



Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és
Fordítóprogramok Tanszék

Undo/redo modellek vizsgálata refaktorálások során ¹

Szabó Richárd

Programtervező Informatikus MSc, I. évfolyam

Témavezető: **Tóth Melinda**

Tudományos segédmunkatárs

Budapest, 2011

¹Támogatta a KMOP-1.1.2-08/1-2008-0002, Európai Regionális Fejlesztési Alap (ERFA) és Ericsson Magyarország

Tartalomjegyzék

1. FingerTree	3
2. Undo/redo modellek	6
2.1. Lineáris undo	6
2.2. Nemlineáris undo	7
2.2.1. Szkript modell	7
2.2.2. US & R modell	8
2.2.3. Hármass modell	8
2.3. Szigorított lineáris undo	9
2.4. Kitekintés: szelektív undo	10
3. Refaktorálás undo/redo modellje	11
3.1. Változások kezelése, összefésülés	11
3.2. Változások kiszámítása	14
4. Összevetés verziókezelő rendszerek megoldásaival	17
4.1. Darcs	17
4.2. SVN	18
4.3. Összegzés	18
5. Esettanulmányok	20
5.1. Konfliktus vizsgálata refaktorálás és szerkesztés esetén	20
5.2. Vizsgálat szintje az összehasonlításnál	21

6. RefactorErl undo/redo integrációja	27
Irodalomjegyzék	32

Kivonat

Programok refaktorálása során más alkalmazásokhoz hasonlóan szükség van az undo/redo funkcióra. Ezt a szövegszerkesztők általában nem támogatják, csak egy egyszerű undo/redo funkciót biztosítanak. A refaktorálások undo/redo műveletének viselkedésével kapcsolatos elvárások figyelembe vételével kidolgoztam egy speciális undo modellt, mely szövegszerkesztőkbe integrálva lehetővé teszi refaktorálások változásainak helyes kezelését undo/redo műveletek során, kezelve az esetlegesen felmerülő konfliktusokat. Az egyetműnkön fejlesztett RefactorErl refaktoráló eszköz Emacs-felületéhez integráltan megvalósítottam az így leírt modellt.

Bevezető

Az undo/redo funkció fontos szolgáltatás grafikus felületű alkalmazásoknál, szövegszerkesztőknél, fejlesztő eszközöknél, hiszen a kényelmes felhasználói felület miatt könnyű hibát ejteni. Egy programfejlesztés során is gyakran előfordul, hogy a felhasználó egy végrehajtott változtatást, műveletet vissza akar vonni (vagy újra végre akar hajtani), mert más úton kívánja megoldani azt amit változtatott, vagy egyszerűen elrontott valamit.

Hagyományos szövegszerkesztésnél a módosítások egy fájlban szekvenciálisan történnek, mindig egy meghatározott ponton érintik a forrást, egy refaktorálás [5] azonban egy fájlban több ponton is érinthet, sőt, akár egyszerre több fájlban is. Ennek megfelelően, ahogy egy refaktorálási művelet végrehajtása bonyolultabb egy egyszerű szerkesztésnél, annak visszavonása vagy megismétlése is nehezebb feladat.

A refaktorálás következő speciális tulajdonsága, hogy több fájlban érintő változások esetén a visszacsévézése konfliktushelyzeteket is hordozhat magában, ugyanis lehet, hogy az éppen visszavonni kívánt refaktorálás után az egyik érintett fájlban még módosítottuk. Ilyen esetben egy undo nem feltétlenül a közvetlen megelőző állapotra való visszalépést jelent, hanem valamilyen összefésült, vagy bizonyos változtatásokat elfelejtett korábbi állapotot eredményez.

A szövegszerkesztők általában nem támogatják az ilyen jellegű undo/redo műveletet, csak buffer szintű, lokális undo-t, a refaktorálás undo pedig bizonyos értelemben globális, hiszen az egy változáshoz tartozó több érintett fájl

változásait együtt kell kezelnie.

Dolgozatomban refaktorálás undo/redo műveletének problémáit vizsgálom: hogyan lehet egy fájl tartalmát, az azon bekövetkezett komplex változásokat hatékonyan tárolni, számolni, felügyelni, hogyan lehet egy refaktorálás visszavonása során fellépő konfliktusokat feloldani. A modellt úgy akartam megvalósítani, hogy azt minél könnyebben be lehessen építeni meglevő fejlesztői környezetekbe, szövegszerkesztőkbe. Ehhez azt meg kellett oldani, hogy a fájl tartalmát is hatékony módon el tudjuk tárolni, hatékony szövegpuffert kellett megvalósítani. Az 1. fejezetben egy FingerTree nevű adatszerkezetről írok, mely hatékonyan használható szövegpuffer megvalósítására. Erre az adatszerkezetre, illetve az adatszerkezet pufferként való használatára már létezett egy Haskell nyelven írt hatékony implementáció, így a refaktorálás undo/redo megvalósításához ennek néhány elemét felhasználtam. Az adatszerkezet rövid bemutatása után ismertetek néhány alapvető undo/redo modellt, majd bemutatok egy saját, refaktorálás undo/redo-ra specializált modellt, amely képes kezelni a refaktorálások komplex, akár több fájlos változásait, illetve refaktorálás visszavonása során fellépő konfliktusokat. Az 5. fejezetben az így definiált modellre vonatkozó esettanulmányok találhatóak, végül mutatok egy konkrét megvalósítást a leírt modellre, melyet a Refactor-Erl eszközhöz integráltam.

1. fejezet

FingerTree

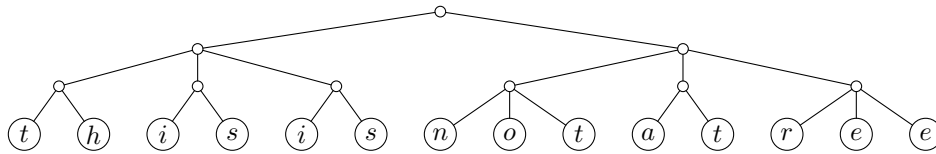
Egy fájl tartalmának tárolására egy lehetséges adatszerkezet, a FingerTree [9], egy olyan általános, kiegyensúlyozott fa, melynek csúcsai egy monoid elemeivel vannak címkézve, az adatokat pedig az ún. ujjakon tárolja. Fontos szerkezeti tulajdonságai, hogy az ujjak minden szintnek az elején, illetve a végén találhatók, így hatékonyan, amortizált konstans költséggel elérhetők. Ezenkívül logaritmikus idejű - $\Theta(\log(\min\{n, m\}))$ - a vágás, illetve az összefűzés művelete. Egy konkrét típus a felhasználás céljának megfelelő monoid, és a tárolandó adattípus megválasztásával áll elő. Például egy prioritásos sort úgy valósíthatunk meg, hogy a belső csúcsokat a gyerekekben levő prioritások minimumával/maximumával címkézzük. Egy lista funkcionalitású, hatékonyan indexelhető struktúra a gyerekek leveleinek számával való címkézéssel implementálható.

Speciálisan, ha az ujjakon sztringeket tárolunk, a címkékben pedig a megfelelő részfában levő karakterek számát, hatékonyan tudjuk használni a struktúrát szöveg puffer megvalósítására. Az említett szerkezeti invariáns miatt az elemek hatékony elérése biztosított, illetve a gyakori módosítás is hatékony a vágás/konkatenálás miatt. Ezenkívül fontos megjegyezni, hogy az adatszerkezet tisztán funkcionális - így perzisztens -, ami szintén növeli a hatékonyságot.

