VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Laisvosios monados ir transakcinė atmintis

Free monads and software transactional memory

Bakalauro darbas

Atliko: 4 kurso 1 grupės studentas
Martynas Lyčius (parašas)

Darbo vadovas: lekt. Viačeslav Pozdniakov (parašas)

Darbo recenzentas: prof. dr. Romas Baronas (parašas)

Vilnius – 2016

Santrauka

Šiame darbe nagrinėjama atminties valdymo problema, kai keli vykstantys procesai konkuruoja dėl atminties resurso. Problemą bandoma spręsti pasitelkiant transakcinės atminties mechanizmą (STM), kurio principas yra operacijų su atmintimi vykdymas transakcijomis. Darbo metu bandoma išnagrinėti ir įgyvendinti tokį mechanizmą, kuris užtikrintų vientisų ir nedalomų transakcijų saugų panaudojimą. Pagrindinė sprendžiama problema yra galimų transakcijos principams prieštaraujančių operacijų (pvz. I/O operacijų) naudojimo apribojimas STM transakcijose. Darbo tikslas yra programinės transakcinės atminties laisvosios monados kūrimas (panaudojant jau esančius STM įrankius Scala programavimo kalboje) ir jos (monados) korektiškumo įrodymas.

Raktiniai žodžiai: Transakcinė atmintis (STM), laisvoji monada, transakcija.

Summary

This research is focused on memory management problem when a few different processes are competing for a resource. Software transactional memory mechanism which uses principles of transactions is used to solve this problem. Mechanism of safe atomic and undividable transactions is examined and developed during this research, which means implementing restriction of possibly non-transactional operations (for example I/O operations) use in STM transactions. The purpose of this research is development of STM free monad (using existing *Scala STM* tools) and proof of such monad correctness.

Key words: Software transactional memory (STM), free monad, transaction.

TURINYS

Įvadas

Viena iš sudėtingiausių informatikos mokslo programavimo disciplinų yra konkurencinis programavimas. Programuojant tokiu būdu yra labai svarbu užtikrinti taisyklingą kelių dėl resursų konkuruojančių gijų veikimą, bei bendrą resursų paskirstymą. Vienas iš resursų, kuris gali labai stipriai paveikti kelių skirtingų procesų veikimą, jeigu bus naudojamas nekorektiškai, yra atmintis [Wei08]. Egzistuoja įvairių būdų, tokių kaip atminties užrakinimas ar transakcinė atmintis, kurie leidžia spręsti šią konkurencinio programavimo problemą. Detaliau šiame darbe bus aptariamas transakcinės atminties mechanizmo pritaikymas.

Transakcinės atminties (STM) mechanizmas remiasi transakcijomis, kurios turi būtinai būti vientisos ir vykdomos kaip viena operacija. Tai reiškia, kad atminties pakeitimai atliekami vienoje nedalomoje veiksmų sekoje ir arba visi jie yra įrašomi arba visi atšaukiami. Kadangi transakcijoje gali vykti keletas veiksmų yra reikalingas mechanizmas, kuris užtikrintų, kad visi šie veiksmai galėtų būti įvykdomi kartu arba atšaukiami be jokių padarinių [Wei08]. Čia iškyla įvairių netransakcinių operacijų, tokių kaip skaitymas ir rašymas, atsiradimo STM transakcijose problema. Tokios operacijos gali bendrai pažeisti transakcijos bazinius principus ir tokiu būdu sukelti pašalinius nepageidaujamus veiksmus, jei transakcija vis dėlto buvo atšaukta. Taigi, tai yra vienas iš iššūkių, kuriuos turi spręsti programiniai karkasai ir įrankiai skirti naudotis transakcinės atminties mechanizmu.

Nesaugių operacijų vykdymo transakcijose problemą gali padėti išspręsti funkciniame programavime naudojamos laisvosios monados. Šios monados leidžia atskirti aprašomo bloko struktūrą nuo jo vykdymo. Naudojantis šiuo mechanizmu galima apriboti nesaugių STM operacijų naudojimą leidžiant tik teisingą transakcinės atminties mechanizmo transakcijų interpretavimą. Verta pastebėti, kad tokia monada turi būti aprašyta taip, kad ji būtų korektiška (atitiktų monadoms keliamas taisykles), nes tik tokiu atveju ji bus tinkama panaudojimui.

Daugeliui funkcinio programavimo kalbų yra sukurtos bibliotekos, kurios įgyvendina transakcinės atminties mechanizmą. Viena iš tokių kalbų yra Scala programavimo kalba ir jos biblioteka *Scala STM*. Ši biblioteka realizuoja transakcinės atminties principus tik dalinai dėl to, kad jos transakcijose galima iškviesti skaitymo ir rašymo ar kitokias nesaugias ir transakcijos atšaukimo atveju neatstatomas operacijas. Taigi, programuojant Scala kalba atsiranda poreikis turėti kitą, STM principus atitinkantį, karkasą (angl. *framework*).

Šio darbo **tikslas** yra sukurti programinės transakcinės atminties laisvąją monadą (panaudojant jau esančius STM įrankius Scala programavimo kalboje) ir įrodyti jos korektiškumą. Šiam tikslui pasiekti nagrinėjami laisvųjų monadų, bei STM principai, aprašomos kylančios problemos, bei funkcinio programavimo Scala kalba ypatumai, kuriant specializuotą kalbą (angl. *domain-specific language*).

Keliami tokie uždaviniai:

- išnagrinėti STM ir laisvųjų monadų principus,
- išspręsti standartinę konkurencinio programavimo užduotį, panaudojant *Scala STM* biblioteką,
- sukurti laisvąją monadą transakcinės atminties principams įgyvendinti,
- aprašyti aukščiau minėtos monados kūrimo principus ir bendras kylančias problemas,
- įrodyti sukurtos monados korektiškumą,
- pademonstruoti laisvosios monados gebėjimą spręsti anksčiau minėtą standartinį uždavinį.

1. Transakcinės atminties apžvalga

1.1. Motyvacija naudoti STM

Viena iš didžiausių problemų kylančių kuriant lygiagrečiai veikiančias ir kelis procesorius vienu metu yra bendrų resursų naudojimas ir paskirstymas. Vienas dažniausiai tokiame programavime naudojamų mechanizmų yra kritinių sekcijų bei rakinimų mechanizmas [ST97]. Nors rakinimų mechanizmas ir padeda lygiagrečiai dalinti resursus, tačiau kuo daugiau yra kritinių sekcijų ir vykdomų resursų rakinimų, tuo yra didesnis amžinojo užrakinimo (angl. *deadlock*) pavojus. Stengiantis sumažinti tokių kritinių sekcijų skaičių, o idealiu atveju ir visiškai jų atsisakyti, kuriami resursų neblokuojantys mechanizmai. [HM93] [Mar05]

Vienas iš tokių resursų neblokuojančių mechanizmų yra transakcinės atminties mechanizmas. Naudojant šį mechanizmą atminties operacijos gali būti atliekamos lygiagrečiai, tačiau tikrinamos būsenos prieš transakciją ir po visų jos operacijų ir atmintis keičiama tik jei būsena yra vis dar ta pati.

