Ohjelmallinen transaktiomuisti

Tuomas Jorma Juhani Räsänen
tuos@jyu.fi

10.4.2008

Tiivistelmä

Säikeiden synkronointi lukkojen avulla on työlästä ja hyvin virhealtista. Lukkoihin perustuvat toteutukset eivät myöskään ole modulaarisia. Moniydinsuorittimien vallatessa tavallisten käyttäjien kotikoneita, on ohjelmoijien löydettävä parempia keinoja saadakseen näiden uusien suorittimien tehot hyötykäyttöön ja hallitakseen entisestään kasvavien ohjelmistojen kompleksisuutta. Tässä paperissa esittelen erästä ratkaisua näihin ohjelmoijan kohtaamiin uusiin haasteisiin, ohjelmallista transaktiomuistia.

1 Johdanto

Tietokoneiden laskentatehon kasvu on seurannut yli 40 vuotta Gordon E. Mooren ennustusta [Moo65]. Nyt on kuitenkin saavuttu pisteeseen, jossa ainoastaan mikropiirin transistorien määrää lisäämällä laskentatehon kasvattaminen ei enää käytännön syistä ole mahdollista [ABC+06, SL05]. Transistorien määrää nostamalla nostamalla, mikropiirin virrankulutus kasvaa nopeasti. Virrankulutus tuottaa luonnollisesti välittömiä kuluja, mutta vielä kriittisempää on virrankulutuksen aiheuttama piirin kuumeneminen. Tehokkaimpien nykysuorittimien jäähdytys alkaakin tuottaa käytännön ongelmia. Laskentatehon kasvattamiseksi tietokoneiden suorittimien määrää kasvatetaan tai käytetään suorittimia joilla on useita laskentaytimiä. Monisuoritinkoneet eivät ole kovinkaan uusi keksintö, mutta nyt ne ovat saavuttaneet massamarkkinat ja saapuvat vauhdilla tavallisten ihmisten koteihin. Tällaiset järjestelmät, joissa samaa jaettua muistia käyttävät useat laskentaresurssit, aiheuttavat suuria haasteita ohjelmistokehitykselle.

Perinteiset menetelmät samanaikaisuuden (engl. concurrency) hallintaan, kuten lukot ja monitorit, toimivat hyvin matalalla tasolla ja niiden käyttö oh-

jelmakoodissa johtaa helposti ohjelmavirheiden syntyyn. Kehittäjät tarvitsevat uusia, helpompia ja modulaarisempia abstraktioita ohjelmien rinnakkaistamiseen. Eräs lupaava parannusehdotus tilanteeseen on viime aikoina aktiivisesti tutkittu ohjelmallinen transaktiomuisti (engl. Software Transactional Memory) [ST95, HMPJH05]. Tässä tutkielmassa viittaan STM-lyhenteellä Harrisin ja muiden esittelemään toteutukseen ohjelmallisesta transaktiomuistista The Glasgow Haskell Compiler (GHC) -kääntäjälle [HMPJH05, Has].

Tämä tutkielma on järjestetty siten, että johdannon jälkeen luvussa 2 esitetään nykyisin vallalla olevien abstraktioiden ongelmia. Luvussa 4 esitetaan ohjelmallisen transaktiomuistin pääpiirteet ja havainnollistetaan sen toimintaa esimerkein Haskell-ohjelmointikielellä. Luvussa **??** pohditaan transaktiomuistin hyötyjä ja haittoja perinteisiin menetelmiin nähden ja lopuksi luvussa 6 on yhteenveto tutkielmassa käsitellyistä aiheista.

2 Lukkojen ongelmat

Jaettua muistia käsittelevien säikeiden synkronointi lukkojen avulla on sovellusohjelmoijalle hyvin työlästä ja vaikeaa. Lukoilla on helppoa tehdä vaikeasti havaittavia virheitä ohjelmakoodiin. Säikeitä synkronoitaessa lukkojen avulla ohjelmoijan pitää muistaa seuraavat asiat:

- Jaetut resurssit pitää muistaa suojata lukoilla. Lukkojen unohtaminen johtaa mahdollisesti virheelliseen ohjelman tilaan. Ohjelmoijan pitää muistaa varata tarvittavat lukot jokaisessa paikkassa, jossa jaettua resurssia käsitellään.