Az adatszerkezet egyszerűsített definíciója a következő:

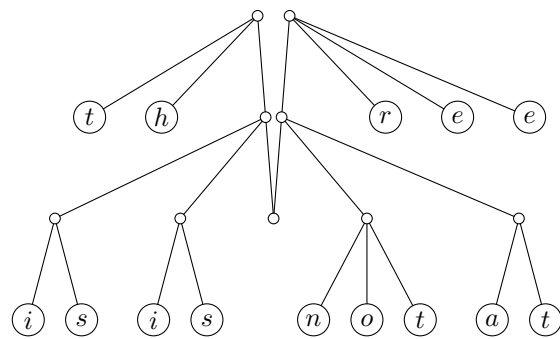
```
data FingerTree a = Empty
                | Single a
                | Deep (Digit a) (FingerTree (Node a)) (Digit a)
```

A definícióban és az 1.2. ábrán láthatjuk, hogy a szintek elején és a végén vannak a tárolt elemek. Az ábrán a struktúra közepén látható kétfelé váló gerincet tekinthetjük egy a fán végighúzódnó útnak. A fa bejárása a 0. szinten levő gyökből indulva ezen a gerincen haladva történik, ebből a középső útból mindkét irányba legfeljebb egy-egy csúcs vezethet tovább az adott szinten. Az adatszerkezet teljes definíciójához hozzátartozik a monoid címkézés is, melynek segítségével a bejárás során az aktuális szinten tovább tudunk haladni a keresett elemhez vezető élen. A fa gerince vagy egy üres csúcsban, vagy egy egy elemű csúcsban végződhet. A következő 1.1 és 1.2 ábrákon láthatjuk az alapvető különbséget a 2-3 fa és a FingerTree szerkezete, alakja közt.



1.1. ábra. 2-3 fa szerkezete

Az egyik legjelentősebb eltérés a két struktúra közt, hogy míg a 2-3 fában az adatok egy szinten, a leveleken vannak tárolva, a FingerTree-ben több szintre eloszlanak, a szintek elején/végén levő ujjakon kerülnek tárolásra. Ennek megfelelően és a fent leírtak alapján az adatok adatok elérése, a fa bejárása is más módon történik: az adatszerkezetben a szinteken a középén található gerincen haladva juthatunk el a címkézés alapján a kívánt elemhez.



1.2. ábra. FingerTree szerkezete

2. fejezet

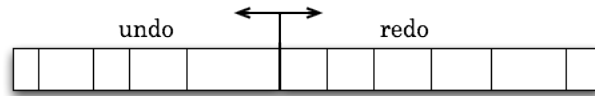
Undo/redo modellek

Számos általános felhasználású undo modell [1] létezik, melyek alapvetően két csoportra bonthatóak: lineáris undo, nemlineáris undo. A lineáris undo a végrehajtott parancsok egy sorozatát rögzíti, ahol közvetlenül csak az utolsó művelet vonható vissza, csak lineárisan lépkedhetünk a különböző állapotok között. Ezzel szemben, a nemlineáris undo megengedi, hogy a felhasználó rögtön egy korábbi változtatást vonjon vissza, a későbbiek visszavonása nélkül, feltéve, hogy az aktuális állapotban a parancs értelmesen végrehajtható.

2.1. Lineáris undo

A lineáris undo modell minden végrehajtott parancsot egy történeti-listában tárol, mely elméletben korlátlan hosszú lehet. A lista végén levő parancs visszavonható. Visszavonáskor a történeti-listából törlődik, a redo-lista elejére kerül. Ez a művelet mindaddig megismételhető, amíg a lista ki nem ürül. Az utolsó visszavont változtatás, mely a redo-lista elejére került újra végrehajtható, az undo-lista elejére tehető. Az új parancsok a redo-lista változtatása nélkül az undo-lista elejére kerülnek. A felhasználó ilyen módon a parancsok sorozatát bejárhatja (2.1. ábra). Egy redo parancsot tekinthetünk egy undo undojának, azonban ebben a modellben a redo parancsok

csak implicit módon vannak jelen, nem részei a történeti-listának, az érintett parancsot a redo-listában külön tároljuk el.



2.1. ábra. undo-redo lista

2.2. Nemlineáris undo

A felhasználó elvárhatja, hogy közvetlenül ne csak az utoljára végrehajtott művelet visszavonására legyen lehetőség. Könnyen előfordulhat ugyanis, hogy már több újabb parancsot végrehajtott, miután egy hibát észrevesz a módosításban. Többfelhasználós alkalmazásokban az undo kényelmes használatához nélkülözhetetlen, hogy az utolsónál korábbi parancs visszavonható legyen, hiszen a felhasználó számára lokálisan utolsó művelet globálisan tekintve nem feltétlenül az utolsó, közben más felhasználó is végrehajthatott parancsokat. A következőkben bemutatok néhány jelentősebb nemlineáris undo modellt.

2.2.1. Szkript modell

A szkript modell a felhasználói műveleteket parancsok egy szkriptjének szerkesztésével modellezi. A felhasználó a szkriptet szerkesztve kitörlésével tetszőleges parancsot visszavonhat, a rendszer egyszerűen újra végrehajtja a parancsokat a kezdőállapotból indítva. Egy másik lehetséges, jóval hatékonyabb módszer, ha csak a törölni kívánt parancsig visszamenően visszavonjuk a változásokat, majd annak kivételével újra végrehajtjuk a visszavont parancsokat. A szkript modell egyszerű, jól működő, azonban felhasználói felületét

tekintve nem a legmegfelelőbb, különösen ha grafikus felületű alkalmazásnál használjuk.

2.2.2. US & R modell

Ez a modell a fenténél egyszerűbb felülettel rendelkezik. A lineáris undo-hoz hasonlóan rendelkezik egy undo és egy redo művelettel, ezen felül egy szkip művelettel, mellyel egy parancs redo-ja átugorható. Ilyenkor az adott parancs nem törlődik, csak megjelölésre kerül. Új parancsok felvételekor a redo lista tartalma megmarad, így tulajdonképpen egy történeti fa keletkezik. Ez a fa valójában egy irányított gráf, hiszen különböző ágakból indulva lehetséges a parancsokat újra végrehajtani, új éleket létrehozni. Bár ez a modell működését tekintve egyszerű, a történetet leíró szerkezet rendkívül bonyolulttá, nehezen áttekinthetővé válhat, használata kényelmetlen.

2.2.3. Hármass modell

A hármass modell szintén három műveletet definiál, de harmadik művelete a szkip helyett a forgatás. A modellhez felhasznált adatszerkezet egy egyszerű undo és egy redo lista párosa. A visszavont parancsok a redo listába kerülnek, a forgatás művelete pedig a redo lista utolsó elemét a lista elejére teszi. Forgatással kiválasztható a következő újra végrehajtandó parancs, illetve a következő visszavonandó parancs helyét is kiválaszthatjuk. Ezek alapján megvalósítható egy tetszőleges, a listában levő parancs visszavonása: a kívánt parancsig visszamenően undo-t kell alkalmazni, forgatni kell a redo listát, majd néhány redo-t kell alkalmazni. Egy a redo listában levő műveletet a már említett módon, forgatásokkal lehet elérni, majd végrehajtani. A hármass modellről az előző két modellhez hasonlóan azt mondhatjuk, egyszerű működési mechanizmusa ellenére, kényelmetlen felülete miatt nem a legmegfelelőbb modell.

2.3. Szigorított lineáris undo

A fenti modellek mindegyikének problémája, hogy azt feltételezik, bármelyik parancs értelmesen újra végrehajtható más állapotban is, mint amiben eredeti végrehajtáskor volt. A problémát szemlélteti a következő, rajzoló-programból vett példa: a felhasználó rajzol egy kört, ezt átmozgatja egy másik helyre. Ezután visszavonja a mozgatást, törli az objektumot. Az előző nemlineáris modellekben ebben a helyzetben lehetséges volna a mozgatás műveletének redo-ja, azonban ez hibát okozna, hiszen az objektum, amire a mozgatás parancsa hivatkozik, nem létezik.