Kitas didelis transakcinės atminties mechanizmo privalumas yra tas, kad programuotojui visiškai nereikia rūpintis atminties resurso paskirstymu programuojant. Jam užtenka aprašyti skirtingų gijų darbą ir leisti konkrečias transakcijas, o skirstymu užsiima pats mechanizmas. Taip lygiagretus programavimas tampa daug paprastesnis ir programa išskaidoma į aiškias prognozuojamai veikiančias dalis.

1.2. STM principai

Transakcinės atminties (angl. *STM*) mechanizmas remiasi klasikiniais transakcijų principais. Kaip ir įprastų duomenų bazių taip ir transakcinės atminties transakcijų pagrindinė idėja yra operacijų įvykdymas tam tikrais blokais. Minėtieji operacijų blokai, dar vadinami atominiais blokais, gali būti arba įvykdomi visi iš karto arba atšaukiami taip pat kartu. Šios idėjos pabrėžia du pagrindinius transakcijų principus [Jon07]:

- transakcijos yra atominės ir nedalomos,
- visi veiksmai transakcijose gali būti atstatomi ir paveikti resursai atstatomi į prieš transakciją buvusią būseną.

Pagrindinė transakcinės atminties transakcijų idėja remiasi tuo, kad atmintis nėra rakinama vykdant kiekvieną transakcijos (jos atominio bloko) operaciją, o visos operacijos su atmintimi yra surašomos į transakcijos veiksmų žurnalą. Pasibaigus operacijoms, pagal veiksmų žurnalą yra nustatoma į kokią būseną turi pereiti atmintis ir patikrinama ar tokie pakeitimai yra įmanomi. Jeigu pakeisti yra galima, tai atminties pakeitimai yra įvykdomi (angl. *commit*), o jeigu ne (pvz. kita transakcija yra jau pakeitusi atmintį iš pradinės būsenos į kitą), visi pakeitimai yra atšaukiami (angl. *rollback*). Toliau transakcijos blokas gali būti vykdomas iš naujo. Tokiu būdu yra

išvengiama ilgo atminties užrakinimo ir sumažinama amžinojo atminties užrakinimo (angl. *deadlock*) problemos atsiradimo rizika. [Wei08]

1.3. STM įgyvendinimas

Pagrindinis įrankis kurį reikia įgyvendinti transakcinės atminties mechanizme yra transakcija. Reikalinga sukurti tokią abstrakciją, kuri leistų sudėti atminties skaitymo, rašymo ir panašias operacijas į vieną bloką ir sugebėtų šių operacijų vykdomus veiksmus surašyti į transakcijų veiksmų žurnalą, o ne vykdyti iš karto. Taip bus užtikrintas transakcijos atomiškumas ir atminties ląstelės nebus keičiamos vykdant kiekvieną operaciją. Be to sukursime izoliuotą transakciją, nes jokia kita gija nežinos apie tai, kas yra vykdoma šioje transakcijoje ir galės matyti tik ankstesnę atminties versiją vykstant transakcijai, bei naują jai įvykus.

Taip pat reikia sukurti pagrindines operacijas skirtas dirbti su transakcijomis. Galima išskirti keturias pagrindines tokias transakcijos operacijas [Sco06]:

- pradėti transakciją (angl. *start*),
- išsaugoti transakcijos pakeitimus (angl. *commit*),
- atšaukti transakcijos pakeitimus (angl. *rollback* arba *abort*),
- pakartoti transakciją (angl. *retry*), kuri yra atšaukimo ir transakcijos pradėjimo operacijų seka.

Šios operacijos remiasi darbu su jau anksčiau minėtu transakcijos veiksmų žurnalu. Jos paima įrašus iš žurnalo priklausančiu įvykdytai transakcijai, patikrina ar gali atlikti pakeitimus ir priklausomai nuo rezultato juos atlieka arba atšaukia. Operacija transakcijos pakartojimui yra analogiška transakcijos rezultatų įrašymo atšaukimo operacijai, tačiau po jos įvykdymo bandoma pakartoti transakciją su jau naujomis atminties ląstelių reikšmėmis.

Kaip buvo minėta anksčiau, STM transakcijos turi būti nedalomos ir visi su atmintimi atliekami veiksmai būtų grįžtami. Įgyvendinant šiuos reikalavimus, turi būti užtikrinta, kad operacijos esančios transakcijos atominiuose blokuose būtų atšaukiamos ir visi veiksmai, kuriuos jos atlieka gali būti atstatomi. Todėl kuriant transakcinės atminties mechanizmą taikančią biblioteką ar specializuotąją kalbą yra ypač svarbu neleisti tokių operacijų kaip skaitymas ir rašymas į ekraną (pvz. išvedus informaciją į ekraną šio veiksmo atliekant transakcijos atšaukimą nebegalime atšaukti). Šio tipo operacijos atlieka veiksmus su išoriniu pasauliu ir iš principo nėra susijusios su atminties transakcijomis.

1.4. STM ir funkcinis programavimas

Tyrose (angl. *pure*) funkcinio programavimo kalbose (tokiose kaip *Haskell*) kuo daugiau naudojamų funkcijų turi būti tyrios. Tyros funkcijos – tai tokios funkcijos, kurios gavę tokį patį argumentą kiekvieną kartą grąžins tokią pat reikšmę ir vykdymo metu niekaip nepriklausys nuo išorinio poveikio. Kitaip sakant tokios funkcijos veikia juodosios dėžės principu ir niekaip

nepaveikia bei nėra veikiamos išorinio pasaulio vykdymo metu. Taip pat tokiose kalbose kaip *Haskell* yra griežta tipų sistema, kurią panaudojant apibrėžiamos STM operacijos ir kitų operacijų vykdymas STM blokuose yra apribojamas.

Netyrose (angl. *impure*) funkcinio programavimo kalbose (pvz. *Scala*) *STM* įgyvendinti yra sunkiau ir kyla daugiau iššūkių, nes tokių kalbų funkcijos yra lankstesnės ir dėl to gali paveikti išorinį pasaulį (pvz. rašymas ir skaitymas iš terminalo, atsitiktinių reikšmių generavimas), o to transakcinės atminties transakcijose ir bandoma išvengti. Tokiose kalbose yra reikalingas papildomas operacijų galimų transakcijose apribojimas.

Transakcinės atminties įgyvendinimui gali būti naudojamas funkciniame programavime taikomas monadų mechanizmas. Monados leidžia nustatyti tam tikrą operacijų seką ir programą vykdyti žingsniais. Tokiu būdu galima užtikrinti transakcijų izoliavimą nuo išorinio pasaulio bei jų atomiškumą. Be to monados yra griežtai tipizuotos struktūros ir jose galima apriboti tam tikro tipo operacijas. Šiuo būdu gali būti sprendžiamas nesaugių operacijų apribojimas.