- Lukot pitää muistaa varata oikeassa järjestyksessä. Lukkojen varaaminen väärässä järjestyksessä voi johtaa lukkiutumiseen (engl. dead lock).

- Lukot pitää muistaa myös vaputtaa. Resurssien jääminen lukkoon estää muiden resurssia tarvitsevien säikeiden suorituksen.

- Lukitus pitää osata tehdä oikealla karkeustasolla. Liian karkea lukitustaso (engl. coarse-grained locking) tarkoittaa ohjelman kriittisten alueiden (engl. critical section) suojaamista hyvin pienellä määrällä lukkoja. Se johtaa väistämättä huonoon ohjelman suorituskyvyn skaalautuvuuteen säikeiden määrän lisääntyessä. Monisuoritinkoneessa ajettava monisäikeinen ohjelma ei pysty käyttämään kaikkien suorittimien tehoja hyväksi mikäli ohjelman suoritus on rajoitettu lukoilla vain yhden säikeen käsiteltäväksi. Hieno lukitustaso (engl. fine-grained locking) taas monimutkaistaa ohjelmakoodia nopeasti edellä mainittujen syiden seurauksena.

Simon Peyton-Jones ja muut korostavat lisäksi [PJ07, HMPJH05], että lukkoihin perustuvista synkronointiabstraktioista ei voida yhdistelemällä muodostaa oikein toimivia korkeampia abstraktioita. Tämä on heidän mukaansa lukkojen yksi merkittävimmistä puutteista.

3 Yksinkertainen pankkisovellus lukoilla

Seuraavat lähdekoodiesimerkit havainnollistavat edellämainittuja ongelmia lukkoabstraktion käytössä.

Lähdekoodilistaus 1: `Account`-luokan toteutus

```
public class Account {

  private double balance;
  private String owner;

  public synchronized void deposit(double sum){
    balance = balance + sum;
  }

  public synchronized void withdraw(double sum) throws AccountError{
    if (balance < sum)
      throw new AccountError("Insufficient balance.");
    balance = balance - sum;
  }

  public synchronized String getOwner(){
    return owner;
  }

  public synchronized void setOwner(String newOwner){
    owner = newOwner;
  }
}
```

Esimerkissä on toteutettu luokka kuvaamaan hyvin yksinkertaistettua tiliä. Tilillä on attribuuttina saldo ja kaksi metodia, `deposit` ja `withdraw`, joilla tilille voidaan tallettaa tai sieltä nostaa rahaa. Noston yhteydessä tarkistetaan onko tilillä nostoa varten riittävästi rahaa. Mikäli saldo on nostettavaa summaa pienempi, nostoa ei suoriteta ja heitetään poikkeus. Tilille tallettaminen onnistuu aina. `withdraw`-metodin allekirjoituksessa on ilmoi-

tettu, että se voi heittää poikkeuksen epäonnistuessaan. Tilille voidaan myös asettaa omistaja.

Javan varattu sana `synchronized` metodien allekirjoituksessa kertoo, että metodia kutsuttaessa lukitaan metodin omistava olio. Jokainen Javan olio sisältää lukon synkronointia varten. Täten taataan kutsujalle, ettei mistään muusta säikeestä päästä kutsumaan olion `synchronized`-metodeita.