Ilyen és ehhez hasonló problémák elkerülhetők a lineáris undo szigorított változatában, mely teljesíti a következő tulajdonságot (*biztonságos végrehajthatóság*): egy parancsot mindig abban az állapotában hajtunk újra végre, mint amiben eredetileg volt, és mindig abban az állapotban vonjuk vissza, amiben az adott parancs eredeti végrehajtása után volt. Az állapot fogalma ezekben a modellekben mindig a program undo művelet szempontjából érdekes jellemzőinek együttesét jelenti.

A lineáris undo egyetlen esetben szegi meg a fenti tulajdonságot, amikor úgy adunk hozzá új parancsot az undo listához, hogy a redo-lista nem üres. Ennek a konfliktusnak egy lehetséges, egyszerű, bevett feloldása, ha ilyen esetben egyszerűen töröljük a redo-lista tartalmát.

A szigorított lineáris undo modell szöveges fájl változásaira specializálva következőképp képzelhető el: a végrehajtott módosításokat - absztrakt formában - eltároljuk egy listában, ami a fájl történetét reprezentálja. A lista elején a legutóbbi változás található, ami következő alkalommal visszavonásra kerülhet. Ha ez megtörténik, az undo listából kitöröljük, inverzét berakjuk a redo lista elejére. Az utolsó visszavont, azaz a redo lista elején levő művelet megismételhető, melynek inverze így újra az undo lista elejére kerül. Az új változások folyamatosan az undo listába kerülnek. Ezenkívül a biztonságos végrehajthatóság tulajdonságát a fent leírt módon teljesíthetjük.

Ez a módszer egyszerű szövegszerkesztő undo/redo funkciójának megva-

lósításához jól használható, azonban a refaktorálás lehetséges konfliktusainak lekezelésére a refaktorálás undo/redo-hoz módosítást, bővítést igényel. Refaktorálás undo-nál a megkövetelt tulajdonság megszégése akkor fordul elő, ha egy több fájlos refaktorálás visszavonásakor az egyik érintett fájlban nem ez a refaktorálás a legutóbbi módosítás. Szemléletesen a fenti, rajzolóprogramban leírt példához hasonlóan akkor léphet fel konfliktus, ha valamely változásban egy korábbi változás eredményre van hivatkozás: például, ha egy refaktorálás utáni változás hivatkozik egy a refaktorálás által létrehozott változóra. A stabil tulajdonság sérülése esetén meg kell vizsgálnunk, hogy értelmes módon meghagyhatjuk-e a refaktorálás utáni változtatás(oka)t, vagy azokat is vissza kell vonnunk. Ehhez a vizsgálathoz azonban szükséges, hogy a változásokat az egymással való összehasonlításához egy közös szintre hozzuk, ugyanis a fájl különböző állapotában következtek be.

2.4. Kitekintés: szelektív undo

A szelektív undo alapötlete az, hogy a felhasználónak megengedhetjük, hogy tetszőleges korábbi parancsot válasszon ki a történeti-listából és beszúrjon egy új parancsot, ami visszafordítja a kiválasztott parancs hatását. Amennyiben egy korábbi parancs ilyen kiválasztása és visszavonása nem értelmes, a felhasználónak nem engedjük meg annak kiválasztását. Speciálisan egy refaktoráló eszköz esetén ez azt jelentené, hogy a felhasználónak felajánlanánk az utolsó néhány változást a történeti listából, a listából kiválaszthatja, melyik változást kívánja visszavonni. Amennyiben a kiválasztott parancs konfliktusmentesen visszavonható, egy a változással ellentétes hatású parancs a redo-listába beszúrára kerül. Ellenkező esetben visszautasítjuk a felhasználó kérését, vagy az adott változást követő módosításokat szintén visszacsévéljük.

3. fejezet

Refaktorálás undo/redo modellje

A következő fejezetben megadok egy modellt, mely a szigorított lineáris undo refaktorálásokra specializált, bővített változata.

3.1. Változások kezelése, összefésülés

Minden fájlra vonatkozóan tároljuk annak tartalmát, valamint a lineáris undo modell alapján a fájl tartalmának módosításait egy undo-redo listapárosban tároljuk, melyben egy változás elemi módosítások sorozataként jelenik meg, ahol egy elemi módosítás az egy beszúrás, vagy egy törlés, annak kezdőpozíciójával és az érintett szöveggel.

```
data URList = URList (undo :: [Change], redo :: [Change])
data Change = Change (changedId :: Int, updates :: [Update])
data Update = Insert (updatePoint :: Point, updateString :: String)
              | Delete (updatePoint :: Point, updateString :: String)
```

A fájl aktuális tartalmát és a változások által érintett szöveget egyaránt FingerTree-ben tároljuk, hiszen így tudjuk kihasználni az adatszerkezet hatékony összefűzését és a perzisztenciát. Ezek alapján már definiálni tudjuk,

két egymást követő c_1, c_2 változás konfliktusosságát. Két egymás utáni változás konfliktusban áll, ha valamely elemi módosításuk közös állapotra hozva konfliktusban áll egymással:

$$c_1 = [u_1, \dots, u_n], \quad c_2 = [v_1, \dots, v_m]$$

$$\exists i \in [1..n] : \exists j \in [1..m] : u'_i \parallel v_j$$

ahol u'_i u_i frissített pozíciókkal, végigvezetve az azt követő $u_{i+1}, \dots, u_n, \dots, v_1, \dots, v_{j-1}$ módosításokon. Két elemi módosítást tehát úgy hozunk közös állapotra, hogy a korábbi végigvezetjük az utána levő – későbbiig tartó – módosításokon. Ez azt jelenti, hogy két egymást követő módosítás vizsgálatakor a pozíciójuk és a típusuk – beszúrás, vagy törlés – alapján szükség esetén kiszámítjuk a korábbira az új pozíciót, ezzel felhozva a következő módosítás szintjére. Két azonos szinten levő módosítás konfliktusosságát nem vizsgáljuk meg előre, hanem megpróbáljuk őket összefésülni, az esetleges konfliktusosságot eközben ismerjük fel, ilyenkor megállunk a módosítások újraszámításával. A következőekben részletesen bemutatom az összefésülés algoritmusát.

Tegyük fel, hogy a két paraméter x és y , ahol x a későbbi módosítás, y a korábbi. A módosításokra úgy kell tekintenünk, hogy jelenleg az eredeti állapotukkal ellentétesen jelennek meg törlésként, vagy beszúrásként. Az x -re úgy kell tekintenünk, hogy ezzel ellentétes módosítás történt y után, y -t pedig a jelenlegi állapotának megfelelően – szükség esetén frissített pozíciókkal – újra végre akarjuk hajtani.

