt aciklikussá tenni. Az előbbi megoldás a korábban előnyként említett valós idejű tulajdonságokat ássa alá –hiszen a programnak várnia kell amíg a bejárásos szemétgyűjtő tetemes időigénnyel lefut–, az utóbbi pedig egy szinttel feljebb: a programozásban okoz megszorításokat ami nehezen elfogadható tulajdonság.

Némi átgondolást igényel a számlálók tárolása is, ezek ugyanis helyet foglalnak a memóriában, jelentősen növelve a memóriaigényét egy hivatkozás számláló algoritmussal dolgozó szemétgyűjtőnek a bejárást alkalmazókkal szemben. Egyes szemétgyűjtők egy teljes gépi szóban tartják nyilván egy objektum számlálóját. Ilyenkor biztosan tudnak ábrázolni tetszőleges számú gyakorlatban elképzelhető hivatkozást, azonban az esetek nagy részében ekkora tárhelyre nincsen szükség. A memóriaigény csökkentését célozzák meg azok a szemétgyűjtők, amelyek kisebb tárhelyen ábrázolják a számlálót, azonban előfordulhat, hogy ezek a számlálók túlcsordulnak. Mivel a biztonságot ilyenkor is garantálni kell, a túlcsordulni készülő számlálót nem növelik tovább és nem is csökkentik a továbbiakban, hanem rögzítik a maximális ábrázolható értéken. A számlálóhoz tartozó objektum sohasem kerül az algoritmus által eltakarításra, erről esetleg egy háttérben működő alkalmanként lefutó bejárásos módszerű szemétgyűjtő tud gondoskodni – amely megoldás mint már láttuk más szempontból is üdvös lehet.

A hivatkozás számlálás hatékonyságának átgondolásakor kétféle számláló-karbantartási költségre kell gondolnunk. Az egyik, hogy mutatók létrehozásakor és törlésekor számláló-növelésre illetve -csökkentésre van szükség, a másik, hogy egy mutató típusú változó értékének módosításakor egyszerre egy számláló-növelést és egy csökkentést is végre kell hajtani. A módszer időigénye tulajdonképpen a program által elvégzett hivatkozás módosító utasítások számával arányos, hiszen a számláló-karbantartás ezekhez kapcsolódó feladat. Vannak olyan esetek, amikor egy-egy hivatkozás igen rövid életű. Ha például meghívunk egy függvényt ami hamar eredményt ad –mert csupán néhány utasításból áll– a neki átadott paraméterekhez hivatkozásokat gyártunk, melyek végrehajtás után el is tűnnek. Ilyen esetben az adott objektum számlálóját megnöveljük, majd néhány utasítás múlva lecsökkentjük. Mivel ez gyakran előfordul egy program lefutása során, a szituációból adódó időigény jelentős. Ezen felesleges időigény a hivatkozás számlálás algoritmusának enyhe variálásával kizárható. Minthogy a rövid életű hivatkozások zöme a program vermén jön létre, csak olyan hivatkozásokat számlálunk amely egy a memóriában tárolt adatobjektumból ered. A módosított szemétgyűjtőt már nem „csapják be" a rövid életű hivatkozások és ezzel időigénye számottevően csökken, viszont oda kell figyelni arra, hogy mit tekintünk szemétnek. Attól, hogy valaminek a hivatkozási számlálója nulla még nem eltakarítható, csak ha a veremből sem hivatkozik rá senki. Ezen algoritmusok rendszerint úgy működnek, hogy időközönként átfésülik a vermet, hivatkozást keresvén a már nulla hivatkozási számlálóval rendelkező objektumokra – amire onnan sem találnak hivatkozást, azt eltakarítják.

A hivatkozás számlálás módszere a ciklikus adatstruktúrák problémája és a költségesség miatt nem túl népszerű, de vitathatatlan előnyei miatt –program futásával párhuzamos működés, a szemét közel azonnali eltakarítása– egyes helyeken mégis használatos. Érdemes megjegyezni, hogy tisztán funkcionális nyelvekben –ahol ciklikus adatstruktúrák létrehozása nem is lehetséges– a fő kifogás a hivatkozás számlálás ellen egyáltalán nem probléma, a számlálók nyilvántartása pedig a szemétgyűjtésen kívül is előnyös olyan rendszerekben, ahol a számlálók hasznosak más szempontból is – például a „csupán egy helyről hivatkozott" (*unique*) tulajdonság ellenőrzésére.

2.2. Egyszerű bejáró algoritmus

Az egyszerű bejáró algoritmus futása során a szükséges objektumok felderítése és a szemét eltakarítása két különálló részt alkot. A bejárás egy bizonyos gyökérhalmaz objektumaiból indul, lehet szélességi vagy mélységi, az érintett objektumok pedig megjegyzésre kerülnek. Ez utóbbi momentum megvalósítható táblázattal, vagy esetleg a megjelölni kívánt objektumokon belül erre szolgáló bitek állítgatásával; és a szükségesség megjegyzése mellett a ciklikus struktúrák többszörös bejárásának elkerülésére szolgál. A szemét eltakarítása érdekében végignézzük a memóriában az objektumokat és ami nincs megjegyezve mint szükséges, annak a területét a szabad területeket összefogó listába fűzzük. Ebből a listából választunk majd szabad memóriaterületet amikor memória igénylésre kerül sor.

Az egyszerű bejáró algoritmus nem túl komplex, elvében hibátlan módszer. Bejárásos mivoltának köszönhetően felderíti a ciklikus adatstruktúrákat is, megvalósítása akár szélességi akár mélységi kereséssel nem bonyolult feladat. Átgondolandó viszont a memória töredezettségének kérdése. Minthogy a tárolt objektumok mérete változatos, a felszabadított memóriaterületek mérete szintén az. Amikor a felszabadult darabkákat erre szolgáló listákba fűzzük, tulajdonképpen megőrizzük a töredezettséget: felhasználtató és foglalt területdarabkák váltakoznak a memóriában. Később ez akkor okozhat problémát, amikor területet szeretnénk foglalni viszonylag nagy objektumok számára – ha nem fér be egyik „lyukba" sem, a szemétgyűjtés megléte ellenére növelnünk kell a felhasznált memória méretét. Megoldásként kínálkozik, hogy legalább figyeljünk, amikor a szabad területeket egybefűző listába térben szomszédos darabkák kerülnek, és tegyük őket egy nagy szabad területté. Ezzel a megoldással lehet valamit javítani a helyzeten, a probléma viszont gyökerében még mindig fennáll.

További szempont a hivatkozások szétszórtsága. Amikor egy program virtuális memóriával dolgozik –tehát memórialapokat tárol a háttértáron– figyelembe kellene venni a lapozások költségét is. A rendszer működése során a régi objektumok egyre inkább újakkal keverednek a memórialapokon, hivatkozásaik azonban rendszerint a hasonló korú társaikhoz kötik őket. Ennek eredményeképpen a program futása során a hivatkozások szétszórtsága miatt folyamatosan be kell lapozni hivatkozott oldalakat, holott az egymáshoz közel használt objektumok esetleg egy lapon is elférnének.

A két említett hátrány igazából a szemétgyűjtés második részfeladatával, vagyis a memóriaterület felszabadításával és nem a szemét felderítésével kapcsolatos, ezért a hivatkozás számlálásnál is jelentkezik. Amint látni fogjuk, vannak olyan szemétgyűjtő algoritmusok, melyek ezen hátrányok kijavítását célozzák.

Az egyszerű bejáró algoritmus időigénye a program által lefoglalt adatobjektumok számával arányos. Minél több objektum van, annál többet kell bejárni egy-egy szemétgyűjtés alkalmával, és annál többet kell felszabadítani. Jelentős visszalépés a hivatkozás számlálás módszeréhez képest viszont az, hogy a program futásával párhuzamosan ez az algoritmus ilyen formában nem végezhető, hiszen nem lenne biztonságos addig változtatni a memória helyzetén amíg a szemétgyűjtés be nem fejeződött. Olyan algoritmusok is ismertek, melyek a bejárás mint felderítés módszerét használva tudnak a programmal párhuzamosan futni, ezeknek a működését később tárgyaljuk.

2.3. Tömörítő bejáró algoritmus

A tömörítő bejáró algoritmus alapötlete, hogy az objektumok áthelyezésével javítható lenne a memória töredezettségének és a hivatkozások szétszórtságának problémája. Az algoritmus kezdő fázisa megegyezik az egyszerű bejáróéval: gyökerekből indulva felderíti a szükséges objektumokat és megjegyzi őket. Amikor a tárterület felszabadítására kerül sor, nem listába fűzi őket, hanem a szükséges objektumokat másolja közvetlenül egymás mögé. A szemétgyűjtés végére így a tárterület elején sorakoznak a szükségesnek nyilvánított objektumok, az utolsó objektum után pedig felhasználható az egész terület. Úgy is tekinthetjük, hogy ez a módszer nem a szemetet gyűjti, hanem a szükséges objektumokat.

Amit látjuk, a tömörítéssel megoldódik az egyszerű bejárónál kifogásolt töredezettségi probléma, a tömörítés folyamán ugyanis az objektumok közötti kisebb-nagyobb lukak feltöltésre kerülnek, a szabadon maradt területre pedig tetszőleges méretű új objektum befér. A hivatkozások szétszórtsága sem jelentkezik a továbbiakban, mert tulajdonképpen a létrehozás sorrendjében sorakoznak a még szükséges objektumok a memóriában – amennyiben a tárterület foglalást folyamatosan a szabad területen előrehaladva végeztük.

Jelentős költséggel jár viszont ez a szép tulajdonságokkal bíró algoritmus. A túlélő objektumok mozgatásához ki kell számolni azok új memóriacímét, át kell őket másolni a kiszámolt címre és minden rájuk mutató hivatkozást ki kell cserélni, hogy az aktuális címre mutassanak. Látszik, hogy mihelyst az első olyan objektumot megtaláltuk amit előre kell csúsztatni, a mögötte elhelyezkedő összes szükséges objektumot is mozgatni kell. A program végrehajtása pedig mindeközben áll, mivel nem engedhetjük meg a terep változtatását amíg a szemétgyűjtés biztonságosan be nem fejeződött. Számos próbálkozás létezik a másolási költség lefaragására. Ha egyszerre felülről és alulról vizsgáljuk a memóriaterületet egyik irányból a szükséges objektumokat másik irányból lukakat keresvén ahová másolhatjuk őket, kevesebb másolásra van szükség. Ezen megoldással időt nyerünk, és ezért a hivatkozások enyhe szétszórása a fizetendő ár.