2. Monados

2.1. Monadų apibrėžimas

Monada (angl. *monad*) - tai yra būdas struktūrizuoti operacijų seką ar sekas į vieną bloką. Monadų mechanizmas leidžia programuotojui dirbti operuojant sekų blokais, o ne pavienėmis funkcijomis ir be to leidžia aprašyti kaip tos funkcijos yra jungiamos. Monadai aprašyti reikalingos tokios funkcijos [Cla99]:

- grąžinimo (angl. *return*) ,
- apjungimo operatorius (angl. *bind* arba >>=).

Grąžinimo funkcija aprašo kaip monados argumentas yra konvertuojamas į monados kintamąjį, o apjungimo funkcija nustato, kas įvyks, kai viena funkcija bus jungiama su kita naudojant šią monadą. Pavyzdžiui, paimkime vieną paprasčiausių *Haskell* kalbos monadų *Maybe* ir jos aprašą:

```
return :: a -> Maybe a
return x  = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) m >>= g = case m of
                Nothing -> Nothing
                Just x  -> g x
```

1 pav. Maybe monada

Čia (1 pav.) matome, kad *return* funkcija gavusi argumentą x visada grąžins apvilktą (angl. *wrapped*) *x* reikšmę *Just x*. Apjungimo operatorius šioje vietoje gavęs argumentą su *Maybe* monada ir funkcija *a* → *Maybe b* gaus arba reikšmę *Nothing* arba funkcijos *a* → *Maybe b* rezultatą su reikšme *x*. Tokiu būdu gauname monadą, su kuria jungiant funkcijas padavę kaip argumentą *Nothing* rezultatą taip pat gausime *Nothing* nepaisant to, kokias funkcijas jungsime.

2.2. Monadų taisyklės

Monadą galime laikyti korektiška tada, kai jos operacijos tenkina šias tris taisykles [Wad95]:

- Kairiojo tapatumo: $Monad(a).flatMap(f) \equiv f(a)$ ši taisyklė reiškia, kad gautą rezultatą iš grąžinimo funkcijos sujungę su funkcija f gausime tą patį, ką gautume suskaičiavę funkciją f su grąžinimo funkcijos argumentu,
- Dešiniojo tapatumo: $m.flatMap(Monad) \equiv m$ ši taisyklė reiškia kad bet kokią monados reikšmę sujungę su gražinimo funkcijos reikšmę gausime tą pačią reikšmę,
- Asociatyvumo: $m.flatMap(g).flatMap(f) \equiv m.flatMap(x => f(x).flatMap(g))$ ši taisyklė reiškia, kad jungdami tris atskiras monadų reikšmes, galime jungti jas bet kokiu eiliškumu ir gausime tą patį rezultatą.

Pastabos: *Monad* čia atitinka grąžinimo funkciją, o *flatMap* – apjungimo.

2.3. Monadų panaudojimas STM

Įgyvendinant transakcijų mechanizmą galime pasitelkti monadų teikiamus privalumus. Aprašę monadą taip, kad joje nebūtų leidžiamos transakcijoje nesaugios išorinį pasaulį veikiančios operacijos, bei kombinuodami tokias monadas galime sukurti tam tikrą transakcijos abstrakciją. Sukūrus tokią monadą turėsime tam tikras nedalomas veiksmų sekas, tačiau tam, kad užtikrintume tų veiksmų atšaukiamumą ir visišką izoliuotumą nuo išorinio pasaulio vien paprasto monadų mechanizmo neužtenka.

Vien naudodami paprastas monadas netyrose funkcinio programavimo kalbose mes visiškai neišvengsime nesaugių operacijų kvietimo. Tokiose kalbose reikia ne tik apriboti vykdomų funkcijų seką, bet reikia ir aprašyti kaip tie veiksmai bus interpretuojami t. y. kaip vykdyti veiksmų seką neleidžiant įvykdyti nesaugių operacijų.

2.4. Laisvosios monados

Anksčiau minėtos netyrų funkcinių programavimo kalbų monadų nesaugumo problemos sprendimas galėtų būti laisvosios monados (angl. *free monad*). Laisvosios monados tai tokios monados, kuriomis aprašoma ne tik pati monada, tačiau ir būdas kaip bus interpretuojama aprašytų veiksmų seka [Bja12]. Tokiu būdu yra atskiriama bloko struktūra nuo to, kaip tas blokas bus vykdomas. Kitaip tariant, programuotojas gali aprašyti tam tikrą veiksmų seką, o jos interpretavimą ir įvykdymą atlikti vėliau atskirai nuo aprašo.

Šiuos principus gana paprasta pritaikyti realizuojant transakcinės atminties mechanizmą. Kurdami laisvąsias monadas, galime aprašyti tam tikras transakcines operacijas, kurias naudosime ir jų vykdymą apibrėžti taip, kad nesaugios skaitymo ir rašymo operacijos būtų apribojamos. Atskyrus užrašymą nuo interpretavimo žymiai sumažiname tokių operacijų įvykdymo tikimybę.

Pavyzdžiui, galime aprašyti tokį atominį bloką, kuris sudėtų du skaičius ir įrašytų juos į atmintį ir vėliau jis būtų įvykdomas. Naudojant laisvąsias monadas, tai atrodytų taip:

- pirmiausia aprašome funktorių (objektą arba klasę, kuri gali reikšmių aibei pritaikyti tam tikrą funkciją),
- funktorių panaudojame sukurdami laisvąją monadą,
- aprašome funkciją, kuri atitiks mūsų interpretavimą t. y. suras kokia operacija yra vykdoma ir vykdys aprašytą operaciją,
- panaudodami sukurtą laisvąją monadą, sukuriame funkciją, kurioje užrašome skaičių sudėjimo ir įrašymo į atmintį veiksmus,
- panaudodami anksčiau aprašytą vykdymo funkciją galime leisti kodą.

Kaip ir paprasta monada, laisvoji monada turi atitikti monadų taisykles. Tik tokią laisvąją monadą galime laikyti korektiška ir panaudojama transakcinės atminties uždavinių kūrimui.

3. Uždavinio su Scala STM sprendimas

3.1. Sprendžiamo uždavinio aprašymas

Vienas iš galimo konkurencinio programavimo su bendra atmintimi panaudojimo pavyzdžių yra vienos juostos tilto problema (angl. *Single-lane bridge problem*) [MK99]. Šio uždavinio aprašas:

Tiltas per upę turi tik tai vieną juostą ir juo vienu metu gali važiuoti tik trys automobiliai. Automobiliai gali privažiuoti iš tilto ir iš vienos ir iš kitos pusės, tačiau ant tilto vienu metu gali važiuoti tik į vieną pusę. Kol vyksta eismas viena kryptimi, kitos krypties automobiliai stovi prie tilto ir laukia kol jis bus visiškai tuščias. Jei automobiliu važiuoja trys tos pačios krypties automobiliai, ketvirtasis ir kiti taip pat laukia prie tilto, kol važiuos du arba mažiau automobilių.