Két beszúrásnál, ha x y előtt van, y -t feljebb kell hoznunk, hiszen x -szel az előtte levő területen töröltünk, így feljebb került a korábbi kezdete. Ha y x előtt elkezdődött, jelenlegi pozícióik átlapolhatnak, de így sem érintik egymás területét, ha x -et lejjebb visszük, hogy y -nak adjuk meg a korábbi, eredeti helyét.

$$\begin{aligned} x : (\text{Insert } p \ s) \ y : (\text{Insert } p' \ s') \\ |p| + |s| \leq p' = (x, \text{Insert } ((p' - |s|)) \ s') \end{aligned}$$

$$\begin{aligned}
| p' \leq p &= (\text{Insert}((p + |s'|)) s, y) \\
| \text{otherwise} &= \text{Nothing}
\end{aligned}$$

Egy beszúrásnál és egy törlésnél, ha x y előtt van, y -t feljebb kell hoznunk, hiszen x -szel az előtte levő területen töröltünk, így feljebb került a korábbi kezdete. Ha y x előtt véget ér, x -et feljebb kell hoznunk, hiszen most előtte x -et kitöröljük.

$$\begin{aligned}
x : (\text{Insert } p \ s) \ y : (\text{Delete } p' \ s') \\
| p + |s| \leq p' &= (x, \text{Delete}((p' - |s|)) s') \\
| p' + |s'| \leq p &= (\text{Insert}((p - |s'|)) s, y) \\
| \text{otherwise} &= \text{Nothing}
\end{aligned}$$

Egy törlésnél és egy beszúrásnál, ha y x előtt van, x -et lejjebb kell vinnünk, hiszen x -szel most elé fogunk beszúrni. Ha x y előtt van, y -et lejjebb kell vinnünk, hiszen x -szel elé szúrtunk be szöveget.

$$\begin{aligned}
x : (\text{Delete } p \ s) \ y : (\text{Insert } p' \ s') \\
| p' \leq p &= (\text{Delete}((p + |s'|)) s, y) \\
| p < p' &= (x, \text{Insert}((p' + |s|)) s') \\
| \text{otherwise} &= \text{Nothing}
\end{aligned}$$

Két törlésnél, ha x y előtt van, y -t lejjebb kell vinnünk, hiszen x -szel az előtte levő területre beszúrtunk, így lejjebb került a korábbi kezdete. Ha y x előtt van, x -et feljebb kell hoznunk, hiszen most előtte x -et kitöröljük.

$$\begin{aligned}
x : (\text{Delete } p \ s) \ y : (\text{Delete } p' \ s') \\
| p < p' &= (x, \text{Delete}((p' + |s|)) s') \\
| p' \leq p &= (\text{Delete}((p - |s'|)) s, y) \\
| \text{otherwise} &= \text{Nothing}
\end{aligned}$$

Ahol nem teljesülnek az összefésülhetőség feltételei, minden esetben megáll az algoritmus negatív válasszal.

Ennek megfelelően rekurzívan definiálhatjuk egy fájl két tetszőleges változásának konfliktusosságát: a korábban bekövetkezett változást – hasonlóan, ahogy az elemi módosításokat – végigvezetjük a későbbiig, közben számolva a módosítások új pozícióit.

Ahhoz, hogy a több fájlos refaktorálásokat összetartozó változtatásként tudjuk kezelni szükséges a változásokat valamilyen módon csoportosítani, az ilyen refaktorálásokat, és az érintett fájlokat külön feljegyezni. Ezért minden változás rendelkezik egy azonosítóval, az így egybe tartozó változások azonosítója pedig megegyezik.

Így tehát egy konfliktusos undo-nál (redo-nál) meg kell vizsgálnunk az adott fájlban a refaktorálás és az azt követő változások konfliktusmentességét, ha ez teljesül, az így kiszámított refaktorálás visszavonása egy összefésült állapotot eredményez. Amennyiben konfliktus van valamely változások közt, ki kell hagynunk az összes változást a refaktorálásig visszamenően, eközben figyelve azt, hogy a több fájlos kitörölt refaktorálás(oka)t rekurzívan vissza kell csévélnünk a többi fájlban is.

3.2. Változások kiszámítása

Egy fájl minden módosításakor kiszámítjuk az előző állapotához képest történt változásokat. Ehhez alapvetően használhatjuk a diff [3] algoritmust, mellyel a két állapotban a tartalmat szövegszerűen, karakterek szintjén összevetjük. Az algoritmus a fájl tartalmának megfelelő sorrendben megadja, mik azok a szövegrészletek, amelyek csak az egyik, vagy csak a másik állapotban találhatóak meg.

$$\text{diff } s \ s' = cs$$

$$cs = [c_1, c_2, \dots, c_n]$$

$$\forall i \in [1..n] : c_i = (d, str)$$

$$d = \begin{cases} B & \text{ha } str \text{ mindkét szövegben szerepel} \\ F & \text{ha } str \text{ csak az első szövegben szerepel} \\ S & \text{ha } str \text{ csak a második szövegben szerepel} \end{cases}$$

Az algoritmus kimenete alapján kiszámolhatjuk, milyen elemi módosítások, beszúrások, törlések sorozata vezet el a korábbi állapotból az újabbá:

updates $((d, s) : cs) p$
 $| d == B = \textit{updates } cs (p + |s|)$
 $| d == S = \textit{Delete } p s : \textit{updates } cs p$
 $| d == F = \textit{Insert } p s : \textit{updates } cs (p + |s|)$

Az eredménylistát feldolgozva mindkét szövegben haladunk az elejétől a vége felé, közben számoljuk a következő törlés vagy beszúrás pozícióját (p) 0 kezdeti értékkel:

- B : mindkét szövegben szereplő részlet esetén ebben az aktuálisan vizsgált részben nem történt változás; növeljük mindkét pozíciót a szövegrészlet hosszával, ezzel előretolva a következő törlési és beszúrási pozíciót is
- S : a második, azaz most a régebbi állapotban szereplő szövegrészlet egy törlést jelez; nem kell előrelépniünk a pozíció számolásával, hiszen ez a következő pozíciókat nem tolja el
- F : az első, azaz most az újabb állapotban szereplő szövegrészlet a régihez képest egy beszúrást jelez; a számított pozícióval előrelépünk a szövegrészlet hosszával, hiszen a beszúrás következő pozíciókat eltolja

Az így kapott törlések, beszúrások sorozatát a szöveg régi állapotán sorban végrehajtva megkapjuk az újnak megfelelő tartalmat. Ezzel nemcsak azt érjük el, hogy megkapjuk az állapotra vonatkozó változásokat, hanem ennek az eredménynek hatékonysági szempontból is jelentősége van: nem cseréljük

le a teljes szöveget, hanem csak ezeket a módosításokat hajtjuk végre rajta, így ki tudjuk használni a FingerTree perzisztenciáját.

Egyszerű, szerkesztési módosításoknál használhatjuk a szöveg karakter-szintű összevetését, hiszen ez általában néhány karakter beszúrását jelenti, a karakterek szintjén történő vizsgálat megfelelő finomságú eredményt ad. Ezzel ellentétben refaktorálás során nem célszerű karakterek szintjén végezni az összehasonlítást, hiszen itt fel kell készülni a komplexebb, egy fájlban belül több pontra kiterjedő változásokra is, ahol a karakteres vizsgálat túlságosan finom részletességű eredményt adna, ami később a módosítások konfliktusának vizsgálatánál jelentene hátrányt. A problémát szemléltetik az 5.2. rész példái. Ehelyett célszerű a vizsgálatot szavak szintjén végezni, azaz a szöveget fehérszóközök mentén felvágni, és az így kapott szeleteket tehát nem karakterenként vizsgálni, hanem teljes egészében. Ezzel elérjük, hogy a kiszámított változások ne legyenek túl finomak, és ne akadjanak össze a pozíciók konfliktus vizsgálatánál olyan esetekben is, amikor valójában nincs konfliktus.

4. fejezet

Összevetés verziókezelő rendszer megoldásaival

A dolgozat által tárgyalt probléma verziókezeléssel való hasonlósága miatt a következő fejezetben a fent leírt modellt összehasonlítom két verziókezelő rendszer megoldásával, modelljével. A két megközelítéssel nem kifejezetten mint verziókezelővel szeretnék foglalkozni, hanem azoknak az undo modellünkkel való fogalmi hasonlóságait kiemelve, így a változások, konfliktusok kezelését ismertetve.