2.4. Másoló bejáró algoritmus

A tömörítés ötletéből kiinduló algoritmus azzal csökkenti az időigényt, hogy a bejárással párhuzamosan végzi a tömörítés folyamatát. Mihelyt elér egy objektumot a bejárás folyamán, az illető adatot átmásolja oda, ahová a szemétgyűjtés eredményképp kerül.

A program futása során a rendelkezésre álló memóriát két részre osztjuk. Két szemétgyűjtés között folyamatosan az egyik rész aktív. Amikor memóriaigénylésre kerül sor, mindig az éppen aktív területről foglalunk helyet. Ha az adott területrész megtelt, szemétgyűjtést tartunk, melynek során bejárjuk a szükséges objektumokat és egymás után másoljuk őket az éppen használaton kívüli területrészre. Amikor az összes szükségeset átmásoltuk, a szemétgyűjtésnek vége, és a továbbiakban az a térrész aktív ami a szükséges objektumokkal feltöltésre került.

Egyik változata ennek a módszernek a *Cheney* algoritmus, amely a szükséges objektumokat szélességi bejárással deríti fel. Először a gyökérhalmazból azonnal elérhető adatot másolja a szabad területre, majd az átmásolt objektumokon elindulva azokat másolja át az aktív területről a sorba, amelyekre ezek hivatkoznak és a hivatkozásokat az új címre cseréli. A ciklikus adatstruktúrák azt a veszélyt jelentik, hogy egy objektum többször is bemásolásra kerül az új területrészre. Ennek kiküszöböléseként az átmásolt objektumok helyére az aktív térrészbe továbbító mutatókat helyezünk el, hogy amennyiben egy hivatkozásunk ismét oda irányítja a bejárás folyamatát, felismerjük, hogy ott már jártunk, és kicseréljük a hivatkozást az új –a továbbító mutató által jelzett– címre. Ha az objektumok során végigmentünk és nincs több hivatkozás, a másolási folyamatnak és ezzel a szemétgyűjtésnek vége.

Amint látjuk, az algoritmus a töredezettség mentesség megőrzésével és a hivatkozások szétszórtságának kiküszöbölésével megőrzi a tömörítő bejárás előnyös tulajdonságait, emellett pedig csökkenti az időigényét. Természetesen az algoritmus memóriaigénye nagyobb, hiszen a két területből egyszerre mindig csak az egyik használatos. Amennyiben a rendelkezésre álló memóriaterületet nem növeljük, hanem csupán két részre osztjuk, láthatjuk, hogy bár az egyes szemétgyűjtések időigénye kevesebb, gyakrabban kell azt végrehajtani mint az előző –memóriamegosztás nélküli– algoritmusnál, vagyis a szemétgyűjtés összességében akár többet is elvehet a futási időből.

2.5. Listás bejáró algoritmus

A listás bejáró algoritmus az előbbi absztrakciójaként mutatható be könnyen. A másoló bejáró algoritmus által használt két memóriaterület nem más, mint két halmaz. Valójában tetszőleges módon implementálhatjuk őket, elvárásunk csupán az, hogy könnyű legyen megállapítani egy objektumról, hogy melyik halmazban van, és hogy a két halmaz közti aktív-inaktív szerepcsere egyszerű legyen. Lényeges, hogy az objektumok bejáráskor a megfelelő halmazba kerüljenek, hiszen így biztosított tulajdonképpen a megőrzésük. Ez az algoritmus abban különbözik lényegesen a másoló bejárótól, hogy a szükséges objektumok másik halmazban való szerepeltetését másolás nélkül oldja meg.

Az ötlet a két halmaz implementálásában rejlik. Minden objektumhoz lefoglalunk egy három tagú „header” mezőt. Itt két mutatót tárolunk, és egy kapcsolót amelyik azt jelzi, hogy melyik halmaznak is eleme az objektum által foglalt memóriaterület. A két mutató arra való, hogy listába fűzze az adott memóriadarabkát – méghozzá vagy az aktív, vagy az inaktív listába attól függően, hogy éppen mi a szerepe. Ezek után csupán arról van szó, hogy az éppen használatban lévő objektumokat tartalmazó területek az egyik listába, a felhasználható szabad memóriaterületek a másik listába tartoznak. Ha kielégíthetetlen memóriafoglalási kérés érkezik, akkor az inaktív lista vagy kiürült, vagy kisebb darabkát tartalmaz a foglalás méreténél. Ekkor szemétgyűjtésre kerül a sor. Átfűzzük a megmaradt kisebb darabkákat az aktív listába (ezzel az inaktív halmaz kiürül) és bejárjuk a gyökerekből kiindulva az objektumainkat. A bejárás folyamán elért adatot azon nyomban átfűzzük az éppen inaktív listába, amit pedig a bejárás végére nem fűztünk át, az szemét – vagyis szabadon felhasználható memóriadarabka. A halmazok közti aktív-inaktív szerepcserét elvégezzük és a szemétgyűjtésnek ezzel vége.

A másoló bejáró algoritmus folyamatos másolgatási utasításainak lefaragásával időt takarítunk meg. Azzal, hogy fizikailag nem mozgatjuk az adatot, nem kell kiszámolni, hogy hova kerüljenek és a létező hivatkozásokat sem kell új címekre állítani. Fontos szempont, hogy a memória teljes méretét kihasználhatjuk, fizikailag nem bontjuk két részre. A jelentős költséglefaragás természetesen hátránnyal jár: ismét előkerül a töredezettség problémája. Minél tovább fut a program, annál inkább kis darabokra osztjuk a memóriát és a lapozási költségek is megnőhetnek a hivatkozások szétszórtsága miatt. A listás bejáró algoritmus az egyszerű bejárónál abban a tekintetben jobb, hogy a szemét eltakarítása a bejárás közben meg is történik, tehát különálló második fázis nem szükséges.

2.6. Feltételezések a szemétgyűjtésben

Amint az előzőekben is láthattuk, a szemétgyűjtő eljárások alapvetően nem rendelkeznek információval arról, hogy a futó alkalmazás mikor melyik tárolt objektumát kívánja használni. Éppen ezért olyan szemétgyűjtő algoritmust ami minden objektumot utolsó felhasználása után azonnal eltakarít nem lehetséges készíteni. Algoritmusaink feltételezésekre –legrosszabb esetekre– hagyatkoznak annak érdekében, hogy a program biztonságos működését ne zavarják. Két ilyen fontos feltételezéssel találkozhattunk az előzőekben.

Az egyik az, hogy minden a program számára elérhető objektum még felhasználásra kerül. Ennek eredményeképpen még elérhető –de használni már nem kívánt– adatot bejárunk, esetleg másolgatunk, rá hivatkozó mutatókat beállítunk – holott az elvégzett munka már tulajdonképpen felesleges. További feltételezésünk, hogy a globális változók, a veremben és a regiszterekben lévő objektumok szükségesek – ezeket tekintettük a bejáró algoritmusoknál gyökérnek. Elképzelhető azonban, hogy már a gyökerek között is van szükségtelen, tehát minden amit bejárunk egy bizonyos gyökérelemből és megtartunk, valójában szükségtelen. Mint tudjuk, ők sem biztos, hogy a továbbiakban a program futása során felhasználásra kerülnek.

Ezen feltételezések kiszűrésével nagyot javíthatnánk a szemétgyűjtő algoritmusok hatékonyságán, de ez csak a fordítóprogrammal való szoros együttműködés esetén lehetséges. A fordítóprogram a fordítás után információt tárolhatna arról, ha egyes objektumok a program bizonyos pontjától már biztosan nem szükségesek. Mivel a módszer további tárolási költséget von maga után –és még mindig csak közelítése a tényleges szükségesség meghatározásának– nem alkalmazott technika[1].

[1] Természetesen a szemétgyűjtő algoritmusok fordítóprogramokkal való alapvető együttműködésére mindig szükség van, akit a téma érdekel, annak a [3] hatodik fejezetét ajánlom megtekintésre.

3. Megszakítható szemétgyűjtés

Valós idejű alkalmazások futtatásakor az elvárt válaszidőt rendkívüli módon aláásshatják azok a szemétgyűjtő eljárások, amelyek teljes lefuttatása alatt szüneteltetni kell a programvégrehajtást. Olyan szemétgyűjtő algoritmusok lennének ilyenkor alkalmazhatók, amelyek megszakíthatóak, vagy valamilyen más módon a program végrehajtásával párhuzamosan működnek. Mint korábban láttuk, a hivatkozás számláláson alapuló algoritmus rendelkezett ezzel a kívánatos tulajdonsággal, az imént bemutatott bejárási elven működő szemétgyűjtők azonban nem. Mivel a hivatkozás számlálást negatív tulajdonságai miatt nem igazán szívesen alkalmazzák, olyan módszerek kerültek kidolgozásra, amelyek a bejárásos algoritmusokat teszik megszakíthatóvá.

A bejárásos algoritmusok megszakíthatóvá tételében a nehézség a feladat végrehajtásának első fázisában –a szükséges objektumok felderítésében– jelentkezik, mégpedig olyanformán, hogy a hivatkozási gráf bejárása közben változhat maga a gráf. A futó alkalmazás mint zavaró tényező működik, mégis valahogyan ilyen jelentős zavarás mellett is meg kellene oldani a szemétgyűjtést. A megszakíthatóvá tett bejárásos algoritmusoknak védekezniük kell a program által a gráfon végzett változásokkal szemben és –ami még fontosabb– nem szabad zavarni a futó alkalmazást a végrehajtott változtatásokkal. Másképp nézve a problémát, egy egyszerű bejárásos algoritmus megszakíthatóvá tétele egy több olvasó - egy író probléma: a szemétgyűjtő és a program is olvassa a gráfot, de csak ez utóbbi módosítja a mutatókat, és csak az előbbi módosítja az esetleges szükségességet jelző biteket. A tömörítő és a másoló bejáró algoritmus esetén a működést több olvasó - több író problémával lehet párhuzamba állítani, mivel a szemétgyűjtő és a program is változtatják a mutatókat a gráfban.