Galimas toks šio uždavinio sprendimo būdas – kiekvieną kartą, kai prie tilto atvyksta naujas automobilis patikrinti, kiek automobilių yra ant tilto:

- jei jau yra trys, pastatyti automobilį į laukiančiųjų eilę,
- jei automobilių ant tilto yra, tačiau mažiau negu trys, reikia patikrinti jų kryptį ir jei ji sutampa su naujo automobilio kryptimi, leisti automobiliui važiuoti, kitu atveju pastatyti laukti,
- jei automobilių ant tilto nėra, reikia leisti automobiliui važiuoti.

Sprendžiant tokiu būdu mes turėsime tik dvi įmanomas automobilio būsenas: laukiantis arba važiuojantis. Taigi, šį uždavinį labai patogu spręsti naudojant transakcijas, kadangi laukimas atitiks transakcijos atšaukimą ir kartojimą, o važiavimas jau įvykdytą transakciją.

3.2. Sprendimas su Scala STM

Scala funkcinio programavimo kalboje yra įgyvendinta transakcinės atminties principus naudojanti *Scala STM* biblioteka. Šiame skyrelyje pademonstruosime vienos juostos tilto problemos sprendimą naudojant šią *Scala* biblioteką (pilnas kodas pateikiamas 1 priede).

```
case class Car(direction: Direction)

sealed trait Direction
object Right extends Direction
object Left extends Direction
```

2 pav. Klasės „*Car*“ aprašymas *Scala* kalboje

Pirmiausia aprašome automobilio abstrakciją (2 pav.), kas mums leis turėti tam tikrą pagrindinio uždavinyje operuojamo objekto struktūrą ir toliau aprašinėti veiksmus atliekamus su juo. Čia pateikiamas paprasčiausias automobilio klasės įgyvendinimas, kai automobilis turi tik savo važiavimo per tiltą kryptį.

```
private def arriveOnBridge(car: Car) {
  atomic { implicit Txn =>
    if ((carsOnBridge.size == 3) ||
      (carsOnBridge.size != 0 && !(car.direction == bridgeDirection))) {
      retry;
    }

    if (carsOnBridge.size == 0) {
      bridgeDirection() = Option[Direction](car.direction);
    }

    carsOnBridge += car;
  }
}
```

3 pav. Blokas *arriveOnBridge*

Funkcijos čia aprašomos kaip atominiai blokai (3 pav.) ir yra automobilių keliavimo per vienos juostos tiltą abstrakcijos.

```
private def exitBridge(car: Car) {
    atomic { implicit Txn =>
      carsOnBridge.-=(car);
    }
}
```

4 pav. Blokas *exitBridge*

Aprašomi du atominiai blokai pagrindiniams veiksmams atlikti:

- blokas *arriveOnBridge* (3 pav.) funkcijoje atlieka veiksmus, kurie yra vykdomi, kai naujas automobilis pasiekia tiltą,
- bloke *exitBridge* (4 pav.) yra sumažinamas automobilių esančių ant tilto skaičius, kai automobilis palieka tiltą.

```
def crossBridge(car: Car) {
    arriveOnBridge(car);
    Thread.sleep(1000);
    exitBridge(car);
}
```

5 pav. Funkcija *crossBridge*

Turėdami šias dvi funkcijas, galime jas sukomponuoti taip, kad susimuliuotume automobilio judėjimą tiltu. Tokia simuliacija pavaizduota funkcijoje *crossBridge*, kur automobilis privažiuoja prie tilto ir bando ant jo patekti, važiuoja tiltu ir galų gale jį pervažiuoja. Ši funkcija ir yra kviečiama keliose gijose, kurios konkuruoja dėl atminties resurso.

Pastebėkime, kad visi veiksmai, kur keičiama tilto kryptis, ar automobilių skaičius ant tilto, vyksta transakcijose. Tokiu būdu yra užtikrinama, kad pasikeitimai atmintyje būtų įvykdomi tik tada, kai jie validūs ir nebūtų paveikiamos kitos gijos. Be to transakcinė atmintis čia padeda išvengti ilgo atminties rakinimo ir paslepia šį procesą nuo programos kūrėjo, kadangi tuo rūpinasi pats mechanizmas.

4. Saugi Scala transakcinės atminties specilizuotoji kalba

Scala programavimo kalboje esantis *Scala STM* karkasas ne visiškai atitinka transakcinės atminties mechanizmo principus ir leidžia nesaugias skaitymo rašymo operacijas transakcijose. Iš to iškyla poreikis sukurti tokį įrankį *Scala* kalbai, kuris apribotų nesaugių operacijų kvietimą ir atitiktų visus transakcinės atminties mechanizmui keliamus reikalavimus. Vienas iš tokios problemos sprendimų galėtų būti transakcinės atminties specializuotos kalbos (angl. *domain-specific language*) kūrimas.

4.1. Pagrindinės savybės

Įgyvendinant transakcinės atminties specializuotąją kalbą *Scala* programavimo kalbos apimtyje, reikalinga apibrėžti keletą pagrindinių architektūros principų. Galima išskirti keletą pagrindinių tokios kalbos dalių, kurias reikia suprojektuoti ir sukurti.

Pirmiausia, transakcijos atomiškumo ir vientisumo užtikrinimui yra reikalinga turėti galimybę kodo dalis sugrupuoti į tam tikras grupes, kurios ir atitiks veiksmų su atmintimi transakcijas. Tokioms operacijų sekoms galima panaudoti jau mūsų anksčiau minėtas laisvąsias monadas. Kuriant tokią monadą reikia apibrėžti operacijas, kurias bus galima kviesti transakcijos viduje. Įgyvendinant darbe aprašomą transakcinės atminties specializuotąją kalbą buvo parinktos pagrindinės veiksmų su atmintimi, bei kelios transakcijoms specifinės operacijos. Pagrindinės pasirinktos operacijos [HM93]:

- atminties skaitymas transakcijoje (angl. *transactional load*),
- atminties rašymas transakcijoje (angl. *store-transactional*),
- išsaugoti transakcijos pakeitimus (angl. *commit*),
- atšaukti transakcijos pakeitimus (angl. *rollback*),
- pakartoti transakciją (angl. *retry*).

Kitas svarbus aspektas yra transakcijos įvykdymo mechanizmo sukūrimas. Turėdami aprašyta monadą su operacijomis, galime sudėlioti reikiamus transakcinius blokus, tačiau šių blokų vykdymui yra reikalingas jų interpretavimo mechanizmas. Interpretavimo mechanizmo ir blokų užrašymo atskyrimui buvo panaudotos laisvosios monados. Kaip ir anksčiau minėta šių monadų esminis skirtumas yra būtent tai, kad aprašytas operacijas galima atskirti nuo jų interpretavimo. Taigi, aprašyta monada buvo kuriama kaip laisvoji monada.