4.1. Darcs

A Darcs [14] és Camp [16] verziókezelő rendszerek működését az úgynevezett patch-algebra [13] írja le. Ez a hagyományos verziókezelőktől eltérő módon nem fájl-fákon, hanem patch-ek, változások halmazán alapul, a repository aktuális állapotát patch-ek halmaza írja le. A modell egyik fontos tulajdonsága, hogy a változásokat szimmetrikusan kezeli, azok nem állnak rendezésben, nem előzi meg egyik változás a másikat, még ha időben fenn is áll köztük valamilyen sorrend. Ehelyett a patch-ek között részbenrendezési reláció áll fenn a függőségek alapján. Ennek oka, hogy két különböző felhasz-

nálótól származó patch érintheti ugyanazt a területet, de ebben a helyzetben nem adhatunk elsőbbséget egyik vagy másik változásnak, egyenrangúnak kell tekinteni őket. Ennek következtében a konfliktusok kezelése jóval bonyolultabbá válik, mintha kihasználhatnánk a változások között fennálló időrendi sorrendet, mint az tesszük a saját undo modellünkben. A patch-algebra konfliktusok kezelésére egy olyan algoritmust ad, mely nem teljes, sőt bizonyos esetekben exponenciális futási idejű, szemben a mi algoritmusunkkal, mely legrosszabb esetben is csak lineáris futási idejű. Ha a modellt refaktorálások undo műveletének megvalósítására használnánk, fent leírt probléma nem állna elő, hiszen csak egy felhasználótól érkeznek változások, azt azonban nem tudjuk garantálni, hogy ilyen speciális esetekben nem fordul elő exponenciális futási idejű összefésülés. A verziókezelők – így az itt tárgyalt két rendszer is – soronkénti összehasonlítást használnak a változások kiszámításához, azonban az eszköznek megadható külső program is a változások kiszámítására, így el tudnánk érni, hogy a saját modellünkhöz hasonlóan szavankénti vizsgálatot használva, finomabb eredményt kapjunk.

4.2. SVN

Az SVN [17] verziókezelő rendszert szintén használhatnánk a probléma modellezésére, megoldására. A soronkénti összehasonlítás miatt alapértelmezésben már egyszerű konfliktusok is problémát jelenthetnek, azonban a fenti eszközökhöz hasonlóan itt is használhatunk szavankénti összehasonlítást egy külső eszköz [4] segítségével, amivel a probléma egy lehetséges megoldását kapnánk.

4.3. Összegzés

Mindezeket összegezve azt mondhatjuk, hogy ezek az eszközök is használhatóak lennének a refaktorálás undo megoldására, azonban jóval általánosabb

célra készültek, mint ez a probléma, így nem képesek annak sajátosságait kihasználni. Az általunk készített modell ezzel szemben kifejezetten erre a feladatra lett specializálva, így jóval finomabb eredményt tudunk adni a konfliktusok kezelésére, jobban testreszabtuk a saját igényeinknek megfelelően, valamint hatékonyabb megoldást is adunk.

5. fejezet

Esettanulmányok

Az alábbiakban néhány példán keresztül szemléltetem a leírt modell működését. A példákban leírt pontokat az ábrák fejlécében levő számok segítségével lehet nyomon követni.

5.1. Konfliktus vizsgálata refaktorálás és szerkesztés esetén

Mindkét példában először végrehajtunk egy refaktorálást, majd szerkesztünk a fájlokban. Ezután megpróbáljuk visszavonni a refaktorálás változásait.

1. Példa (5.1. ábra).

1. a *pzip* függvényt áthelyezzük a *from* modulból az *xlists* modulba
2. szerkesztjük az *xlists* modult olyan területen, melyet a refaktorálás nem érintett: *flatsort* definíciójában töröljük az egyik sortörést
3. ezután visszavonjuk a refaktorálást a *from* modulban: a két változás által érintett terület nem lapol át, így a változásokat sikeresen össze-

fésülte; a refaktorálás változásai visszavonásra kerültek, a szöveg szerkesztett része változatlan maradt

2. Példa (5.2. ábra).

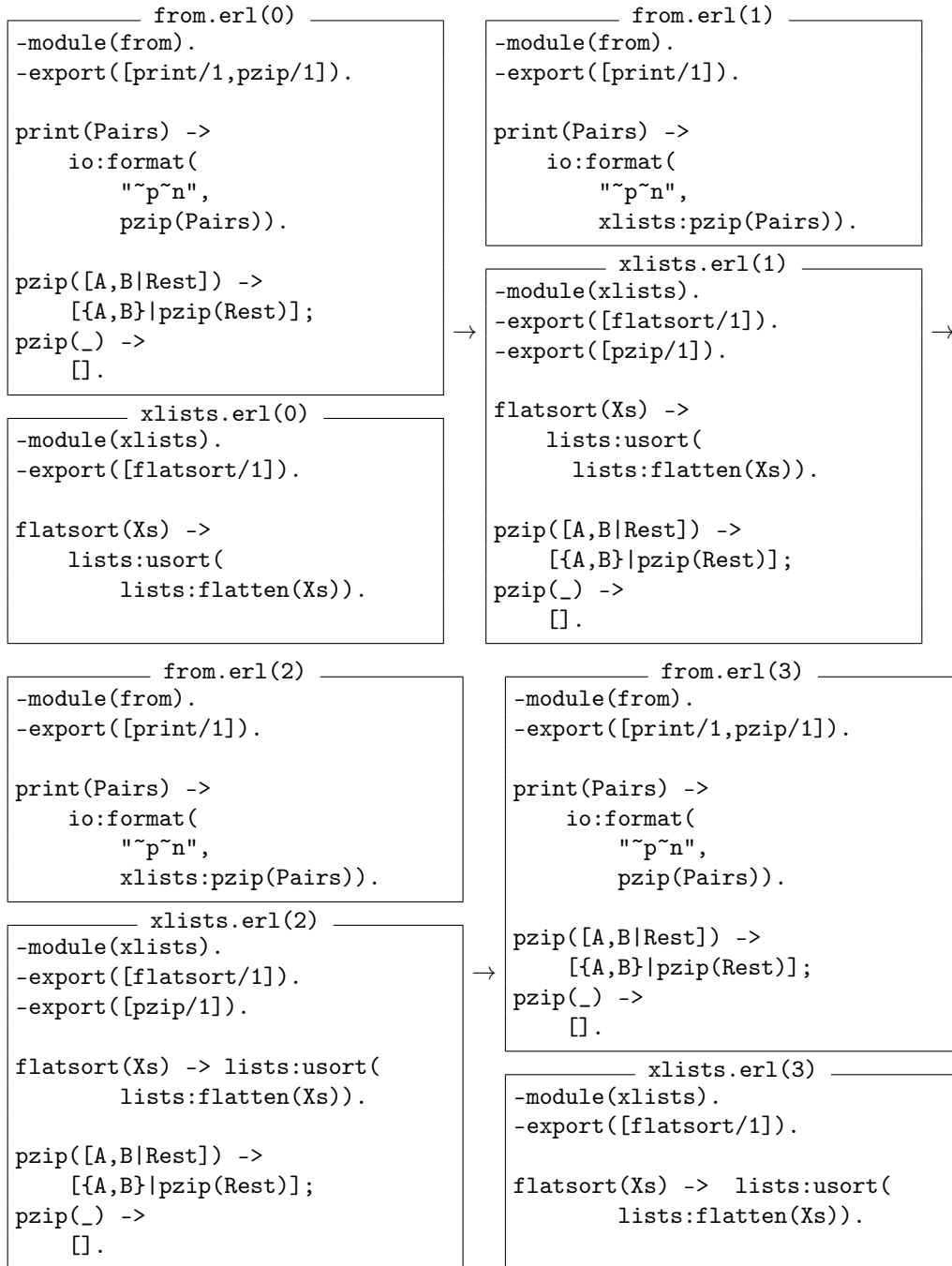
1. a *pzip* függvényt áthelyezzük a *from* modulból az *xlists* modulba
2. szerkesztjük az *xlists* modult a refaktorálás által érintett területen: a *pzip* függvény definíciójában törölünk egy sortörést
3. ezután visszavonjuk a refaktorálást a *from* modulban: a két változás területe átlapol, így konfliktus lép fel, a szerkesztés változtatásai is visszavonásra kerülnek, a sortörés újra bekerül a *pzip* függvény definíciójába