Edellä esitetty `Account`-luokka on säieturvallinen. Se on niin sanottu monitori (engl. monitor) [Han72]. Monitori on olio, jonka tilaa voi muuttaa vain yksi säie kerrallaan kontrolloitujen metodien kautta. Metodikutsut lukitsevat olion kutsun suorituksen ajaksi. Javassa tämä on toteutettu nimenomaisesti `synchronized`-avainsanalla metodien allekirjoituksessa. Luokka on siis säieturvallinen vain, koska jokainen metodi on muistettu varustaa synkronoinnin takaavalla taikasanalla. Säieturvallisuudesta huolimatta `Account`-luokan toteutus esittelee kuitenkin monisäikeisessä ympäristössä huonoon suorituskykyyn johtavan ongelman: lukitus on hoidettu hyvin karkealla tasolla. Säikeen kutsuessa esimerkiksi `deposit`-metodia olio on lukittu kutsujalle, eikä mikään muu säie voi kutsua olion metodeja. Tämä ei ole tarkoituksenmukaista esimerkiksi silloin, kun toinen säie haluaisi kutsua `setOwner`-metodia. Nämä kaksi metodia käsittelevät toisistaan riippumattomia attribuutteja ja näitä tulisi sen vuoksi voida kutsua samanaikaisesti. Esimerkkitoteutus ei tätä kuitenkaan mahdollista ja rajoittaa siten ohjelman suorittamista samanaikaisesti useammassa säikeessä. Ongelman voisi luonnollisesti kiertää esittelemällä eksplisiittisesti omat lukot molemmille attribuuteille ja muokkaamalla attribuutteja käsittelevät metodit varaamaan ja vapauttamaan nämä lukot. Menetelmä lisää kuitenkin toteutuksen kompleksisuutta merkittävästi ja rohkaiseekin etsimään vaihtoehtoisia ratkaisuja.

Esitellään lisäksi luokka kuvaamaan pankkia ja sen tarjoamia muutamaa toimenpidettä:

Lähdekoodilistaus 2: `Bank`-luokan toteutus

```
public class Bank {

  private Account interests;
  private Account managersPersonalAccount;

  public synchronized void transfer(Account from, Account to, double amount)
      throws AccountError{
    from.withdraw(amount);
    to.deposit(amount);
  }
```

```
  public synchronized void acquitInvoice(Account to, double amount){
    try {
      transfer(interests, to, amount);
    } catch (AccountError e) {
      try {
        transfer(managersPersonalAccount, to, amount);
      } catch (AccountError e) {
        // Block until there is money and then retry.
      }
    }
  }
}
```

`Bank`-luokan metodi `transfer` luo käsitteen tilisiirrolle tilin primitiivioperaatioiden `deposit` ja `withdraw` avulla. Esimerkkitoteutuksessa on kuitenkin kaksi perustavanlaatuista ongelmaa. Ensinnäkin, toteutus ei takaa, että lähdetililtä nosto ja kohdetilille suoritettava talletus tapahtuisivat peräkkäin. Voi olla täysin mahdollista, että jokin toinen säie saa suoritusvuoron `withdraw`-metodikutsun jälkeen. Tällöin ohjelman tila on virheellinen. Lähdetililtä on nostettu rahaa, mutta sitä ei vielä ole talletettu kohdetilille. Tälläinen ominaisuus ei ole kovin toivottavaa pankkisovellukselta. Näin ollen `transfer`-metodia ei voida koostaa `Account`-luokan esittelemien metodien pohjalta. Jotta tilisiirto voitaisiin toteuttaa oikein, "kaikki tai ei mitään" -periaatteella, tulisi `transfer`-metodin olla mahdollista lukita sekä lähdetili että kohdetili:

Lähdekoodilistaus 3: Korjattu `transfer`-metodi

```
public synchronized void transfer(Account from, Account to, double sum){
  from.lock();
  to.lock();
  from.withdraw(sum);
  to.deposit(sum);
  to.unlock();
  from.unlock();
}
```

Nyt `deposit`- ja `withdraw`-metodit paljastavat sisäisestä toteutuksestaan lukot eivätkä enää siten ole modulaarisia. Lukkoihin perustuvista toteutuksista ei voi siis yhdistelemällä rakentaa laajempia toteutuksia.

Toisekseen, toteutuksessa ei ole virheiden käsittelyä. Mikäli `deposit`-metodin kutsuminen johtaisi virhetilanteeseen ja tilisiirron peruuntumiseen, pitäisi rahat palauttaa tilille, jonka `withdraw`-metodia kutsuttiin. Tähän esi-

merkkiin tällainen operaation peruminen olisi vielä suhteellisen yksinkertaista toteuttaa, mutta monimutkaisempien operaatioiden kohdalla tilan palauttaminen alkuperäiseksi olisi jo huomattavasti haasteellisempaa.