Taip pat labai svarbu yra sukurti būdą, kaip aprašyti kintamųjų tipus darbui su atmintimi. Apibrėžtų kintamųjų reikšmė gali būti keičiama tik transakcijos viduje ir jų vidinė struktūra turi užtikrinti, kad lygiagrečiai keičiant jų reikšmę būtų atsižvelgta į prieš tai buvusią ir pagal tai nuspręsta ar išsaugoti transakcijos veiksmus ar visus juos atšaukti.

4.2. Įgyvendinimas

Šiame skyriuje trumpai aprašomi saugios transakcinės atminties specializuotos kalbos įgyvendinimo etapai. Pilnas kodas įgyvendinantis šią kalbą pateikiamas 2 priede.

4.2.1. Transakcinės atminties kintamieji

Prieš aprašant operacijas yra reikalinga apsibrėžti kintamųjų tipus, kuriuos naudosime šiose operacijose. Kuriant saugią Scala transakcinės atminties specializuotąją kalbą buvo pasinaudota jau esančiais *Scala STM* karkaso tipais, kurie naudojami transakcijos kintamiesiems apibrėžti. Kuriamai kalbai buvo pasirinktos dvi duomenų struktūros, kurios leis apibrėžtų didžiąją dalį visų reikalingų transakcinės atminties kintamųjų:

- rodyklė į kintamąjį (angl. *reference*),
- aibė (angl. *set*).

```
final class SafeVal[A](a: A) {
    private[ScalaFreeSTM] val tRef: Ref[A] = Ref(a)
}
```

6 pav. Rodyklės į kintamąjį apibrėžimas

Rodyklė į kintamąjį (6 pav.) apibrėžiama taip: jos viduje saugoma *Scala STM* karkaso rodyklės į transakcinį kintamąjį reikšmė (*Ref*). Ši reikšmė yra privati visoje specializuotos kalbos klasėje, kas reiškia, kad išorinėse funkcijose naudojančiose šį kintamąjį nebus galima tiesiogiai pasiekti šios reikšmės ir šis kintamasis bus pilnai pasiekiamas tik transakcijos kontekste. Aprašant tokio tipo kintamąjį reikalinga tiesiog nurodyti kokio tipo ir kokia reikšmė saugoma šiame kintamajame.

```
final class SafeSet[A]() {
    private[ScalaFreeSTM] val tSet: TSet[A] = TSet()

    private[ScalaFreeSTM] def add(value: A)(implicit txn: InTxn): Unit = {
      tSet.+=(value)
    }

    private[ScalaFreeSTM] def remove(value: A)(implicit txn: InTxn): Unit = {
      tSet.-=(value)
    }
}
```

7 pav. Aibės aprašas

Apibrėžiant aibės tipo kintamąjį (7 pav.) analogiškai „užvelkama“ (angl. *wrap*) *Scala STM* transakcinės aibės (*TSet*) tipo reikšmė. Ši reikšmė taip pat yra privati specializuotoje kalboje siekiant ją paslėpti nuo išorinio naudotojo ir priverčiant jį naudotis transakcijos operacijomis norint operuoti šia reikšme. Sukūrus naują aibės tipo kintamąjį minėta reikšmė taip pat yra inicializuojama. Norint palengvinti transakcinių operacijų atliekamų su šia struktūra aprašymą yra sukuriamos pridėjimo ir išėmimo iš aibės funkcijos, kurios iš tikrųjų kviečia pridėjimo ir išėmimo operacijas *Scala STM* transakcinės aibės tipo privačiam kintamajam. Šios operacijos taip pat yra

privačios specializuotos kalbos lygmenyje, nes jos bus naudojamos tik aprašant operacijas ir negali būti kviečiamos ne transakcijos viduje.

4.2.2. Operacijų apibrėžimas

Kitas žingsnis yra transakcinių operacijų apibrėžimas. Apžvelgus siūlomas (4.1. poskyris) pagrindines atminties operacijas ir turimus kintamųjų tipus buvo nuspręsta sukurti tokias operacijas:

- *AddTSet* – operacija skirta pridėti reikšmę į aibės kintamąjį,
- *RemoveTSet* – operacija skirta pašalinti reikšmę iš aibės kintamojo,
- *CountTSet* – operacija skirta sužinoti aibėje esančių reikšmių kiekį,
- *ReadTVal* – operacija skirta gauti reikšmę iš rodyklės į kintamąjį tipo,
- *WriteTVal* – operacija skirta įrašyti reikšmę į rodyklės į kintamąjį tipą,
- *Retry* – operacija skirta pakartoti transakcijos vykdymą.
- *NoOp* – operacija, kuri nieko nedaro ir yra skirta *if else* blokams

Verta pastebėti, kad kitos dvi įvardintos kaip būtinos transakcinės operacijos – transakcijos veiksmų išsaugojimas, bei transakcijos veiksmų atšaukimas – čia nėra aprašomos, nes jomis pasirūpins *Scala STM* bibliotekoje esantis *atomic* blokas, kuriame bus vykdomos transakcijos. Taip pat skirsis *retry* operacijos aprašymas, nes ši operacija bus paslepiama ir vykdoma tada, kai atitiks *if* sąlygą *for-yield* bloko viduje.

```
sealed trait STMOp[A]
private object STMOp {
  case class AddTSet[A](value: A, set: SafeSet[A]) extends STMOp[Unit]
  case class RemoveTSet[A](value: A, set: SafeSet[A]) extends STMOp[Unit]
  case class CountTSet[A](set: SafeSet[A]) extends STMOp[Int]

  case class ReadTVal[A](tVal: SafeVal[A]) extends STMOp[A]
  case class WriteTVal[A](value: A, tVal: SafeVal[A]) extends STMOp[Unit]

  case object Retry extends STMOp[Unit]

  case object NoOp extends STMOp[Unit]
}
```

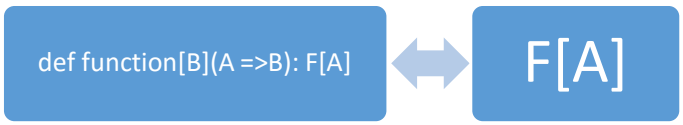
8 pav. Transakcinių operacijų apibrėžimas.

Kaip matome iš apibrėžimo (8 pav.) visos transakcinės operacijos paveldi iš specialiai tam sukurto *STMOp* tipo. Toks kelias buvo pasirinktas dėl to, kad visoms operacijoms būtų vienas tas pats abstraktus tipas, kas vėliau leistų jas apibendrinti ir aprašyti, kad tik tokio tipo operacijos gali būti kviečiamos transakcijos viduje. Apibrėžti operacijoms yra naudojamas tipinis Scala sąlyginių klasių (angl. *case class*) ir sąlyginių objektų (angl. *case object*) mechanizmas, kurio pagalba vėliau interpretuodami aprašytas operacijas ir naudodami šablonų sutapimo mechanizmą (angl. *pattern matching*) galėsime lengvai atrinkti, kuri operacija yra kviečiama.