5.2. Vizsgálat szintje az összehasonlításnál

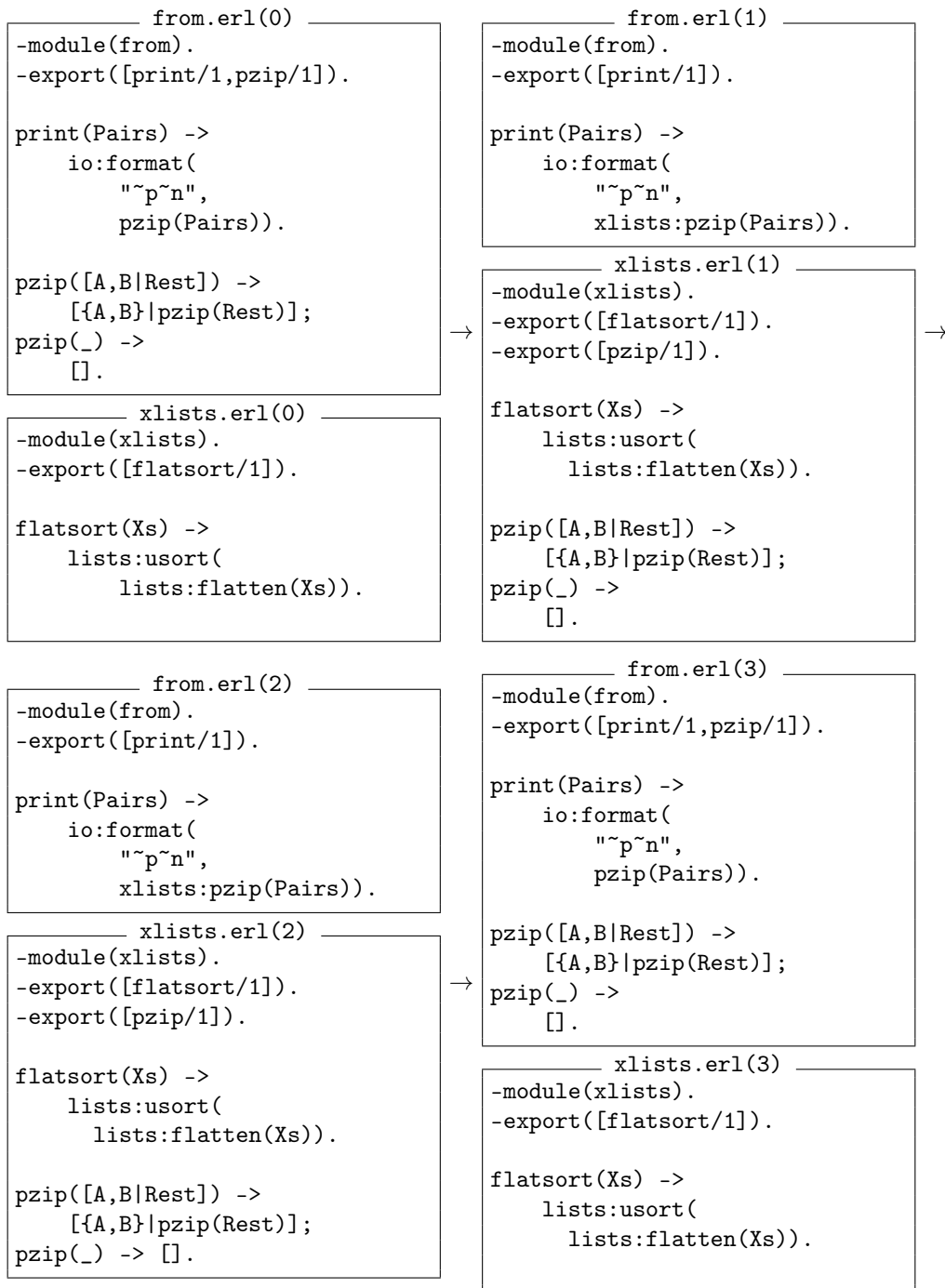
A következő példákban megmutatom, milyen különbséget jelent, hogy a változások számításakor karakterenként végezzük az összefésülést, vagy szavanként, a fehérszóközök mentén felvágva. A szemléltetéshez megadom a kiszámított változást is mindkét esetben az első refaktoráláshoz.

3. Példa (5.3. ábra). A változásokat karakterenként való összehasonlítással számoljuk.

1. A *g* függvényt áthelyezzük az *a* modulból a *b* modulba. A kiszámított változás: `Insert (25 "g/1]).\n -export(["`. Láthatjuk, hogy az algoritmus az `export`-listában a változást nem egyszerűen egy sor törlésének tekinti, hanem egy ennél finomabb eredményt ad a változásra.
2. A *h* függvényt áthelyezzük az *a* modulból a *c* modulba.



5.1. ábra. Függvény áthelyezése, szerkesztés (összefésülés)

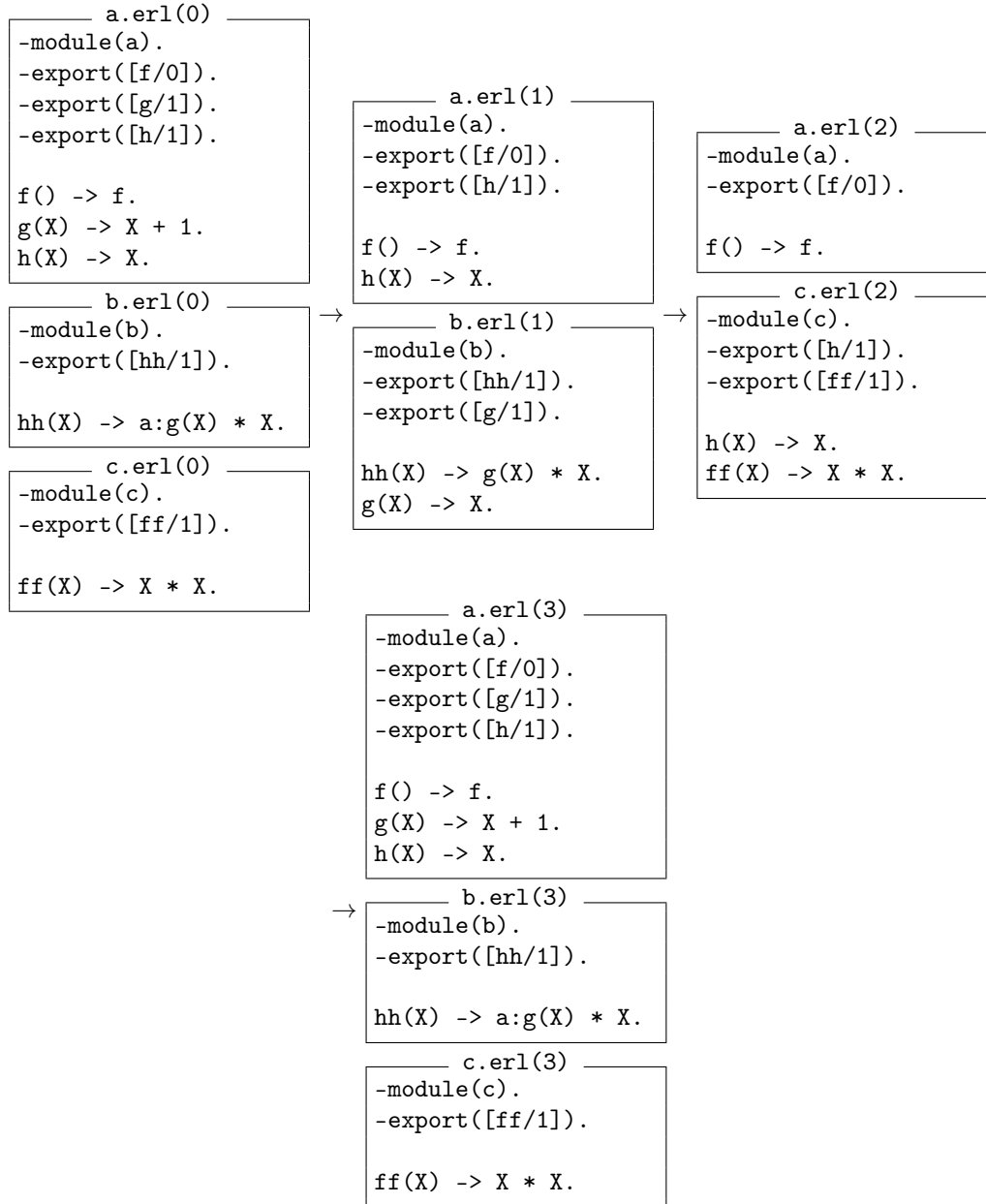


5.2. ábra. Függvény áthelyezése, szerkesztés (konfliktus)