A probléma szemléltetésére használatos a gráfszínezés. A bejárás éppen aktuális állapota egy három színnel –fekete, szürke, fehér– színezett gráfon jól látható. Legyen egy olyan bejáró algoritmusunk ami a gráfot folyamatosan színezi. Induláskor minden a memóriában található objektumot fehérre színez. A bejárás első lépéseként a gyökereket szürkére színezi, bejárja a leszármazott csúcsaikat, és amikor egy gyökér minden közvetlen leszármazottját érintette, feketére színezi. Minden egyéb csúccsal is így tesz, vagyis amikor először eléri szürkére színezi őket –ezzel jelezvén, hogy az adott csúcs elérhető a hivatkozási gráfból de van utóda ami még nem került vizsgálat alá– majd amikor a közvetlen leszármazottait már mind érintette, feketére színezi őket. Természetesen ha olyan csúcsba érkezik amit már korábban feketére színezett, azt nem festi vissza szürkére és nem vizsgálja az utódait ezzel kiszűrvén a ciklikus adatstruktúrák többszörös bejárását. A bejárás befejeztével csak fekete és fehér objektumaink lesznek, a feketék szükségesek, a fehérek nem.

Vegyük észre, hogy a korábban ismertetett bejáró algoritmusok pontosan így működnek azzal az apró különbséggel, hogy szürke színre nincs szükségük, csak feketét és fehéret használnak. Az egyszerű bejáró algoritmus szükségességi bitek beállításával vagy a csúcs táblázatba való felvételével színezett feketére, a másoló bejáró algoritmus pedig a szükségesek inaktív részbe való bemásolásával színezett feketére. A szürke színre azért van szükség, hogy az algoritmus megszakítása után világos legyen, hogy mely csúcsok *kiterjesztésével*[2] kell folytatni a végrehajtást.

Látható, hogy a színezéses szemétgyűjtő nem tér vissza azokhoz a csúcsokhoz amiket már bejárt, márpedig lehet, hogy az általuk reprezentált objektumok a program időközbeni futása során feleslegessé váltak. Ez a szituáció azt vonja maga után, hogy nem szükséges objektumok nem kerülnek eltakarításra, de ez biztonságosság szempontjából nem probléma, esetleg hatékonysági kérdésként tárgyalható. Az viszont biztos, hogy a következő szemétgyűjtés alkalmával –amikor már az algoritmus első lépésétől kezdve szemétként lesznek tárolva– a kérdéses objektumok eltűnnek. Ebből is látszik, hogy a tény, miszerint a szemétgyűjtés végére bejárt gráf már nem felel meg az aktuális hivatkozási struktúrának önmagában még nem okozza a biztonság sérülését mindaddig, amíg a szemétgyűjtő nem tekint olyasmit szemétnek ami nem az.

Sajnos elképzelhető olyan helyzet is, amikor a már bejárt objektumok megváltoztatása befolyásolja a biztonságos működést. Ha például egy mutatót állítunk egy fekete –már bejárt– objektumból egy fehérre majd a máshonnan oda hivatkozó mutatókat felülírjuk, akkor az illető fehér csúcs a bejárás végéig fehér marad és biztosan eltakarításra kerül, holott szükséges lehet. Algoritmusunk tehát akkor működik helyesen, ha nem mutat egyetlen hivatkozás sem fekete objektumból fehérre; gondolnunk kell természetesen azokra az adatobjektumokra is, amelyeket a program a szemétgyűjtő algoritmus működése közben foglalt le. Látszik, hogy a konkurens módon végrehajtott szemétgyűjtés koordinációt igényel.

Az előbbiekben furfangosan felvázolt problémára kétféle megoldás létezik – az egyik az olvasás-, a másik pedig az írásfigyelő. Ha olvasásfigyelőt al-

[2] utódainak vizsgálatával

kalmazunk, az azt jelenti, hogy minden egyes alkalommal amikor a program kiolvassa egy fehér objektumra hivatkozó mutató értékét, az illető adatobjektumot rögtön szürkére színezzük. Ez azért jelent megoldást, mert a program ezek után nem tud úgy mutatót átmásolni máshova, hogy az fehérre mutasson.

Írásfigyelőt kétféleképpen alkalmazhatunk. Az egyik esetben azt figyeljük, hogy egy fekete objektumba mikor írtunk mutatót, a másik esetben pedig azt, hogy egy nem fekete objektumból mikor töröltünk vagy írtunk felül egy mutatót. Mindkét módszer a megfigyelt esemény hatására olyan extra utasításokat hajt végre, melyek aztán biztosítják, hogy a szemétgyűjtő informáltsága megmaradjon. Ezeket az utasításokat akár már a fordító is elhelyezheti statikusan a programkódban, hiszen fordítási időben is hozzá tudunk tenni valamit minden mutató íráshoz.

Az írásfigyelő algoritmusok kevesebb költséggel járnak mint az olvasásfigyelők, hiszen a mutató írás kevésbé gyakori művelet, mint a mutató kiolvasás – így extra kódot ritkábban hajtanak végre.

3.1. Írásfigyelés lefényképezéssel

A lefényképezéses írásfigyelés olyan, amelyik nem engedi, hogy a szemétgyűjtés kezdetekor létező objektumok eltűnjenek a szemétgyűjtő szeme elől. Minden esetben amikor mutató átírásra vagy törlésre kerül sor, az eredeti értéket elmenti, a szemétgyűjtés végén pedig az elmentett mutatókat gyökérként kezelve bejárja az azokból induló hivatkozási gráfot is. Az időközben lefoglalt memóriaterületek kapásból feketék, így azok felszabadítására sem kerülhet sor. A lefényképező algoritmus valóban biztosítja azt, hogy csak ténylegesen felesleges dolgokat tároló memóriaterület szabaduljon fel, hiszen még ha a program valahova –már bejárt területre– el is helyezett egy mutatót a felülírt helyett, akkor is biztosak lehetünk benne –bár a már fekete csúcsokat újra nem vizsgáljuk–, hogy a hivatkozott objektum megőrzésre kerül.

Ugyanezt hátrányként megfogalmazva a következőt mondhatjuk: az algoritmus futása közben feleslegessé vált memóriaterület már nem kerül felszabadításra, csak a következő szemétgyűjtés alkalmával. Ez azt is jelenti, hogy az újonnan létrehozott objektumok –ha csupán néhány utasítás erejéig voltak is szükségesek– csak a következő szemétgyűjtésnél lesznek eltakarítva.

Bár ez az algoritmus a fenti problémát megoldja, kevéssé hatékony a feltételezés miatt, miszerint minden a szemétgyűjtés elején szükséges objektum a szemétgyűjtés végén is szükséges.

3.2. Írásfigyelés visszatéréssel

A visszatérő algoritmus az eltárolt mutatókkal dolgozik, azt figyeli, hogy mikor helyez el a program egy mutatót már fekete –vagyis bejárt– csúcsba. Ha ilyesmi előfordul, az adott csúcsra a szemétgyűjtő eljárás visszatér. Ellentétbe állítva a lefényképező algoritmussal: nem azt figyeli, hogy egy mutatót mikor töröltünk olyan helyről ami még nem került bejárásra, hanem azt, hogy mikor került bemásolásra egy mutató már bejárt csúcsba. Ebből a megközelítésből szintén következik, hogy szükséges objektumot nem lehetséges tévedésből letörölni, viszont előnye az, hogy a szemétgyűjtés közben feleslegessé vált objektumok egy része –nevezetesen azok amelyeket a szemétgyűjtő bejárása még nem ért el feleslegessé válásuk pillanatáig– eltakarításra kerül.

A visszatérést akár a fekete akár a hivatkozott objektum szürkévé tételével előidézhetjük. A két megközelítés közti különbség az, hogy amennyiben a második lehetőséget választjuk úgy szerencsés esetben egy lépést előredolgoztunk a visszatérő bejárásnak, szerencsétlen esetben viszont felesleges objektumot járunk be, ha a mutatót ismét felülírjuk egy harmadikkal mielőtt a bejárás visszatérne.

Amikor új memóriaterületet foglalunk, azt ebben az algoritmusban nyugodtan színezhetjük fehérre. Ha a hivatkozó mutatót fekete objektumba helyeztük el akkor valamilyen módon úgyis megjelöljük, ha pedig nem feketébe, akkor elér majd oda is a bejárás. Előnye ennek az, hogy az újonnan lefoglalt de szinte azonnal feleslegessé vált memóriaterület már az aktuális szemétgyűjtéssel felszabadul.

Ez az algoritmus hatékonyabban teszi megszakíthatóvá a bejárást mint az előbbi. Itt sem igaz ugyan, hogy a szemétgyűjtés végén minden aktuálisan felesleges objektum eltűnik, de nem is őrizgeti biztonságból az összeset mely a folyamat közben vált feleslegessé.

3.3. Olvasásfigyelés másolással

Az olvasásfigyelés alapötlete az, hogy amely mutatót a program kiolvassa, azt rögtön szürkére festjük, hogy a megfelelő adatobjektum megőrzésre kerüljön.

Az itt bemutatott olvasásfigyelő algoritmus a másoló bejárásos szemétgyűjtésen alapszik, és a memóriában tárolt elérhető objektumok struktúráját szélességi bejárással deríti fel. Első lépésként –amely első lépést még nem hagyunk megszakíttatni a program által– átmásoljuk az inaktív részbe a gyökérhalmazból közvetlenül elérhető objektumokat. Miután ez megtörtént a program tovább futhat, szemétgyűjtő algoritmusunk pedig a már ismertetett módszer szerint halad.

Ha másolásos bejárást szeretnénk a programmal párhuzamosan futtatni, figyelnünk kell rá, hogy miután egy objektumot az inaktív memóriarészbe átmásoltunk, a program a másolattal dolgozzon, hiszen a szemétgyűjtés befejezése után az eredeti eltakarításra kerül, így az azokon végrehajtott változtatások is elvesznek. Olvasásfigyelőnk éppen ezért akkor teljesíti feladatát amikor a program megpróbál egy mutató hivatkozást feloldani. Ekkor a szemétgyűjtő ideiglenesen megszakítva a normális menetrend szerinti működést, rögtön megvizsgálja, hogy az illető objektum már átmásolásra került-e az inaktív térrészbe, és ha nem, akkor ezt azonnal megteszi. Miután ilyen módon a kérdéses gráfcsúcsot „szürkére festette", a bejárás és lépésenkénti másolás folyamatát ott folytatja ahol félbehagyta.