Operacijoms reikalingus parametrus aprašome kaip sąlyginės klasės parametrus, o grąžinamas reikšmes kaip parametrizuotą tipą (angl. *generics*) *STMOp* klasei.

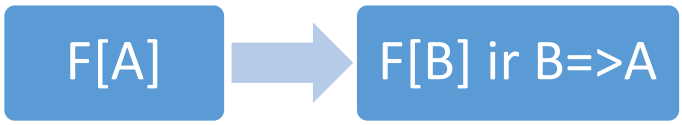
4.2.3. Laisvosios monados sukūrimas naudojant CoYoneda

Aprašius *STMOp* operacijas, kurios bus naudojamos laisvojoje monadoje, galima pereiti prie pačios monados sukūrimo. Laisvosios monados aprašymui skirtas *scalaz* funkcinio programavimo karkaso *Free* tipas reikalauja aprašyti funktorių iš kurio ir bus konstruojama monada. Kadangi operacijos apibrėžtos tik kaip paprastos sąlyginės klasės objekte funktoriaus neturime ir reikalinga pasinaudoti tam tikrais kategorijų teorijoje esančiais ir *scalaz* karkase įgyvendintais įrankiais. Vienas iš tokių įrankių, kuris šiuo atveju ir naudojamas konstruojant funktorių, yra *CoYoneda*.



9 pav. *Yoneda* lema

Sąvoka *CoYoneda* yra kilusi iš kategorijų teorijoje esančios *Yoneda* lemos. (Daugiau apie *Yoneda* lemą galima pasiskaityti nurodytame šaltinyje [BW90]) Paprastai tariant *Yoneda* lema sako, kad jeigu turime tokią funkciją, kuri kaip parametrą gauna funkciją $A \rightarrow B$ ir grąžina *F[B]* (F čia funktorius), tai galima pasakyti kokia bus *F[A]* reikšmė. Ši sąlyga galioja ir atvirkščiai, t.y. jeigu turime reikšmę su tipu *F[A]* bet kokiam funktoriui *F* ir bet kokiam tipui *A*, tai žinome ir anksčiau minėtą funkciją (9 pav.).



10 pav. *CoYoneda* lema

Pamodifikavus *Yoneda* lemą galime gauti specializuotos kalbos kūrime naudojamą *CoYoneda* lemą. Pagrindinis tokios lemos kuriamoje kalboje naudojamas principas yra tas, kad jeigu turime *F[A]* tipo reikšmę bet kokiems *F* ir *A*, tai mes visada galime ją sustruktūrizuoti taip, kad gautume reikšmę *F[B]* ir funkciją, kurios tipas $B \rightarrow A$ (10 pav.). Svarbiausia specializuotos kalbos kūrime naudojama šios lemos savybė yra tai, kad minėtas *F* nebūtinai turi būti funktorius, o pati *CoYoneda* jau yra funktorius pagal savo apibrėžimą. Taigi, tokiu būdu sukuriame laisvosios monados aprašymui reikalingą funktorių.

```
type STMCoyoneda[A] = Coyoneda[STMOp, A]
type FreeSTM[A] = Free[STMCoyoneda, A]

def addTSet[A](value: A, set: SafeSet[A]): FreeSTM[Unit] = liftFC[STMOp,
Unit](AddTSet(value, set))
def removeTSet[A](value: A, set: SafeSet[A]): FreeSTM[Unit] = liftFC[STMOp,
Unit](RemoveTSet(value, set))
def countTSet[A](set: SafeSet[A]): FreeSTM[Int] = liftFC[STMOp,
Int](CountTSet(set))

def readTVal[A](tVal: SafeVal[A]): FreeSTM[A] = liftFC[STMOp, A](ReadTVal(tVal))
def writeTVal[A](value: A, tVal: SafeVal[A]): FreeSTM[Unit] = liftFC[STMOp,
Unit](WriteTVal(value, tVal))

def noOp[A]: FreeSTM[Unit] = liftFC[STMOp, Unit](NoOp)
```

11 pav. Specializuotos kalbos funkcijų apibrėžimas

Panaudojus *CoYoneda* ir aprašius laisvąją monadą vėlesnio specializuotos kalbos panaudojimo paprastumui yra sukuriamos funkcijos, kurias bus galima kviesti iš kalbos klasės tiesiogiai, nesigilinant į tai, kaip jos buvo aprašytos ar realizuotos (11 pav.). Tam panaudota *scalaz* karkaso *Free* klasės (naudotos aprašant laisvąją monadą) metodas *liftFC*, kuris gavęs kaip parametrą anksčiau aprašyta sąlyginę klasę, ją uždengia (angl. *wrap*) ir grąžina laisvosios monados tipo reikšmę. Tokiu būdu, dėl lengvesnio panaudojimo, dar kartą aprašomos visos anksčiau minėtos monados transakcinės operacijos.

4.2.4. Interpretavimas

Turint laisvosios monados aprašą, bei apibrėžtas reikalingas operacijas, paskutinis likęs specializuotos kalbos kūrimo žingsnis yra operacijų interpretatoriaus aprašymas.

```
private type InTxnInterpreter[A] = InTxn => A
  private val interpretOp: STMOp ~> InTxnInterpreter =
    new (STMOp ~> InTxnInterpreter) {
      def apply[A](op: STMOp[A]): InTxnInterpreter[A] = {
        op match {
          case AddTSet(value, set)     => { implicit txn => set.add(value) }
          case RemoveTSet(value, set) => { implicit txn => set.remove(value) }
          case CountTSet(set)          => { implicit txn => set.tSet.size }

          case ReadTVal(tVal)          => { implicit txn => tVal.tRef() }
          case WriteTVal(value, tVal) => { implicit txn => tVal.tRef() = value }

          case Retry                   => { implicit txn =>
          scala.concurrent.stm.retry }
          case NoOp                    => { implicit txn => }
        }
      }
    }
```

12 pav. Transakcinių operacijų interpretavimas

Kaip matome iš interpretatoriaus aprašymo (12 pav.), jis yra gana paprastas. Visos STMOp operacijos yra transformuojamos (~> atlieka transformaciją) į *InTxnInterpeter* tipą. Naudojamas Scala kalboje esantis šablonų atpažinimo (angl. *pattern matching*) mechanizmas ir taip surandama kokią operaciją reikia interpretuoti. Kiekviena iš interpretuojamų operacijų yra transakcinė, todėl

jai reikalingas *Txn* argumentas, kuris *Scala STM* bibliotekoje ir reiškia, kad operacija bus vykdoma transakcijos viduje.

```
def runTransaction[A](trans: FreeSTM[A]): A =
   atomic {
     runFC[STMOp, InTxnInterpreter, A](trans)(interpretOp)
   }
```

13 pav. Transakcijos paleidimo funkcija.