Pankki maksaa tileille korkoa `acquitInvoice`-metodilla. Korko maksetaan joko korkoja säilövältä tililtä tai mikäli siellä ei ole rahaa niin korko maksetaan pankijohtajan omasta pussista. Mikäli pankinjohtajallakaan ei ole varaa, tulisi maksusuorituksen jäädä odottamaan varojen karttumista.

Sisäkkäiset ehtorakenteet, operaation uudelleensuoritus, virheiden tarkistus ja "kaikki tai ei mitään"-periaatteen toteuttaminen yhdessä tekee säieturvallisen sekä modulaarisen ohjelman toteuttamisesta lukkojen avulla huomattavan työlästä ja virhealtista [Lee06, ATKS07, PJ07].

4 Transaktiomuisti

Eräs lukoille vaihtoehtoinen malli samanaikaisuuden hallintaan juontaa juurensa tietokantojen transaktioperiaatteesta. Herlihy ja Moss esittelivät paperissaan [HM93] transaktiomuistin periaatteen jaetun muistin lukottomaan synkronointiin. Heidän ehdotuksensa perustui laitteistotason arkkitehtuuriratkaisuun. Shavit ja Touitou esittelivät ensimäisinä ajatuksen täysin ohjelmallisesta transkatiomuistista [ST95]. Toimintaperiaatteiltaan molemmat ratkaisut ovat toistensa kaltaisia ja perustuivat tietokannoista tuttujen transaktioiden ominaisuuksiin. Tässä luvussa sekä esimerkein havainnolistavassa aliluvussa 5 käsitellään Harrisin ja muiden esittelemää mallia ohjelmallisesta transaktiomuistista [HMPJH05]. Se eroaa Shavitin ja Touitoun ehdotuksesta muutaman laajennoksen osalta joihin palataan myöhemmin.

Transaktion kaksi keskeisintä ominaisuutta ovat atomisuus (engl. atomicity) ja eristyneisyys (engl. isolation) [HM93, PJ07]. Atomisuudella tarkoitetaan luvussa 3 kuvattua "kaikki tai ei mitään" -periaatetta. Atomisen lohkon kaikki operaatiot suoritetaan peräkkäin ja näin ollen muut säikeet eivät saa suoritusvuoroa lohkon operaatioiden välissä. Atominen lohko näkyy siis muille säikeille yhtenä jaottomana (engl. indivisible) operaationa. Eristyneisyys tarkoittaa transaktion olevan sen suorituksen ajan täysin riippumaton muista säikeistä. Transaktion alussa ohjelman tila eristetään ja transaktio muuttaa nimenomaisesti tätä eristettyä tilaa. Näin transaktio ei häiriinny muiden säikeiden toimenpiteistä eikä myöskään transaktiota suorittavan säikeen toiminta häiritse muita säikeitä.

5 Yksinkertainen pankkisovellus muistitransaktioilla

Tarkastellaan STM:n ominaisuuksia toteuttamalla luvussa 3 esitetty yksinkertainen pankkisovellus muistitransaktioiden avulla. Esitellään tilille oma tietue:

```
data Account = Account { balanceOf :: TVar Double
                       , ownerOf :: TVar String }
```

Tilille on määritetty kaksi kenttää kuvaamaan tilin saldoa sekä omistajaa. Tyyppi `TVar` on määriteltynä `Control.Concurrent.STM`-modulissa ja se on viite jaettuun muistipaikkaan. `TVar`-tyyppistä viitettä voidaan käsitellä seuraavilla komennoilla.

```
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Komento `newTVar` luo uuden transaktiomuuttujan ja asettaa sille alkuarvon. Komento `readTVar` lukee muuttujan arvon ja `writeTVar` asettaa muuttujan parametrina saadun arvon. Edellä mainitut komennot toimivat ja palauttavat arvon STM-monadissa.

Määritetään tilille `deposit`-komento:

```
deposit :: Account -> Double -> STM ()
deposit to amount = do balance <- get balanceOf to
                       set balanceOf to (balance + amount)
```

Komento saa parametrinaan tilin ja sille talletettavan summan. Komennon tyypistä nähdään sen operoivan STM-monadissa ja palauttamatta mitään mielekästä arvoa. Toteutus itsessään on varsin luettava jopa ilman merkittävää ymmärrystä Haskellista. Komento pyytää tililtä sen nykyisen saldon ja asettaa tilille uuden saldon, joka on vanhan saldon ja lisättävän rahamäärän summa.

Komento `withdraw` on lähes yhtä suoraviivaisesti toteutettu:

```
withdraw :: Account -> Double -> STM ()
withdraw from amount = do
  balance <- get balanceOf from
  case balance < amount of
    True  -> retry
    False -> set balanceOf from (balance - amount)
```

Suurimpana erona luvussa 3 esitettyyn vastaavaan Java-toteutukseen on toimenpide, joka suoritetaan tilin saldon ollessa tililtä nostettavaa summaa

pienempi. Haskell-toteutuksessa ei heitetä poikkeusta vaan suoritetaan `retry`-komento, joka käskee transaktion aloittamaan suorituksensa alusta. Tarkemmin `retry`:n semantiikkaan palataan myöhemmin tässä luvussa.

Java-toteutuksen `transfer`-metodissa ongelmana oli metodin atomisuuden takaaminen ilman `deposit`- ja `withdraw`-metodien toteutuksen paljastamista. Metodi pyrittiin toteuttamaan yhdistämällä aiemmin määritellyt primitiivioperaatiot ja luomalla siten uusi tiliä käsittelevä abstraktio. Lukoilla toteutettu synkronointi ei tätä kuitenkaan mahdollistanut. Mahdollinen `transfer`-komennon Haskell-toteutus näyttää seuraavalta:

```
transfer :: Account -> Account -> Double -> STM ()
transfer from to amount = do withdraw from amount
                             deposit to amount
```

Komento on suoraviivainen yhdistelmä komennoista `deposit` ja `withdraw`. Komento itsessään ei vielä takaa sen toimivan säieturvallisen transaktion tapaan. Toisaalta, komento itsessään on käyttökelvoton Haskell-ohjelmassa, jonka päätaso on `main`-komento:

```
main :: IO ()
```

STM-komentoja pitää siis pystyä kutsumaan IO-monadista. Tätä varten STM-moduli määrittää komennon:

```
atomically :: STM a -> IO a
```

Transaktiomaisuuden takaava `atomically`-komento on koko STM:n ydin. Sille annetaan parametrina STM-komento, jonka se suorittaa taaten suoritettavalle komennolle atomisuuden ja eristyneisyyden. On huomattavaa, että suoritettava STM-komento voi olla itsessään yhdistelmä muista STM-komennoista edellä esitetyn `transfer` -komennon tapaan. Tilisiirto voisi toimia esimerkiksi näin:

```
main :: IO ()
main = do a <- atomically $ mkAccount 50.0 "James"
          b <- atomically $ mkAccount 90.0 "Alice"
          atomically $ transfer a b 25.0
          putStrLn "Transfer completed."
```

STM:n transaktiot ovat toteutettu optimistisen synkronoinnin periaatteella. Transaktion aloittaa `atomically`-komennon kutsu. Kutsuvalle säikeelle luodaan tyhjä transaktioloki. Transaktion `writeTVar`-kutsun yhteydessä lokiin merkitään käsiteltävän `TVar`-muuttujan osoite, muuttujan vanha arvo sekä muuttujalle asetettu uusi arvo. `TVar`-muuttujia lukevan `readTVar`-kutsun kohdalla lokista etsitään onko luettavaan muuttujaan aikaisemmin transaktion aikana kirjoitettu arvoa. Mikäli ei ole, luetaan muuttujan arvo

muistista ja merkitään lokiin muuttujan osoite sekä luettu arvo. Uudeksi arvoksi merkitään vanha arvo. Lukuoperaatio ei siis muuta muuttujan arvoa. Huomattavaa on ettei missään vaiheessa transaktion suorituksen aikana kirjoiteta suoraan muistiin vaan muutokset merkitään transaktiolokiin. Transaktion lopussa suoritetaan lokin tarkistus (engl. validation) ja muutosten vahvistus (engl. commit). Lokin tarkistuksessa tarkastetaan lokin merkittyjen muuttujien vanhojen arvojen ja muistissa olevien arvojen yhtäläisyys. Jos muistin tila on pysynyt lokin mukaan muuttumattomana, kirjoitetaan lokin kirjoitusmerkintöjen arvot muistiin ja transaktion suoritus päättyy. Mikäli jonkin merkinnän vanha arvo eroaa muistissa olevasta arvosta, jokin toinen säie on muuttanut kyseistä muuttujaa transaktion aikana. Tällöin transaktion tilan sanotaan olevan ristiriidassa muistin kanssa. Ristiriitatilanteessa transaktio suoritetaan uudestaan. Tällöin transaktiolokin sisältö voidaan huoletta jättää huomiotta ja uudelleensuoritus aloittaa tyhjällä transaktiolokilla.

Tämä on mahdollista vain, koska STM-monadin komennot eivät voi aiheuttaa peruuttamattomia sivuvaikutuksia. STM-komennot ovat rajoitettu Haskellin vahvan tyyppisysteemin avulla suorittamaan vain `TVar`-tyyppisiä muistiviitteitä käsitteleviä komentoja. STM-komennot eivät siis pysty esimerkiksi tulostamaan `putStrLn`-komennolla näytölle. Syy tähän löytyy STM:n optimistisesta ja lukottomasta synkronoinnista sekä transaktioiden ominaisuudesta peruuttaa (engl. rollback) suoritetut toimenpiteet mikäli transaktion suoritus jostain syystä epäonnistuu. Peyton-Jonesin varsin hilpeä esimerkki [PJ07] havainnollistaa erinomaisesti syyn IO-operaation poissulkemiseen STM-komennoista:

```
launchMissiles :: IO ()

atomically $ do x <- readTVar xv
                y <- readTVar yv
                if x>y then launchMissiles
                       else return ()
```

Atomisen lohkon suoritus ilmiselvästi saattaa aiheuttaa vakavia peruuttamattomia sivuvaikutuksia. Mikäli lohkon lopussa suoritettavassa transaktiolokin tarkistuksessa ilmenee luettujen muistipaikkojen arvojen olevan virheelliset, perutaan suoritetut operaatiot tyhjentämällä loki ja suoritetaan transaktio uudestaan. Jos ohjukset tuli laukaistua lohkon suorituksessa, niiden laukaisua ei enää pystytä perumaan. Pahempaa on luvassa, kun transaktio suoritetaan uudestaan. Tämä saattaa aiheuttaa uuden ohjuspatterin laukaisun.

Aiemmin tässä luvussa esitellyssä `withdraw`-komennossa kutsutaan `retry`:a tilin saldon ollessa nostettavaa summaa pienempi. Ohjelmoijalle komennon semantiikka on hyvin yksinkertainen: transaktio suoritetaan uudestaan alusta tyhjällä transaktiolokilla. Mielenkiintoinen kysymys toteutuksen kannalta on milloin uudelleensuoritus tulisi aloittaa. Jos transaktio suoritetaan välittömästi uudestaan, on hyvin todennäköistä, että luettavat transaktiomuuttujat sisältävät samat arvot kuin edellisellä suorituskerralla. Tällöin suoritus alkaisi jälleen alusta. Tämä menetelmä kuluttaa tarpeettomasti suoritintehoa. Parempi menetelmä on käyttä hyväksi transaktiolokiin säilytettyä tietoa transaktion aikana luetuista muuttujista. Uudelleensuoritus on suorituskyvyn kannalta järkevää tehdä vain jos jonkin transaktion aikana luettavan muuttujan arvo on muuttunut.

`Bank`-luokan `acquitInvoice`-metodissa suoritetaan korkojen maksu asiakkaan tilille pankin korkotililtä tai korkotilin saldon ollessa liian pieni maksu suoritetaan pankinjohtajan tililtä. Jos johtajan tililläkään ei ole rahaa, yritetään maksun suoritusta alusta uudestaan. Kysymyksessä on siis ehtorakenne, jossa suoritus määräytyy edellisen operaation onnistumisen mukaan. STM-moduli esittelee vastaavanlaista rakennetta varten komennon:

```
orElse :: STM a -> STM a -> STM a
```

Komento yrittää suorittaa ensimäisenä parametrina saadun STM-komennon. Mikäli tämä komento päätyy yrittämään uudelleen, joko ohjelmoijan kirjoittaman `retry`-komennon seurauksena tai transaktiolokin ja muistin ristiriidasta, suoritetaan toisena parametrina saatu STM-komento. Paluuarvona `orElse`-komento palauttaa suoritetun parametrin paluuarvon. Edellä esitetyn komennon avulla voidaan kirjoittaa `acquitInvoice`-komento varsin selkeästi:

```
acquitInvoice :: Bank -> Account -> Double -> STM ()
acquitInvoice bank to amount =
    fromInterest `orElse` fromManager `orElse` retry
    where fromInterest = do interestAccount <- getInterestsAccount bank
                            transfer interestAccount to amount
          fromManager = do managersAccount <- getManagersAccount bank
```

Siinä yritetään maksaa korot ensin pankin tililtä tai mikäli `fromInterest`-komento päätyy yrittämään uudelleen, maksu suoritetaan pankinjohtajan tililtä. Jos kumpikin maksusuoritus epäonnistuu, suoritetaan koko `acquitInvoice`-komento uudestaan.

6 Yhteenveto

Aiemmissa luvuissa on esitetty lukkojen perustavanlaatuisia ongelmia. Ne tekevät säikeiden synkronoinnista erittäin haastellista. Lukkojen yksi suurimmista ongelmista on mahdollisuus pelättyyn lukkiutumiseen. Lukkiutuminen on tuhoisaa ohjelman suoritukselle ja syitä lukkiutumiseen on vaikea löytää lähdekoodista. Ohjelmistot ovat yhä vaikeammin hallittavia ja uudet massamarkkinoille saapuneet moniydinsuorittimet pakottavat tavallisetkin ohjelmoijat kohtaamaan samanaikaisuuden aiheuttaman ylimääräisen monimutkaisuuden. Lukkojen perustavanlaatuinen ongelma on vaikeus piilottaa ne korkeampien abstraktioiden sisälle. Ohjelmoijan tärkein keino kompleksisuuden hallintaan on abstrahointi, mikä pakottaakin uusien synkronointimenetelmien etsimisen.

Ohjelmallinen transaktiomuisti on hyvin lupaava menetelmä hallita monien säikeiden synkronointia. Se vapauttaa ohjelmat täysin lukkiutumisen vaarasta, sillä lukkoja ei yksinkertaisesti käytetä. Luvussa 5 nähtiin miten transaktioita yhdistelemällä on mahdollista luoda korkeamman tason toimivia operaatioita. Haskellin vahva tyyppijärjestelmä tekee STM:n käytöstä entistä turvallisempaa. STM-komentojen sivuvaikutuksettomuus sekä jaettujen muistipaikkojen transaktiomainen käsittely voidaan täysin taata käännösaikaisella tyyppitarkistuksella. STM ei kuitenkaan ratkaise kaikkia säikeistettyjen ohjelmien virheitä, mutta onnistuu ainakin poistamaan kaikkein vaikemmin havaittavia ja tekemään monisäikeisestä ohjelmasta hieman helpommin ymmärrettävän. Avoimeksi kysymykseksi tämän tutkielman osalta jäi STM:n todellinen suorituskyky aidosti monisäikeisessä ympäristössä. Tämä kysymys tarjoaa kirjoittajalle mielenkiintoisen lähtökohdan jatkaa STM:n ja muiden säikeiden synkronointia helpottavien menetelmien tutkimista.

Lähteet

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ATKS07] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2007.

[Han72] Per Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.

[Has] Haskell.Org. The glasgow haskell compiler. World Wide Web electronic publication: http://www.haskell.org/ghc/index.html.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electornics Magazine*, 38(8), April 1965.

[PJ07] Simon Peyton-Jones. Beautiful concurrency. In *Beautiful code*. O'Reilly, 2007.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.