Mivel a minden egyes alkalommal való ellenőrzés költséges, némileg lefarag a költségekből az a megoldás, amely minden egyes objektum mellé egy elirányító mutatót helyez. Az irányító mutató mindig az éppen aktuális változatra mutat, először magára az objektumra, majd miután másolásra került, a másolatra. Ezzel az objektumok elérése az indirekció költségét mindig maga után vonja, de a még költségesebb feltétel ellenőrzésre azonban egyáltalán nincsen szükség.

Látszik, hogy aktív és inaktív területrész közti éles különbség elmosódik, úgy is megfogalmazhatjuk, hogy a szerepcsere már a szemétgyűjtés elején megtörténik, utána már csak az elérhető de inaktív részben rekedt adat kimentésén dolgozunk.

Az újonnan létrehozott adatobjektumok számára rögtön az új térrészben lesz lefoglalva a szükséges hely, ezzel biztosítva, hogy az objektumok nem vesznek el. A régi és új objektumok ebből következő teljes összekeveredését próbálja kivédeni az a módszer, ami az új térrész egyik végébe másolja a korábbi térrészből kimentett adatot és a másik végébe hozza létre az újakat.

Az eljárás a visszatéréses írásfigyelésre hasonlít abban az értelemben, hogy a gyűjtés alatt szemétté vált de még be nem járt objektumok tárhelye felszabadításra kerül, de annyival rosszabb, hogy az újonnan létrejött objektumok a lefényképező írásfigyeléshez hasonlóan még ha hamar feleslegessé is válnak, csak a következő szemétgyűjtéssel tűnnek el.

3.4. Olvasásfigyelés másolás nélkül

A másolás nélküli olvasásfigyelés a listás bejáró algoritmussal hozható összefüggésbe. Nagyon hasonlít az előző olvasásfigyelő megoldásra, az ötlet az, hogy memóriatérrészek helyett egy ciklikus listát használ az aktív és inaktív halmaz megvalósítására. A ciklikus két irányba láncolt lista egyszerre az aktív és inaktív halmazt is megvalósítja, sőt, mivel a módszer különválasztja az újonnan –a szemétgyűjtés alatt– létrehozott objektumokat és a szabad területet is, valójában négy halmazt ábrázol. Négy elválasztó jelzésre van szükségünk a halmazhatárok jelöléséhez az amúgy folytonos struktúrában. Sorrend szerint az inaktív (régi) halmaz, az aktív (új) halmaz, az új objektumok halmaza és a szabad területek halmaza követi egymást.

A szemétgyűjtés kezdetekor a középső két halmaz üres, az inaktív halmaz elemei fehérek. Az eljárás során folyamatosan kerülnek átláncolásra a bejárt elemek az inaktívból az aktív halmazba (listarészbe), és rögtön szürkére színeződnek. Amikor egy szürke elemnek már az összes leszármazottja át lett fűzve az aktív halmazba, az elem feketévé válik. Ha az olvasásfigyelő aktivizálódik, és az éppen olvasni kívánt hivatkozás az inaktív halmazban volt, átkerül az aktív halmazba szürke színnel. Memóriaterület foglalása az új objektumok és a szabad területek halmazát elválasztó jelzés arrábbhelyezésével történik, az újonnan lefoglalt terület (az előző algoritmushoz hasonlóan) fekete.

Amikor a bejárásnak vége, az aktív és az új elemeket tartalmazó –szomszédos– halmazok elemei mind feketék. A két halmazt összevonjuk az őket elválasztó jelzés előretolásával, így az új elemeket tartalmazó lista része lesz az előbbinek és együtt a következő szemétgyűjtés inaktív halmazát alkotják. A szabad területeket illetve az inaktív halmazt megtestesítő listát szintén összevonhatjuk, és az őket elválasztó jelzést szintén előretoljuk, méghozzá szintén az aktív és az új elemeket tartalmazó összevont halmaz végéig. Az új szemétgyűjtésben az így egyesített listarész a szabad memóriaterületeket tartalmazza, a halmazokat elválasztó jelzések pedig ciklikus szerepcserével töltik be funkciójukat (például az idáig inaktív-aktív halmaz határát jelölő határvonal mostantól a szabad területeket választja el az inaktív halmaztól – vegyük észre: ez a jelzés a helyén maradt).

A másolás nélküli olvasásfigyelés előnye éppen a másolási költség lefaragása. Nincs szükség a máshová áthelyezett objektumokra való hivatkozások átírására sem, hiszen fizikailag minden adat a helyén marad. Az eljárás közben keletkezett szemétről ugyanazt mondhatjuk el, mint az előzőleg bemutatott másolásos olvasásfigyelésnél.

3.5. Írásfigyelés másolással

A másoló bejáró algoritmust teszi még egyszerűbb módon megszakíthatóvá írásfigyelés segítségével a következőkben bemutatott algoritmus. Az elve hasonlít az olvasásfigyelés másolással módszeréhez, a különbség az, hogy a program folyamatosan az objektumok régi verziójával dolgozik a memóriában, akkor is, ha az illető adat már átmásolásra került az új térrészbe. Amíg a bejárás és a másolás folyik, a program egyáltalán nem látja a másolatokat, de amikor a bejárásnak vége, megtörténik a szerepcsere aktív és inaktív halmaz között és ezzel az inaktív halmaz objektumai eltakarításra kerültek.

Probléma ezzel nyilván akkor adódhat, amikor a program módosítja a már az inaktív térrészbe átmásolt objektumokat az aktív térrészben. Mivel a módosított adat már visszafordíthatatlanul eltakarításra kerül, a változást valahogy számon kell tartani. Ezt végzi el az írásfigyelő. Írásfigyelésünk tehát folyamatosan ellenőrzi a futó alkalmazás író utasításait, és ha az írás olyan adatot ír felül ami már bemásolásra került az inaktív térrészbe, a változást megjegyzi. Amikor a bejárást befejeztük, a szerepcsere elvégzése előtt a megjegyzett változtatásokat a szemétgyűjtő ismételten végrehajtja – ezúttal az új térrészben.

Új objektumokat az inaktív (új) térrészben foglalunk ezzel biztosítva, hogy ne legyenek szemétnek nyilvánítva. A bejárás elvégzése közben szemétté vált adat egy része –amit még nem másolt át a bejárás– eltűnik, amit már átmásoltunk, az a kövezkező szemétgyűjtésnél szűnik meg.

Az algoritmus drágának tűnhet, hiszen azt gondolhatnánk, hogy a módosítás igen gyakori. Olyan nyelvek is léteznek azonban, amelyekben az adatobjektumok destruktív módosítása egyáltalán nem megengedett vagy igen ritka – gondoljunk például a Clean-re vagy Haskell-re.

3.6. Hatékonyság a megszakítható szemétgyűjtésben

A hatékonysági kérdések szempontjából nem említett hivatkozás számlálás algoritmusa mint tudjuk megszakítható, pontosabban folyamatosan a program mellett működik. Előnyösebb az előbb bemutatott módszereknél abban, hogy gyakorlatilag azonnal eltakarít minden szemétté váló objektumot, viszont folyamatos összehangolása a programmal költséges, és mint láttuk a ciklikus adatstruktúrák hivatkozásainak felderítésében gyenge.

Összefoglalva elmondhatjuk, hogy a szemétgyűjtés megszakíthatóvá tételéhez a futtatott program és a szemétgyűjtő algoritmus összehangolására van szükség. Az ismertetett algoritmusok között a lényegi különbséget –az összehasonlíthatóság alapját– nem is a módszerek menete, hanem a hatékonysági kérdések jelentik. Hatékonyságon egyszerre értünk műveletigényt és az eltakarított valamint a létrejött szemét arányát. Az utóbbi értelemben az lenne az optimális módszer, ami a területek felszabadításának pillanatában az összes aktuálisan feleslegesen foglalt memóriadarabkát újra felhasználhatóvá tenné. (Egyik algoritmus sem szabadít fel több tárterületet mint amennyit a szemét elfoglal – ez következik abból amit biztonság alatt értettünk.)

Az eltakarított és összes szemét arányát ezeknél az algoritmusoknál az határozza meg, hogy a folyamat közben létrejött hulladékot, illetve az újonnan lefoglalt objektumokat –amik szintén potenciális hulladékdarabkák a szemétgyűjtés végéig– hogyan kezelik. Amennyiben feltételezik, hogy ezek az objektumok szükségesek akkor kevésbé hatékonyak, ha pedig ezzel a feltételezéssel nem élve bejárják az új objektumokra vonatkozó hivatkozásokat is, akkor hatékonyabbak. Általánosságban elmondhatjuk, hogy minél kevésbé hagyatkozik feltételezésekre egy algoritmus, annál hatékonyabb ebből a szempontból – viszont ezzel párhuzamosan annál több koordinációra van szükség a futó alkalmazással, vagyis annál nagyobb a műveletigény. Egyes esetekben az is bonyolítja a helyzetet, hogy a sok feltételezéssel élő algoritmusok kevesebb műveletigényük miatt gyakrabban lefuttathatók és ezzel esetleg még a szemét eltakarításának kérdésében is hatékonyabbak lehetnek.

Mivel a műveletigény a hatékonyság másik szempontja, láthatjuk, hogy az optimalitásról akkor nyilatkozhatunk, ha meghatároztuk a két szempont fontossága közti egyensúlyt – ami azonban az aktuális alkalmazástól függ. Ha egy program a működése során folyamatosan olyan objektumokat hoz létre amelyek szinte azonnal feleslegessé válnak, jobban beválnak az új objektumok elérhetőségét szintén vizsgáló algoritmusok, dacára a koordinációs műveletigénynek. Ezzel szemben ha a lefoglalt objektumok általánosságban hosszú életűek a program futása során, felesleges a program tevékenységének szoros nyomon követése hiába marad meg egy-egy szükségtelen objektum. Mivel szemétgyűjtő eljárást manapság nem konkrét programhoz hanem programozási nyelvhez készítenek, az alkalmazásra érdemes technika kiválasztásakor meg kell vizsgálni a nyelv lehetőségeit és gyakori felhasználásának területeit.