Aprašius interpretatorių, apibrėžiama ir transakcijos paleidimo funkcija (13 pav.). Pagrindinės šios funkcijos ypatybės yra tokios:

- transakcijos paleidimas vyksta *atomic* bloke,
- laisvosios monados sukombinuotos operacijos atitinka vykdomą transakciją ir yra perduodamos kaip parametras šiai funkcijai,
- naudojama *Scala STM* funkcinio programavimo karkaso *Free* klasės *runFC* funkcija, kuri priima laisvąją monada ir interpretatorių kaip parametrus ir įvykdo aprašytus veiksmus,
- funkcija grąžina reikšmę, kuri yra tokia pat kaip ir *Scala STM* bibliotekoje *atomic* bloko grąžinama reikšmė.

4.2.5. Papildomų funkcijų aprašymas

Aprašyti vien tik transakcines operacijas neužtenka. Transakcijos bloke yra naudinga turėti sąlygų aprašymo ir tikrinimo galimybę, kuri leistų kurti lankstesnes programas. Šiam tikslui apibrėžiamos kelios papildomos kuriamos specializuotos kalbos funkcijos:

- aprašomas *filter* metodas sukurtai monadai, kad būtų galima naudoti *if* sakinius *for-yield* viduje,
- naudojant aprašytą *NoOp* operaciją galima naudoti *if else* konstrukciją transakcijos bloke, jeigu norima vykdyti tik tam tikrą operaciją be *else* dalies.

```
implicit class FreeSTMOps[A](freeSTM: FreeSTM[A]) {
    def filter(f: Unit => Boolean): FreeSTM[Unit] = {
        f() ? liftFC[STMOp, Unit](Retry) | noOp
    }
}
```

14 pav. Funkcijos *filter* aprašymas

Aprašoma funkcija *filter* (14 pav.) paleis transakcijos pakartojimo funkciją, jeigu pateikta *if* sąlyga bus teisinga. Tokiu būdu mes paslepiame *retry* operaciją ir padarome transakcinius blokus sintaksiškai gražesnius ir suprantamesnius.

```
_ <- if(a) op else noOp
```

15 pav. Bloko *if else* panaudojimas

Sukurta *NoOp* operacija leidžia aprašyti tokius *if else* konstrukcinius blokus (15 pav.), kurie atitiktų natūralią *for-yield* bloko sintaksę ir leistų aprašyti sąlygą be *else* dalies. Šiame pavyzdyje

(15 pav.) *op* atitinka bet kokią transakcinę operaciją, kuri bus įvykdyta, jeigu sąlyga *a* yra teisinga. Aprašius tokias konstrukcijas sukurtoje specializuotoje kalboje įgyvendintas pilnas sąlygų apdorojimo mechanizmas.

4.3. Sukurtos monados korektiškumo įrodymas

Kuriant saugią Scala programavimo kalbos transakcinės atminties specializuotąją kalbą, buvo sukurta laisvoji monada. Sukūrus monadą reikia įrodyti jos korektiškumą. Monada bus korektiška, jeigu atitiks tris anksčiau minėtas monadų taisykles. Šiame skyrelyje visas šias tris taisykles patikrinsime dviem būdais: samprotavimo, bei leidžiant Scala programavimo kalba parašytus testus.

4.3.1. Atitikimas kairiojo tapatumo taisyklei

Kairiojo tapatumo taisyklė užtikrina kad jungiant monadas nebus keičiamas monados vidus. Turint sukurtą *FreeSTM* monadą ir kviečiant jungimo metodą (Scala atveju tai yra *flatMap*) su tam tikra funkcija, jos viduje tiesiog bus iškviesta funkcija su monadoje esančia reikšme ir grąžintas rezultatas, kuris bus analogiškas rezultatui, jei kviestume tik funkciją su reikšme esančia monadoje. Tuo nesunku įsitikinti jeigu panagrinėsime mūsų uždengtas (angl. *wrapped*) operacijas. Jeigu mes kviesime šią operaciją monadoje su tam tikra reikšme pagal interpretavimą matome, kad bus tiesiog iškviečiama atitinkama funkcija ir neatliekama daugiau jokių pašalinių veiksmų.

```
1) Monad[FreeSTM].point(ref) flatMap operation
2) operation(ref)
```

16 pav. Monados atitikimas kairiojo tapatumo taisyklei

Nurodytos eilutės (16 pav.) grąžina analogišką rezultatą (kitaip tariant 1 ekvivalentus 2). Šiame pavyzdyje *ref* yra transakcinės atminties kintamasis, kuris yra paduodamas funkcijai. Metodas *flatMap* priima vieną iš transakcinės atminties funkcijų *operation* ir iškviečia ją reikšmei *ref* taip atlikdamas analogišką funkcijos tiesioginiam kvietimui su parametru *ref* veiksmą.

Pastaba. Funkcija *point* čia atitinka anksčiau aprašytą monados grąžinimo funkciją.

4.3.2. Atitikimas dešiniojo tapatumo taisyklei

Dešiniojo tapatumo taisyklė teigia, kad prijungę prie monados kitą monadą, kuri yra tuščia t. y. jos viduje nėra jokių aprašytų operacijų, turime gauti pirmąją monadą.

```
1) val a = operation
2) val monad = for {
     x <- operation
   } yield(x)
```

17 pav. Monados atitikimas dešiniojo tapatumo taisyklei

Nurodyti du monadų aprašymai (17 pav.) yra analogiški. Pirmame aprašoma bet kokia transakcinė *operation* operacija, kurios reikšmė tiesiog įrašoma į *a* kintamąjį. Antrajame iš pradžių kviečiama ta pati operacija, kaip ir pirmajame, o paskui kviečiama monados grąžinimo funkcija

reikšmei *x* (tai atitinka *yield(x)* konstrukcija). Kadangi grąžinimo funkcija neatitinka jokios operacijos (kitaip sakant turi tuščią veiksmų sąrašą), ji jokio papildomo poveikio visam blokui nepadarys ir dėl to abu blokai bus analogiški ir grąžins analogišką rezultatą (kitaip tariant 1 ekvivalentus 2).