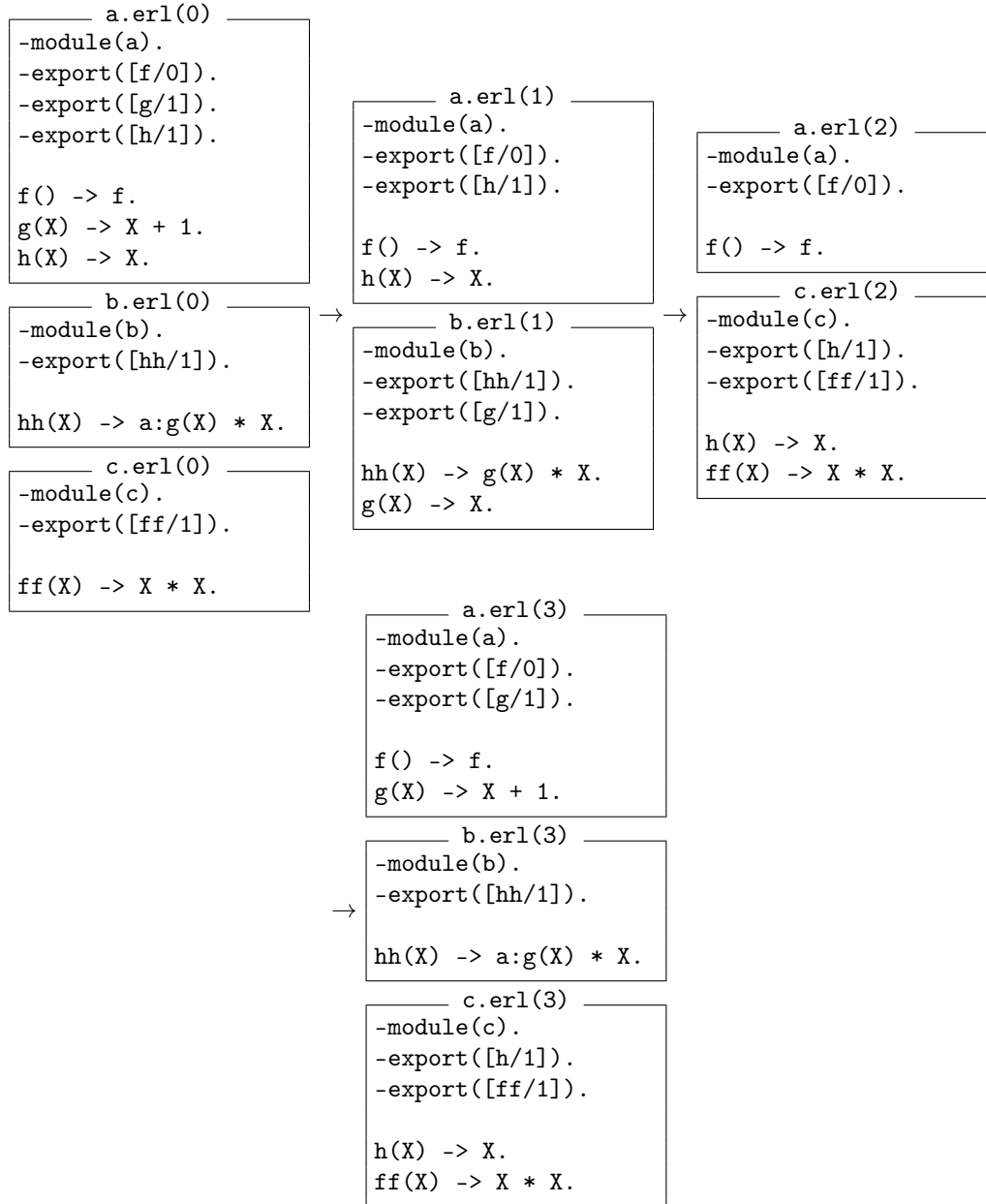
3. Ezután visszavonjuk a refaktorálást a b modulban: a két (1., 2.) változás által érintett terület látszólag nem lapol át az a modulban, azonban a karakterenkénti vizsgálat túl finom eredményt ad már az első refaktorálásnál, így konfliktus lép fel a pozíciók vizsgálatánál. Mindkét refaktorálás visszavonásra kerül, a fájlok az eredeti (0.) állapotnak megfelelő állapotba kerültek.

4. Példa (5.4. ábra). A változásokat szavanként való összehasonlítással számoljuk.

1. A g függvényt áthelyezzük az a modulból a b modulba. A változások számításának eredménye: `Insert (16 "-export([g/1]).\n")`. Az algoritmus az export-listában a változást elvárásainknak megfelelően egy sor törlésének tekinti.
2. A h függvényt áthelyezzük az a modulból a c modulba.
3. Ezután visszavonjuk a refaktorálást a b modulban: a két (1., 2.) változás által érintett terület nem lapol át az a modulban. A durvább vizsgálat most jobb eredményt ad az előzőhöz képest, valóban nincs konfliktus a pozíciók vizsgálatánál. Az a modulba visszakerül a g függvény, az időben ezután következett, c modult is érintő változás hatása pedig megmarad, sikeres összefésülés történt az a modul két változása közt.



5.3. ábra. Összehasonlítás karakterenként (konfliktus)



5.4. ábra. Összehasonlítás szavanként (sikeres összefésülés)

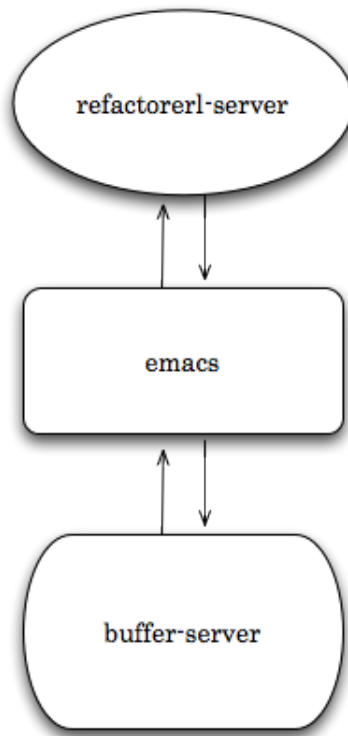
6. fejezet

RefactorErl undo/redo integrációja

Az egyetemünkön fejlesztett RefactorErl [6] erlang refaktoráló eszköz Emacs-felületéhez [12] integrálva elkészítettem a leírt refaktorálás undo modell egy megvalósítását. A program korábban is rendelkezett egy művelettel refaktorálások visszavonására, azonban ez nem volt kényelmesen használható, ugyanis minden a refaktorálás után történt módosítást visszavonta a refaktorálással együtt. Ezenkívül ez egy globális undo volt, azaz nem pufferek szintjén lehetett visszavonni az utolsó változást, hanem globálisan tekintve. Mivel ez a megoldás továbbfejlesztéssel sem lett volna képes egységesen kezelni egyszerre a szerkesztések és a refaktorálások változásait, illetve az undo során fellépő konfliktusokat, szükséges volt egy új módszerre.