4. Generációs algoritmusok

A generációs algoritmusok alapötlete azon a megfigyelésen alapszik, hogy egy program futtatása során általában a lefoglalt objektumok nagy része igen hamar feleslegessé válik, míg kis hányaduk tartósan szükséges. Ettől a másolásos algoritmusok nem igazán hatékonyak. Minden egyes alkalommal amikor a másolásos szemétgyűjtés lefut, jelentős (másolási) költséget jelent az összes szükséges objektumot illetően – ha pedig nem másolásos algoritmus mellett döntünk, akkor a memória töredezettsége, a hivatkozások szétszórtsága az eredmény. Míg a rövid életű objektumok jellemzően maximum egyszer kerülnek átmásolásra, a sokáig szükségesek állandóan, minden szemétgyűjtés alkalmával áthelyeződnek. Ilyenkor az algoritmus feladata a sok régi adat folyamatos ide-oda másolása, vagyis jelentősen hatékonyabbá tehető, ha tudjuk melyek a hosszú életű és melyek a néhány utasítás erejéig szükséges objektumok.

A generációs módszerek valamilyen módon elkülönítik a régi és fiatal adatobjektumokat. A régiek körében ritkábban, az újak között gyakrabban tartanak szemétgyűjtést. Mivel az új objektumok gyakran rövid életűek, csak kis hányaduk bizonyul szükségesnek a felderítés során, így kevés az átmásolandó adat. Ha egy objektum már elért egy bizonyos kort, áthelyezik a régiek közé, így a továbbiakban ritkábban jelent költséget.

Ha nem megszakítható algoritmust használunk, akkor különösen előnyös generációs módszerrel dolgozni, hiszen a program futásának leállítása rövidebb ideig tart. A szemétgyűjtések nagy hányadában csupán a fiatal objektumokkal foglalkozunk, ritkán akasztjuk meg a program futását olyan átfogó szemétgyűjtéssel, mely az összes adatot végignézi.

4.1. Egy konkrét példa

Ahhoz, hogy megértsük milyen kérdések merülnek fel egy generációs algoritmus létrehozása során, nézzünk először egy konkrét módszert! Másoló generációs algoritmus alkalmazásakor annyiban térünk el a másoló bejárótól, hogy a memóriát nem csupán két térrészre osztjuk, hanem 2-3 területre, melyeken belül mindegyiknek meglesz a maga különálló két része. Minden terület egyegy generációhoz tartozik. Ha új memóriaterületet foglaltunk, azt mindig a legfiatalabb generáció területén tesszük. Szemétgyűjtést akkor tartunk, ha megtelik ennek a generációnak az aktív térrésze. Másoló bejáró módszerrel végignézzük a generációt és ami szükséges, azt az inaktív részbe másoljuk. Ha olyan elemeket találunk, melyek már több szemétgyűjtés alkalmával szükségesnek bizonyultak, nem az inaktív térrészbe, hanem a következő generáció aktív részébe másoljuk át őket.

Ha már sokszor történt szemétgyűjtés a fiatal generációban, elképzelhető, hogy a következő generáció aktív térrésze megtelik. Ekkor szemétgyűjtést tartunk a következő generációban pontosan ugyanúgy ahogy azt tettük a legfiatalabbikban. A legrégebbi objektumokat tartalmazni képes generáció felett már nincs több, így onnan már nem tudjuk feljebb mozdítani az adatot – szerencsés esetben idáig már csak az igen tartósan szükséges objektumok jutnak el, szemétgyűjtésre itt kerül sor a legritkábban.

4.2. Implementációs kérdések

Az imént leírt algoritmus még tartalmaz rögzítetlen részleteket, amiken érdemes elgondolkodni. Az egyik ilyen kérdés az, hogy hány generációt érdemes nyilvántartani és milyen arányban célszerű elosztani a rendelkezésre álló memóriát a generációk között. Megvizsgálhatjuk azt is, hogy mikor tekintsünk egy objektumot elég idősnek ahhoz, hogy a következő generációs szintbe helyezzük, illetve hogyan tartsuk nyilván egy-egy adatobjektum korát. Annak ellenére, hogy a különböző generációkban különálló szemétgyűjtéseket szeretnénk tartani, elkerülhetetlen, hogy megvizsgáljuk az egy-egy objektumra másik generációból mutató hivatkozásokat. Hogyan kellene ezeket a mutatókat nyilvántartani, megvizsgálni? Érdemes-e egyáltalán külön generációkra osztanunk az adatot, ha úgyis be kell járni az egészet a releváns mutatók előkerítéséhez?

4.2.1. Idős objektumok

A kérdésre, hogy mikor érett meg már egy objektum a feljebbmásolásra, nem mindegy milyen választ adunk. Az egyik véglet az, amikor minden egyes szemétgyűjtés alkalmával eggyel feljebb kerül a szükséges adat. Ez igen kevés költséggel megoldható, hiszen nem kell generáción belül nyilvántartani az objektumok korát semmiféle „header” mezőben. Jelentős megtakarítást jelent az is, hogy egy-egy generációnak csupán egy térrészre van szüksége, hiszen a túlélő adatot nem a szokás szerinti inaktív térrészbe, hanem rögtön a következő generációban másolja.

Ez a technika a hosszú életű objektumok szempontjából jó, hiszen hamar felkerülnek felső generációkba így nem sokszor másolgatjuk őket –hiszen egyre feljebb elvileg egyre ritkább a szemétgyűjtés–, rossz azonban azoknak az objektumoknak a szempontjából, amelyek rövid időre jöttek létre de közvetlenül a szemétgyűjtés elvégzése előtt. A rövid életű objektumok felsőbb generációba kerülése lelassítja a hatást, hiszen miattuk az idősebb adatok között is gyakoribb a szemétgyűjtés. Megoldásként segít az, ha minden második szemétgyűjtésnél emeljük csak fel a túlélő objektumokat. Ez még mindig nem igényel objektumonként plusz mezőt, csak éppen bele kell törődnünk, hogy a felemelt adatobjektumok között vannak amelyeknek az életkora eggyel különbözik egymástól.

A döntést természetesen az is befolyásolja, hogy hány generációval dolgozunk. Ha az előbb említett minden alkalommal felemelés technikáját alkalmazzuk, sok generáció kell ahhoz, hogy az alapötlet –a jelentősen különböző generációk szétválasztása– valóra váljon. Itt is szükséges természetesen egy legfelső generáció –nincs végtelen nagy kapacitásunk– ahonnan már nem emelünk. Ha viszont csak kevés generációt szeretnénk vagy tudunk nyilvántartani –extrém esetben csupán kettőt–, érdemes sokat várni az objektumok felemelésével, hogy az idősebbik generációba már tényleg a minél tartósabban szükséges adat kerüljön.

4.2.2. Generációk a memóriában

Hatékonysági okokból fontos döntés az, hogy a különböző generációkat hogyan helyezzük el a memóriában. Mivel a bejáró algoritmus számára fontos, hogy melyik objektum melyik generációból való, ennek minél könnyebben meghatározhatónak kell lennie. Az egyik legegyszerűbb hatékony megoldás az, ha a különböző generációk folytonos memóriadarabkákon –virtuális memória használata esetén akár külön memórialapokon– helyezkednek el, így a kívánt objektum hovatartozását már a címéből megtudhatjuk. Ez a módszer objektumonkénti plusz mezők tárolását is megspórolja. Ha azonban másolás nélküli szemétgyűjtést használunk, ez nem kivitelezhető – ilyenkor marad a „header" információ. Generációnként akár kétszer annyi helyet használhatunk fel a minden alkalommal áthelyező technikával –olyankor csupán a legidősebb generációnak kell két térrész–, ennek azonban már az előbb említettük a hátrányait.

Ha kevés generációt használunk és sokáig szeretnénk az objektumokat az első generációban tartani, rendezettebbé tehetjük ezen első generációt azzal, hogy három térrészen dolgozunk. Ezen módszer szerint új objektumok mindig a harmadik térrészben jönnek létre, szemétgyűjtés alkalmával pedig az aktív és a harmadik térrészből együtt gyűjtjük össze a szükségeseket és másoljuk az inaktív térrészbe. Ennek előnye az, hogy az az adat kerülhet be az aktív-inaktív körforgásba amelyik már legalább egy szemétgyűjtés alkalmával szükségesnek bizonyult. Vegyük észre, hogy a megoldás nem különbözik attól, mintha két generációt vettünk volna fel egy helyett, ahol az első generációból azonnal felemeljük a szükségeseket, míg a másodikban tartogatjuk egy ideig.

A legidősebb generáció az ahol a legritkábban tartunk szemétgyűjtést. Ennek a generációnak a két térrészre osztása memóriakihasználtság szempontjából nem igazán hatékony. Olyan algoritmus is létezik, amelyik a legidősebb generáció számára (is) csupán egy térrészt foglal, abban a térrészben viszont nem másoló, hanem tömörítő bejáró algoritmust alkalmaz. Így ugyan a legidősebb generációban való szemétgyűjtés hosszabb a többinél, de erre úgyis ritkán kerül sor, viszont összességében sokat spórolunk a tárterületen.

Egyes rendszerekben létezik egy olyan mértékig szükséges és állandó objektumokat tartalmazó generáció, ahol –normális esetben– sohasem fut szemétgyűjtés. Ide rendszerinformációk, programkód vagy olyan adat kerülhet, ami szinte sohasem változik és folyamatosan hivatkozott.

Bár a generációs algoritmusok elvének bemutatásához a másoló bejáró algoritmust használom, természetesen másolás nélküli algoritmus is tehető generációssá. Megvalósított generációs szemétgyűjtő létezik az egyszerű bejárásból kiindulva is. Ez az algoritmus másolási költségben és memória felhasználás terén előnyös, sajnos azonban ismét előkerül a memória töredezettségének és a hivatkozások szétszórtságának problémája. A különböző generációk fizikai keveredése a memóriában azt is jelenti, hogy feltétlenül szükséges „header" információkat tárolni, hiszen a memóriacím nem informatív a generációs hovatartozást illetően.

Bevált ötlet az is, hogy a nagy méretű objektumokat külön kezeljék a többitől –például egyszerű bejáró algoritmussal– hogy elkerüljék ezek másolgatását, ugyanakkor másolásos eljárást alkalmazzanak a kisebb adatnál, hogy ezzel is csökkentsék a memória töredezettségét.

Az algoritmusok megszakíthatósága független kérdés generációs mivoltuktól. Léteznek generációs megszakítható bejárásos algoritmusok, sőt olyan vegyes eljárások is, melyek a fiatal generációkban a program futását a szemétgyűjtés egész idejére leállító, míg az idősebb generációkban megszakítható módszert alkalmaznak. A logika ezen megközelítés mögött az, hogy a fiatal generációban való szemétgyűjtés viszonylag rövid és gyakran kívánatos momentum a program futása során.

4.2.3. Generációból kimutató hivatkozások

Az előbb felvázolt kép a generációs algoritmusokról némileg leegyszerűsített, vagy legalábbis elhallgattuk a legtöbb fejtörést igénylő problémát. A különböző generációk szemétgyűjtése ugyanis nem teljesen független egymástól. Célunk ugyan, hogy idősebb vagy fiatalabb objektumok között más-más időpontban tartsunk szemétgyűjtést, de egy-egy ilyen eljárás alkalmával ismernünk kell azokat a hivatkozásokat is, amelyek másik generációból mutatnak összegyűjtendő objektumainkra. Fontos ez egyrészt azért, mert a csak másik generációból hivatkozott objektumokat sem szeretnénk letörölni, másrészt azért, mert az ilyen hivatkozásokat is ki kell javítani, ha a hivatkozott objektumot áthelyezzük (másoló szemétgyűjtés alkalmával).

Minthogy az idősebb generációkból a fiatalabbakba jellemzően ritkább a hivatkozás mint a fordítva, gazdaságosnak tűnik az ilyen mutatók valamiféle nyilvántartása, a fiatalabb generációkból felfelé hivatkozó mutatók nyilvántartása viszont semmiképpen sem kifizetődő. Gyakori az olyan szemétgyűjtő eljárás, ami egy adott generáció szemétgyűjtésekor az idősebb generációkból mutató hivatkozásokat nyilvántartja, míg a fiatalabbakból mutatókat bejárja – vagy egyáltalán nem tart a fiatalabb generációktól független szemétgyűjtést.

Az idősebből fiatalabb generációkba történő hivatkozások nyilvántartására valamiféle írásfigyelést célszerű használni. A memóriába eltárolt mutatók írásakor kell odafigyelni és megvizsgálni, hogy a generációból „lefelé" mutató hivatkozás jött-e létre. Új objektumok inicializálásakor nem kell figyelni, hiszen azok a legfiatalabb generációban vannak így még fiatalabb generációba nem tudnak mutatni, csupán a felülírásokat kell nyomon követni.

Mivel az írásfigyelés és a kérdéses hivatkozások nyilvántartása igencsak költséges, jelentős részét teszik ki a szemétgyűjtési algoritmus műveletigényének. Sokféle megközelítés létezik az írásfigyelés megvalósítására, a következőkben ezeket tekintjük át.

Közvetett mutatók

Megoldást jelent a generációkból lefelé mutató hivatkozások nyilvántartására a közvetett mutatók használata. A módszer lényege, hogy a rendszer nem engedi, hogy létrejöjjön ilyen közvetlen mutató, hanem egy táblázatba való hivatkozás jön létre, a táblázat pedig mindig tartalmazza a hivatkozott objektum aktuális memóriacímét. Több táblázat létezik, méghozzá generációnként, vagyis minden generációnak saját táblázata van ami az oda történő hivatkozásokat irányítja el. A program természetesen közvetlennek kell lássa a hivatkozásokat, egy-egy olvasás alkalmával gondoskodik a rendszer arról, hogy az indirekciót feloldja. Szemétgyűjtéskor az algoritmus az adott generáció táblázatának bejegyzéseit szintén gyökérnek használva jár el.

A megoldás sajnos elég költségesnek bizonyul, hiszen minden egyes mutató íráshoz vagy olvasáshoz további utasításokat kell végrehajtani. Nagy műveletigénye miatt a módszer nem használatos.

Objektumok nyilvántartása

A halmazos nyilvántartás technikája engedélyezi a közvetlen hivatkozásokat a generációk között, viszont megjegyzi, hogy hová tárolt a program ilyen hivatkozásokat. Ha egy fiatalabb generációba mutató hivatkozás jön létre, felveszi a hivatkozó objektumot egy halmazba. Ez a megoldás csupán írásfigyelést igényel, nem kell minden egyes hivatkozás olvasáskor szubrutint alkalmazni, hogy a közvetettséget feloldjuk.

Sajnos ennek a módszernek is jelentős költségei lehetnek, a szemétgyűjtésnek ugyanis előnyösebb lenne ha mindjárt a mutatókat tárolnánk, mintsem objektumokkal kelljen dolgoznia. Ha tehát méretes objektumból történik hivatkozás és szemétgyűjtéskor látjuk, hogy ott hivatkozást kell keresni, át kell fésülni az egész objektumot, hogy hol van(nak) benne a kérdéses mutató(k) és mire is hivatkoznak valójában. Ez főleg akkor jelentős, ha egyes nagy méretű objektumokon folyamatosan dolgozik a program, így lényegében minden egyes szemétgyűjtésnél be kell járni őket. Másik felesleges műveletigény az, amikor két szemétgyűjtés között többször is felülírunk egy mutatót. Ekkor az írásfigyelő ellenőrzését, hogy vajon generáción túlmutató hivatkozás jött-e létre minden egyes alkalommal végrehajtjuk, holott tulajdonképpen csak legutoljára lenne szükséges.

Memórialapok nyilvántartása

Ez a technika abban különbözik az előzőtől, hogy objektumok helyett memórialapokat jegyez meg, amelyekben van kritikus hivatkozás. Szemétgyűjtéskor pedig nem az objektumokat nézzük át amelyek hivatkozást tartalmaznak, hanem a kérdéses memórialapokat.

A módszer előnye akkor mutatkozik meg, ha a memórialapjaink viszonylag kicsik –hiszen át kell vizsgálni a kérdéses lapokat szemétgyűjtéskor– az objektumok pedig méretesek. Igazából átlagos hardveren implementálva a módszer igen műveletigényes.

Gépi szavak nyilvántartása

Sokkal gazdaságosabbnak bizonyult átlagos hardveren az a megoldás, ami azokat a gépi szavakat jegyzi meg, ahol a mutatók szerepelnek. Pontosabb ez a nyilvántartás mind a halmazos mind a memórialapos nyilvántartásnál, mert nem kell sem objektumokat sem lapokat végigvizsgálni, konkrétan a mutató helye kerül eltárolásra.

Másik optimalizáló momentum a módszerben az, hogy ha egy gépi szóban módosítottunk egy mutatót, az rögtön megjegyzésre kerül és csak szemétgyűjtés alkalmával vizsgáljuk azt, hogy tulajdonképpen generáción túlmutató hivatkozást tartalmaz-e. Ennek előnye az, hogy ha egy mutatót többször is felülírunk, az idáig írásfigyelő feladataként számon tartott ellenőrzést csak szemétgyűjtésenként kell végrehajtani. Ha kritikus mutatót fedezünk fel a szó továbbra is megjegyzésre kerül –hiszen a következő szemétgyűjtésnek is figyelembe kell vennie függetlenül attól, hogy felülírjuk-e addig– ha pedig csak generáción belüli vagy felfele mutat a hivatkozás, töröljük a nyilvántartásból az adott szót.

A módszer hátránya ez esetben a helyigény, hiszen az összes kritikus gépi szó nyilvántartásához sok „halmaz elem" szükséges.

Memóriakártyák nyilvántartása

A memóriakártyák nyilvántartása az egyensúlyt próbálja megtalálni a memórialapok túl nagy volta miatti szemétgyűjtéskor jelentkező magas átvizsgálási költség és a gépi szavak kicsinysége okozta tárolási helyköltség között. A memóriakártyák méretének belövése attól függ, hogy inkább tárolási helyet, vagy műveletigényt szeretnénk spórolni, illetve mi az ami olcsó az adott hardveren.

A technika nehézségét az okozza, hogy egy memóriakártya nem mindig kezdődik objektummal. Szemétgyűjtéskor meg kell keresni azt a megelőző kártyát ami objektummal kezdődik, hogy folyamatos előrehaladással megtaláljuk az első objektum „header" információit. Hasonlóan a gépi szavak nyilvántartásához, a mutatók generációból kimutató voltát csak a szemétgyűjtéskor vizsgáljuk.

4.3. Problémák a generációs alapelvvel

A generációs algoritmusok alapelve egy feltételezés, amelynek teljesülése nem garantált. Nevezetesen nem feltétlenül teljesül, hogy az objektumok korával összefügg a szükségességük. Ha egy program minden objektumot nagyjából azonos ideig használ, nem nyerünk a generációs algoritmuson, viszont sokat vesztünk az azzal járó adminisztrációs költségeken.

4.3.1. Generációk közti hivatkozások

Feltételeztük az írásfigyelés bevezetésekor, hogy kevés hivatkozás lesz idős objektumokból fiatalabbakba. Ha azonban egy mutatókat tartalmazó tömböt tekintünk, aminek az elemeit úgy módosítjuk, hogy allokálunk egy új objektumot és ráállítjuk a tömb mutatóját, megdőlt az elv, a tömb ugyanis hosszú ideig tárolt struktúra, míg az általa hivatkozott objektumok rövid életűek lesznek. Az eredmény az lesz, hogy a sok és gyorsan változó hivatkozások nyilvántartása írásfigyeléssel komoly költségeket jelent.

Megoldásként kerülhetjük a mutatókat tartalmazó tömbök használatát és konkrétan az elemeket tárolhatjuk egy tömbben hivatkozások helyett, ekkor azonban előre meg kell határozni az elemek pontos típusát, ami egy dinamikus típusokat alkalmazó rendszerben megszorítást jelent. Ha azonban egy fa struktúrára gondolunk, rögtön látjuk, hogy ez a megoldás csupán egy speciális esetre alkalmazható.

4.3.2. Nagy gyökérhalmaz

Nagyobb szoftver-rendszerekben a globális illetve lokális változókból igencsak sok lehet, márpedig a bejárás gyökereként ezeket fel kell venni. Hiába szűkítjük a szemétgyűjtést egy generációra, a gyökérhalmazt magát teljes egészében kell tekinteni, így egy fiatalabb generáció szemétgyűjtése is igen költséges lehet. Megoldási ötlet, hogy egyes gyökérváltozókat is memóriaobjektumként kezeljünk. Ekkor ezeket nem kell a gyökérhalmazba belevenni, hanem rájuk is írásfigyelőt alkalmazunk és változásaikat úgy követjük nyomon mint a memóriaobjektumokéit, viszont költségesebb lesz ezek után ezeket a –valójában lokális– változókat felülírni, használni.