Ehhez felügyelnünk kell valamilyen módon minden a fájlokon történt változást, annak tartalmát és a változásokat el kell tárolnunk. Ennek alapjául egy létező ötletet [2] használtam fel, mely a FingerTree adatszerkezetet használja a fájlok és a változások szövegének hatékony tárolására, illetve egy egyszerű, a refaktorálásokat nem kezelő undo/redo-mechanizmust is tartalmaz, mely a fent leírt szigorított lineáris undo modellen alapul. A szolgáltatást ezek felhasználásával egy különálló Haskell programmal valósítottam

meg, mely a standard bemeneten, kimeneten kommunikál a külvilággal, jelen esetben a RefactorErl Emacs-integrációjával, annak megfelelő programrészével. A kiegészített refaktoráló eszköz felépítését a 6.1. ábra mutatja. A refaktoráló eszköz (*refactorerl-server*) közvetlenül a szerkesztővel kommunikál (*emacs*), a felhasználói felületen indított kérések a refaktoráló eszközhöz érkeznek, a választ a szövegszerkesztő feldolgozza. Az undo/redo műveletet megvalósító, *buffer-server* néven jelölt komponens közvetlenül a szerkesztővel kommunikál szöveges kérésekben és válaszokban. A refaktoráló eszköz és a pufferkezelő egység közt a szövegszerkesztő jelenti a kapcsolatot: a refaktorálásokról üzenetet küld a *buffer-server*-nek, az undo/redo általi változásokról szintén üzenetet küld a *refactorerl-server*-nek.



6.1. ábra. Az eszköz felépítése

A program egységei közti kommunikációra definiáltam egy protokollt, mely néhány egyszerű parancsból áll:

- load <file>: betölti, vagy frissíti a megadott fájlt
- undo <file>: visszavonja a fájlban az utolsó változást
- redo <file>: megismétli a fájlban az utolsó változást
- begin-update: jelzi egy frissítési folyamat kezdetét
- end-update: jelzi egy frissítési folyamat végét
- list: kiírja a felügyelt fájlok neveit
- print <file>: kiírja a megadott fájl tartalmát
- rename <file> <file>: átnevezi az első megadott fájlt a második névre
- close <file>: kiveszi a megadott fájlt, a felügyelt bufferekből
- quit: leállítja a szerver

A protokollban a változások érkezésének kezdetét és végét jelző utasítások között érkeznek a tényleges frissítő utasítások. Így meg tudjuk különböztetni az egy fájlt érintő változásokat a többfájlosoktól.

Az alkalmazás futása során, az érkező kérések hatására megnyitott fájlokat egy-egy FingerTree-ben, a hozzájuk tartozó változásokat egy-egy undo-redo listában tárolja, ezenkívül a többfájlos refaktorálások azonosítóit és az érintett fájlokat is eltároljuk egy listában. A refaktoráló eszköz Emacs felületének elisp programját így ki kellett egészíteni néhány funkcióval: a Haskell-es szolgáltatás egy külön processzként fog megjelenni, illetve meg kellett valósítani a szöveges kommunikációt. Refaktorálás, illetve szerkesztés esetén a szövegszerkesztő jelez a buffer-szervernek, hogy frissítse az adott fájlt. Undo (redo) művelet meghívásakor a fájlra vonatkozó undo (redo) kérést küld a

szervernek, mely ennek hatására abban visszalép a megelőző állapotba - figyelve azt, hogy több fájl érintő refaktorálás volt-e az utolsó művelet -, majd kiírja a merevlemezre a módosított fájl(oka)t. A frissült tartalomról a szövegszerkesztő a következő formában kap értesítést: a bemeneten megkapja az undo hatására megváltozott fájlok listáját, majd ennek megfelelően újraolvassa az itt felsorolt fájlokat.

Az elészített eszköz fontos tulajdonsága, hogy környezetének csak a fent leírt egyszerű, szöveges protokoll szabályait kell teljesíteni, így könnyen integrálható a RefactorErl más felületéhez - pl. Vim [11] -, sőt akár más refaktoráló eszközhöz is.

Összegzés

Dolgozatomban a refaktorálások undo/redo művelete során felmerülő problémákkal, azok lehetséges kezelésével foglalkoztam. Ehhez megvizsgáltam néhány jelentősebb undo modellt, majd bemutattam egy saját, refaktorálásokra specializáltat. Ehhez kapcsolódóan írtam egy rövid ismertetőt a FingerTree nevű nevű adatszerkezetről, mely hatékonyan használható szöveg tárolására, így szövegpuffer megvalósítására. A refaktorálás undo modellben ismerttettem a változások kezelésének, számításának egy módszerét, illetve megadtam egy heurisztikát refaktorálások visszavonásakor fellépő konfliktusok kezelésére. Néhány példán keresztül bemutattam a heurisztika működését, illetve bemutattam egy megvalósítását a leírt modellnek, mely a RefactorErl Emacs-felületébe beépülve teszi lehetővé refaktorálások undo/redo műveletét. A szolgáltatás a legutóbbi 0.9.11.08-es kiadásban már elérhető. Mind a modell, mind ez a megvalósítás kellően általános ahhoz, hogy jól csatolható legyen más eszközhöz, más felülethez, így a jövőben meg szeretnénk valósítani ezt a funkciót a refaktoráló eszköz Vim felületéhez is. Az így kapott modell egy erőteljes, jól használható eszközt ad programok refaktorálása során.

Irodalomjegyzék

- [1] Thomas Berlage: *A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects* ACM Trans. Comput.-Hum. Interact. 1(3), 269-294, 1994
- [2] Yi text editor homepage.
<http://www.haskell.org/haskellwiki/Yi>.
- [3] diff algorithm
<http://en.wikipedia.org/wiki/Diff>
- [4] GNU wdiff.
<http://www.gnu.org/software/wdiff>
- [5] Martin Fowler: *Refactoring - Improving the design of existing code*, Addison Wesley, 2000, ISBN-0201485672
- [6] RefactorErl - Refactoring Erlang programs.
<http://plc.inf.elte.hu/erlang>
- [7] Finger tree - Wikipedia.
http://en.wikipedia.org/wiki/Finger_tree, 2011.03.
- [8] Finally: Finger Trees!
http://scienceblogs.com/goodmath/2009/05/finally_finger_trees.php, 2011.04.

- [9] Ralf Hinze, Ross Paterson: *Finger Trees: A Simple General-purpose Data Structure*, Journal of Functional Programming, 16:2, 197-217, 2006, <http://www.soi.city.ac.uk/~ross/papers/FingerTree.html>, 2011.03.20.
- [10] Finger Tree: A Functional Value Object
<http://www.dalnfre.com/wp/2011/09/finger-tree-a-functional-value-object>, 2011. 11.
- [11] vim - the editor.
<http://www.vim.org>
- [12] GNU Emacs text editor homepage.
<http://www.gnu.org/software/emacs>
- [13] Ian Lynagh: *An algebra of Patches*, 2006. 10. 30.
<http://urchin.earth.li/~ian/conflictors/paper-2006-10-30.pdf>, 2011. 12.
- [14] darcs revision system.
<http://darcs.net>
- [15] Darcs - Wikipedia.
<http://en.wikipedia.org/wiki/Darcs>, 2011. 12.
- [16] Camp version control system.
<http://projects.haskell.org/camp>
- [17] SVN - Apache Subversion.
<http://subversion.apache.org>