4.3.3. Nagy adatstruktúrák

Jelentős költséget jelent másolásos generációs szemétgyűjtésnél egy nagy adatstruktúra létrehozása, ami sokáig fennmarad. A struktúra előbb az első, majd a második generációba kerül és szép lassan jut feljebb, közben pedig folyamatos másolási műveletigényt okoz. Ha ilyen esetre gyakran számítunk, fontos, hogy az objektumokat ne várakoztassuk sokáig egy-egy generációban, hanem minél hamarabb toljuk őket feljebb, csak figyelni kell az egyensúlyra – nehogy ezen igyekezetünkkel túl gyakorivá váljon a szemétgyűjtés a magasabb generációkban. Olyan módszer is létezik, amelyik folyamatos nyomon követéssel dinamikusan próbálja meghatározni azt az optimális időintervallumot amit az objektumok egy generációban töltenek.

4.4. Megszakítható generációs szemétgyűjtés

A generációs szemétgyűjtést lehet megszakítható technikák alkalmazásával kombinálni. Óvatosnak kell azonban lenni, hiszen ha a generációs algoritmusok alapfeltételezése nem teljesül, tönkretehetjük a megszakítható szemétgyűjtés előnyös tulajdonságait és a rendszer nagyon nehézkesen működő, sok felesleges műveletigénnyel dolgozó szemétgyűjtéssé válik. Akkor célszerű a kombináció, ha garantálni tudjuk a generációs algoritmus hatékonyságát.

A kombinálásnak egyik bevált lehetősége, hogy a generációkat nem teljesen függetlenül takarítjuk, hanem az egyes szemétgyűjtések csak az alsó n generációban gyűjtik a szemetet, de azt egyszerre – megszakítható módon. Így teljesül, hogy a felső generációkban ritkábban foglalkozunk a feladattal, a fiatalabb generációkból felfelé történő hivatkozásokat pedig implicit kezeljük. Figyelnünk kell viszont arra, hogy egy esetleg sok generációt érintő hosszadalmas szemétgyűjtés időben befejeződjön, hogy mire elfogyna a szabad hely, ismét felszabadítsunk memóriaterületet.

5. Dinamikus szerkesztés a Clean nyelvben

A Clean nyelv fejlesztésével a hollandiai Nijmegen-ben foglalkoznak. Az ottani egyetem több tanára, doktoranduszok és programozók alkotják a Clean csoportot. Igyekezetükkel a funkcionális programozásból adódó előnyöket próbálják olyan tulajdonságokkal ötvözni, mely a nyelvet ipari használatra vonzóbbá teszi. A nyelv programozási lehetőségei folyamatosan bővülnek, miközben a helyességbizonyítási eszközök fejlesztése is intenzíven folyik. Feladatuk azért összetett, mert nehéz a mai imperatív programozási nyelvek szolgáltatásait úgy megvalósítani, hogy a Clean nyelv tiszta funkcionalitását megőrizzék – márpedig ehhez a helyességbizonyítás megvalósíthatóságának érdekében ragaszkodnak. [3]

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* beolvashatnák és használhatnák. A dinamikus szerkesztés lehetősége számtalan esetben hasznos lehetne.

5.1. A dinamikus szerkesztésben rejlő lehetőségek

1. Ha egy számítógépen fenn lenne 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 beszerkeszthetnék és kiértékelhetnék a meghívott függvényeket, használhatnák az ott definiált típusokat.

2. A programok tudnának a *kódállományok*on[4] 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

[3]Mivel a dolgozat hátralévő részében előfordulnak programozási példák és a funkcionális programozással kapcsolatos fogalmak, a témakörben járatlan olvasónak ajánlom az irodalomjegyzékben szereplő [4] illetve [9] irodalmakat.

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

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 (az eddig 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 parancsok mindegyike pontosan meghatározható típussal rendelkezik és megfelelő típusú paramétereket vár el. Az így elkészült operációs rendszer rendkívül biztonságosan bánna a tárolt adatokkal és működésében igen megbízható lenne[7].

A Clean csoport az ötletet komolyan vette, és nekikezdett a megvalósításnak. Mire másodmagammal bekapcsolódtam a csapat munkájába, a dinamikus szerkesztés alapvetően már működött, módosítások és javítások után a rendszer megújult és a működését elősegítő programok készültek. Az összeállított rendszer működését tekintjük át a következő alfejezetben.

5.2. A `Dynamic` típus és műveletei

A megvalósítás alapötlete a `Dynamic` típus[6]. 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.

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 a Clean nyelvben gyakran használt módszerrel: mintaillesztéssel működik. A mintaillesztés itt két fázisú: először végrehajtódik egy *típus* mintaillesztés, majd ha ez sikeresen megtörtént, 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 következő 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 visszaadott `Bool` érték a mentés sikeres végrehajtását jelzi. Egyszerű példaként tekintsük az alábbi függvényt mely a paraméterül kapott érték faktoriálisát menti el – ahol a `fac :: Int -> Int` típusú függvény faktoriálist számol:

```
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 következőképpen definiált függvénnyel:
   `readDynamic :: String *World -> (Bool,Dynamic,*World).`
   Így tudunk tehát kódállományt a programunkba beszerkeszteni. Például 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.

5.3. 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, a definiált típusok leírása egy *typ* kiterjesztésű fájlba, illetve egy *bat* fájl jön létre a program futtatásához. Az előbbi két fájl egy a programozó által nem hozzáférhető könyvtárban jön létre. Ezen fájlok nevei tartalmuknak a Ronald L. Rivest által kifejlesztett *MD5*-ös algoritmussal[8] meghatározott ellenőrzőösszegéből jönnek létre a következőképpen: <*lib*fájl ellenőrzőösszege>_<*typ*fájl ellenőrzőösszege> és az ehhez járuló kétféle kiterjesztés. Az elnevezések is tükrözik, hogy a két fájl szorosan egymáshoz tartozik, csupán később látható hatékonysági és biztonsági szempontok alapján tároljuk külön őket.

A program futtatása a *bat* kiterjesztésű fájl futtatását fogja jelenteni, ami parancssori hívást tartalmaz a *dinamikus szerkesztőprogram*ra a programhoz tartozó könyvtár-[5] illetve típusfájl[6] 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[7].

Amikor egy dinamikus alkalmazás kiír egy kifejezést kódállományba, a létrejövő fájl egy csak kódállományokat tartalmazó könyvtárba kerül, amelyhez a programozó szintén nem fér hozzá. Neve tartalmának *MD5*-tel kiszámolt ellenőrzőösszege, kiterjesztése *dynsys*. Eltárolt hivatkozást tartalmaz a programhoz tartozó könyvtár- illetve típusfájlra azok ellenőrzőösszegeinek formájában. Ahhoz, hogy a programozó használni is tudja ezt az elmentett fájlt, létrejön a kívánt könyvtárban egy mutatófájl a kódállományra. (A mutatófájl ebben az esetben csupán egy szövegfájl *dyn* kiterjesztéssel, aminek a nevét a programozó határozhatja meg az alapján, hogy mit tárolt az elmentett kódállományba, tartalma pedig a hivatkozott kódállomány ellenőrzőösszege.)

Ha egy másik dinamikus alkalmazás később beolvassa a kiírt kifejezést, akkor a dinamikus szerkesztőprogram először beolvassa a mutatófájlból a valódi

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

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

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

kódállomány nevét, majd azt előkerítve az általa hivatkozott 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 (a Clean-ben megszokott módon 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 kódállományokban nem tároljuk többszörösen ezeket, hanem 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. A program végrehajtása ott folytatódhat ahol a beolvasással megszakítottuk.

Az *MD5*-ös ellenőrzőösszegek és a fájlok elrejtése a biztonságot szolgálja. Hiába lenne ugyanis biztonságos maga az elképzelés a típusinformációk ellenőrzése terén, ha rosszindulatú felhasználók a típusfájlban tárolt információk meghamisításával rávehetnék a dinamikus szerkesztőprogramot arra, hogy nem megfelelő típusú nyelvi elemet szerkesszen be a egy programba futás közben. A program futása összezavarodhat ilyen helyzetben, még ha esetleg nem is furfangos felhasználók, hanem véletlen fájlsérülés vezet a szituációhoz. Az ellenőrzőösszegek lehetővé teszik, hogy a szerkesztőprogram –mielőtt beszerkesztene egy fájlt– megbizonyosodjon arról, hogy létrehozása óta az illető fájl nem változott. Ha a fájlnév és a tartalomból kiszámolható ellenőrzőösszeg nem stimmel, megtagadja a beolvasást.

A fájlok elrejtése azért fontos, mert a felhasználónak nem szabad őket átnevezni, áthelyezni vagy törölni. Ha megfelelő könyvtár- és típusfájlok nincsenek jelen egy kódállomány beolvasásához, az illető kódállomány használhatatlan.

5.4. Alkalmazási példa

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

5.4.1. 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
```

5.4.2. 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 viszont 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 kódállományokat tartalmazó rendszerkönyvtárba, és egy rá hivatkozó mutatófájl a felhasználó által kívánt könyvtárba. A kódállomány hivatkozást 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 ellenőrzőösszegét 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 a hozzájuk tartozó könyvtárfájlokkal együtt. Valójában itt a `read_dynamic_apply`-ban történik meg a függvény alkalmazása - a kifejezés kiértékelése, majd mindezek után a képernyőre való kiírás.

6. A felmerült szemétgyűjtési feladat

6.1. Függőségek

Mint az az előzőekből látható, a dinamikus szerkesztőprogram folyamatosan írja ki a könyvtár- és típusfájlokat valamit a kódállományokat az őket tartalmazó két rendszerkönyvtárba. A kiírt állományokat hivatkozások kötik össze. A példában is nyomon követhettük, hogy a kódállományok mindegyike hivatkozik a hozzá tartozó típus- és könyvtárfájlokra, illetve amennyiben nem teljesen kiértékelt „lusta" kódállományról van szó –mint például a példában szereplő *result*– hivatkozik még a kiértékeléséhez szükséges további kódállományokra. Ez a hivatkozási rendszer egy fa struktúrát alkot. A könyvtár- és típusfájlok mindig a hivatkozásfa levelei –hiszek ők már nem hivatkoznak semmire– a hivatkozásfa belső csúcsai mindig kódállományok, gyökerei pedig a felhasználói könyvtárakból származó kódállományokra mutató fájlok. Ciklikus hivatkozás sohasem alakulhat ki.

Olyan kisebb (három csúcsú) hivatkozási struktúrák is kialakulnak, amik a felhasználónál tárolt *bat* fájlokat fűzik össze a hozzájuk tartozó könyvtár- és típusfájlokkal.

Mint látható, elég komoly veszteségeket okozhatna, ha a felhasználó jogosult lenne letörölni a kódállományokat valamint a típus- és könyvtárfájlokat, mert ha azokat más fájl hivatkozza, az a törlés után használhatatlanná válik – ugyanis a beszerkeszthetőségéhez, kiértékelhetőségéhez szükséges információ vész el. Ezt elkerülendő kerülnek létrehozáskor külön könyvtárba ezek az állományok, a felhasználó szeme elől elrejtve. Ha egy *bat* fájlt töröl a felhasználó, az nem gond, hiszen nyilván azért törölte mert nem akarja többé azt a programot futtatni, más pedig ilyen fájlra nem hivatkozik. A kódállományokhoz azért készít a rendszer mutató fájlokat, hogy a programozó –akinek a számára a mutatófájlok logikailag a kódállományt magát jelentik– úgy bánhasson velük ahogy neki tetszik: törölhesse, átnevezhesse őket. Mivel ezekre a fájlokra senki sem hivatkozik, nem okoz gondot, ha eltűnnek a helyükről.

6.2. A függőségekből adódó probléma

A hivatkozásfa a felhasználónak csupán egy logikai struktúra, ezt a struktúrát nem látja, nehezen deríti ki, hogy melyik fájl melyik másikra hivatkozik – mint tudjuk a fájlok neve automatikus, emberi szemlélő számára értelmetlen 128 bites azonosító. Minthogy azonban a létrejövő állományok igen terjedelmesek lehetnek –a könyvtárfájlok például gépi kódot tartalmaznak– valamilyen módon törölni kellene a feleslegesen tárolt állományokat. A felhasználónak ezt a munkát biztonsági okokból nem engedjük elvégezni, így automatikus szemétgyűjtő programot kell elkészíteni.

A megoldandó feladat és a programok futtatása közben szükséges memóriabeli szemétgyűjtés között párhuzamot vonhatunk. Ami ez utóbbinál a memória, az itt a két rendszerkönyvtárnak feleltethető meg. A memóriaobjekumok a fájlok, a mutatók hivatkozási struktúrája pedig a fájlok közti függőségi gráf – ami ráadásul fa. A gyökérobjektumok a globális változók, a regiszterekben vagy a vermen tárolt változók helyett itt a felhasználói könyvtárakban tárolt kódállomány mutatók és a dinamikus alkalmazások elindításához szükséges *bat* fájlok.

A két feladat természete különbözik is. Mivel a gyűjtés célterülete a háttértár –sőt az sem teljesen– már maga a szemétgyűjtő program is bizonyos absztrakciós szintről fogja látni a tárterületet – nem lesz például tisztában a tároló töredezettségével.

Az elképzelt felhasználás koncepciója a következő: a programozó amikor úgy határoz, hogy nincs már szüksége egy kódállományra törli a mutatófájlját; ha nem akar már egy programot használni, törli a programfájlt; és amikor úgy gondolja, hogy már elég sok szemét gyűlhetett össze, lefuttatja a szemétgyűjtő alkalmazást. A következő különbség tehát az, hogy a szemétgyűjtés elindítása nem a tárterület foglaló eljárás feladata. Mivel dinamikus alkalmazás futtatása közben nem fontos szemétgyűjtést végezni, megszakíthatóságra sem kell gondolni.

Hasonlít viszont a két feladat a leglényegesebb ponton: itt is fontos, hogy a biztonságot szem előtt tartva dolgozzunk, hiszen egy hivatkozott fájl törlése éppúgy futási idejű hibához vezet, mint a memóriabeli szemétgyűjtés esetén.

A feladat tehát olyan szemétgyűjtő program elkészítése, amely a háttértár dinamikus rendszeréhez tartozó rendszerkönyvtárakban felderíti a feleslegesen tárolt állományokat és azokat de csak azokat törli.

6.3. Az elkészült megoldás

Sikerült megvalósítanom a rendszer szemétgyűjtő eljárását.

A program első lépésként egy megadott könyvtárból[8] kiindulva összegyűjti

[8] A könyvtár nevét egy erre szolgáló konfigurációs fájlból olvassa ki.

az összes alkönyvtárban található *dyn* és a *bat* kiterjesztésű fájlokat – az utóbbiak esetén természetesen ellenőrzi, hogy valóban dinamikus alkalmazásról van-e szó. Az összegyűjtött fájlokból mint gyökerekből elindulva felépíti a hozzájuk tartozó hivatkozási fákat, így kapja meg a rendszer biztonságos működéséhez a rejtett könyvtárakban megtartandó fájlok listáját. A harmadik ütemben összehasonlítja a kapott listát a dinamikus szerkesztőprogramtól megkapott rendszerkönyvárban tárolt fájlok listájával, majd a nem hivatkozott könyvtár- és típusfájlokat illetve rendszerhez tartozó kódállományokat törli.

Az elkészült szemétgyűjtő program tehát az egyszerű bejárásos algoritmus alapján dolgozik. Nem a felesleget, hanem a szükséges objektumokat deríti fel, és komplementerképzéssel határozza meg a felesleges objektumok halmazát. A szokásos szemétgyűjtési fázisok –szükséges objektumok bejárása, felesleges objektumok törlése– elé harmadik fázis kerül: a gyökerek felderítése, amit az alkalmazás a megadott kiindulási könyvtár alkönyvtárainak mélységi bejárásával végez el, közben a megfelelő kiterjesztésű fájlok után kutatva.

Az algoritmus nem megszakítható, hiszen ha a gyökerek összegyűjtése után egy dinamikus alkalmazás futtatásának eredményeképpen új kódállomány jön létre, az már nem kerül bejárásra hanem töröljük – annak ellenére, hogy a létrejött mutatófájl hivatkozza. A szemétgyűjtés elvégzésével egyidejűleg tehát dinamikus alkalmazást nem futtathatunk. (Egyéb természetű programok konkurens futtatása természetesen nem zavaró tényező.)

A gyűjtés kiindulási könyvtárának megadási 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 fejlesztésekkel.

Meggondolandó tényező még az egyes operációs rendszerekben megvalósított szemétkosár (recycle bin) 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 szemétkosárban lehet. Ha aztán a felhasználó a letörölt fájt visszahelyezi a helyére, a rendszer megbízhatósága felborul. A dolog megoldása egyrészről az lehet, hogy a szemétgyűjtőnek elérhetővé tesszük a szemétkosár könyvtárat 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.

6.4. Egyéb megoldási lehetőségek

Érdemesnek tartottuk még elgondolkodni hivatkozási számlálós megoldáson. Ahhoz a megoldáshoz olyan nyilvántartásra lett volna szükség –mondjuk egy fájlra valamelyik rendszerkönyvtárban– amelyikben tároljuk, hogy melyik állományra hányan hivatkoznak. Már említettük, hogy egy-egy hivatkozási struktúra fa, tehát a ciklikus hivatkozások felderítésének hiánya nem jelentett volna problémát. A dinamikus szerkesztőprogram minden egyes állomány létrehozásakor karban tudta volna tartani a hivatkozási számlálókat (növelni őket amikor új hivatkozás jön létre). A számlálók karbantartásához azonban figyelemmel kellett volna kísérni azt is, hogy a felhasználó mikor töröl vagy éppen másol le mutató vagy *bat* fájlokat, és azok mire hivatkoztak. Ennek a nyilvántartása már túlmutatott az elkészítendő program hatáskörén.

Másolásos algoritmust könnyen alkalmazhattunk volna, hiszen új rendszerkönyvtárak létrehozása és a szükséges állományok átmásolása után az eredeti rendszerkönyvtárak teljes tartalmának törlése valóban eltakarítja a szemetet. Gondoljuk azonban át, hogy az algoritmusban elég komoly műveletigényt vonna maga után a másolgatás, nagy méretű objektumokról lévén szó. Azt se felejtsük el, hogy a másolásos algoritmusok fő előnye a memória töredezettségének és a hivatkozások szétszórtságának megelőzése, ami viszont itt egyáltalán nem szempont. Egyrészt azért mert a háttértár töredezettsége nem ezen szemétgyűjtő feladata, másrészt nincs is hozzáférése erről az absztrakciós szintről a háttértár fizikai képéhez.

Konklúzió

Diplomamunkámban azzal foglalkoztam, hogy a Clean csoport munkájában a dinamikus programszerkesztés lehetőségének megteremtése közben milyen probléma került elő, és hogyan vonható párhuzam a felmerült probléma és a programok futtatása közben szükséges memóriában való szemétgyűjtés között. Láthattuk, hogy milyen algoritmusok születtek korábban szemétgyűjtési feladatok elvégzésére, és ezen algoritmusok egyikének felhasználásával megoldottuk a feladatot.

Az így létrejött megoldásnak erősségeit és gyengeségeit egyaránt megismertük, összességében pedig megfelelőnek értékeltük. Az elkészített program azóta használatban van –és rendeltetése szerint működik– a programozók által a dinamikus szerkesztés használata közben készített felesleges fájlok közti szemétgyűjtés elvégzésére.

Irodalomjegyzék

[1] P. Cheng: *Scalable Real-time Parallel Garbage Collection for Symmetric Multiprocessors*; PhD thesis, Carnegie Mellon University, 2001.

[2] S. E. Abdullahi, E. E. Miranda, G. A. Ringwood: *Collection schemes for distributed garbage*; International Workshop on Memory Management, St-Malo, September 1992. Springer Verlag, LNCS 637, pp. 43-81.

[3] P. R. Wilson: *Uniprocessor garbage collection techniques*; Technical report, Submitted to ACM Computing Surveys, University of Texas, January 1994.

[4] 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

[5] 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.

[6] 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

[7] 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.

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

[9] 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.