4.3.3. Atitikimas asociatyvumo taisyklei

Asociatyvumo taisyklė užtikrina, kad monadų sujungimo būdas neįtakoja galutinio rezultato. *FreeSTM* monada šią taisyklę taip pat užtikrina, nes jungdami atskirus transakcijų blokus po du ir paskui gautus transakcinius blokus sujungus gausime tą patį rezultatą.

```
val a12 = for {
  _ <- a1
  _ <- a2
} yield()

val b12 = for {
  _ <- b1
  _ <- b2
} yield()
```

```
val ab = for {
  _ <- a12
  _ <- b12
} yield()

val a12b12 = for {
  _ <- a1
  _ <- a2
  _ <- b1
  _ <- b2
} yield()
```

18 pav. Monados atitikimas asociatyvumo taisyklei

Kaip matome iš aprašymo (18 pav.) *a1, a2* ir *b1, b2* (šiuo atveju tai STM transakcijos) yra sujungiamos atskirai ir vėliau sujungiamos į vieną bloką *ab*. Šiuo atveju blokas *ab* yra visiškai analogiškas blokui *a12b12* ir sujungimo būdas neturi įtakos rezultatui. Lygiai taip pat būtų galima jungti ir pvz. *a2b1* atskirai, kitas monadas prijungti prie pagaminto bloko, kas taip pat neturėtų įtakos rezultatui ir būtų visiškai analogiška nurodytame pavyzdyje (18 pav.).

4.3.4. Atitikimas taisyklėms remiantis Scalaz funkcinio programavimo karkaso testais

Taip pat visas šias taisykles galima patikrinti ir naudojantis *Scala* programavimo kalbos testavimo įrankiais. Tą galima atlikti keliais būdais:

- kviečiant konkrečias funkcijas monadoms pagal monadų taisykles,
- naudojant *scalaz* karkase esančius įrankius monadų korektiškumo testavimui.

Scalaz funkcinio programavimo karkasas šiuo atveju pateikia labai patogų visų monadų taisyklių patikrinimą iškviečiant vieną patikrinimo metodą skirtą būtent minėtoms taisyklėms. Panagrinėjus taisyklių tikrinimo metodą giliau galima pastebėti, kad šis metodas lygiai taip pat kviečia atitinkamas monados funkcijas (apjungimo arba grąžinimo), tačiau suteikia galimybė patogiau generuoti testines reikšmes. Taigi, šis *scalaz* tikrinimo metodas reikalauja, kad būtų apibrėžtos dviejų tipų funkcijos:

- reikšmių lyginimo (*Equal*),
- reikšmių generavimo (*Arbitrary*).

```
implicit val FreeSTMEqual: Equal[ScalaFreeSTM.FreeSTM[Int]] =
              Equal.equal(runTransaction(_) == runTransaction(_))
```

19 pav. Monadų palyginimo apibrėžimas

Testų kūrimo metu aprašytoje monadų palyginimo funkcijoje (19 pav.) tiesiog paimamos dviejų monadų reikšmės, jos leidžiamos transakcijose ir abi transakcijų paleidimo funkcijų grąžinamos reikšmės yra palyginamos. Reikia pastebėti, kad *FreeSTM* monados reikšmės tipas yra *Int*. Toks pasirinkimas buvo priimtas, nes būtent tokio tipo reikalauja vėliau aprašoma reikšmių generavimo klasė.

```
implicit def FreeSTMArbitrary[A](implicit a: Arbitrary[A]):
  Arbitrary[FreeSTM[A]] =
 a map { a => val value = new SafeVal(a); writeTVal(a, value);
           (readTVal(value)); }
```

20 pav. Reikšmių generavimo funkcija

Sukurtame reikšmių generavimo metode (20 pav.) naudojama anksčiau aprašytos *writeTVal* ir *readTVal* funkcijos. Šios funkcija pasirinktos atsitiktinai ir realios reikšmės testų sėkmingumui ir monados korektiškumui neturi, nes kaip ir buvo aprašyta monadų taisyklių apibrėžimuose, visos taisyklėse naudojamos funkcijos gali būti bet kokios. Pačios reikšmių generavimo funkcijos esmė paprasta – imamos įvairios atsitiktinės a reikšmės ir dedamos į *SafeVal* tipo kintamuosius. Tokia gauta atsitiktinių duomenų aibė ir naudojama testavimui.

Paskutinis veiksmas po funkcijų aprašymo yra testų paleidimas, kurių nesėkmingas įvykdymas įrodytų, kad taisyklių monada neatitinka, o sėkmingas parodytų, kad testai taisyklių atitikimo nepaneigia. Monadų taisyklių testai kviečiami panaudojant šią komandą:

```
monad.laws[FreeSTM].check
```

21 pav. *Scalaz* karkaso monados taisyklių testų rinkinio paleidimas

Specializuotos kalbos realizavimo metu sukurta monada yra nurodoma laužtiniuose skliaustuose (21 pav.) ir paleidžiamas metodas *check*. Šis metodas įvykdo testus ir į konsolę išveda rezultatą.

```
+ monad.bind.associativity: OK, passed 100 tests.

+ monad.right identity: OK, passed 100 tests.

+ monad.left identity: OK, passed 100 tests.
```

22 pav. Testavimo rezultatai

Verta paminėti, kad pateikti rezultatai (22 pav.) yra tik dalis metodo *check* išspausdinamų reikšmių, nes šis metodas tikrina ir daugelį kitų taisyklių, kurios yra papildomos ar išvestinės iš monadų taisyklių. Tokiu būdu šis metodas visapusiškai patikrina testuojamą monadą ir jo rezultatai dėl to yra tikslesni.

Taigi, sukurta *FreeSTM* monada buvo patikrinta keliais būdais ir galime teigti, kad ji atitinka monadų taisykles. Tai įrodo, kad ši monada teisinga ir gali būti naudojama transakcinės atminties specializuotoje kalboje, nes nepažeis iš monadų sudaromų transakcinių blokų principų.

4.4. Standartinio uždavinio sprendimas naudojant sukurtą specializuotą kalbą

Parodysime minėtosios vienos juostos tilto problemos sprendimą panaudojant mūsų sukurtą Scala programavimo kalbos saugios transakcinės atminties specializuotąją kalbą (pilnas sprendimas pateikiamas 3 priede).

```
private def arriveOnBridge(car: Car) {
  val atomicArrive = {
    for {
      carsCount <- countTSet(carsOnBridge)
      currentDirection <- readTVal(direction)

      if((carsCount == 3) || (carsCount != 0 &&
           !(car.direction == currentDirection)))

      _ <- if(carsCount == 0)
            writeTVal(Option[Direction](car.direction), direction)
           else
            noOp

      _ <- addTSet(car, carsOnBridge)
    } yield ()
  }
  runTransaction(atomicArrive)
}
```

23 pav. Vienos juostos tilto problemos sprendimo fragmentas panaudojant saugią transakcinės atminties specializuotąją kalbą

Šiame fragmente (23 pav.) aprašoma mūsų jau minėtas *arriveOnBridge* transakcinis blokas. Čia yra naudojama Scala programavimo kalbos *for-yield* konstrukcija šiuo atveju skirta transakcinių blokų sudarymui, t. y. monadų sujungimui. Ši konstrukcija beveik atitinka *Scala STM* karkaso *atomic* blokus, tačiau jos vykdymo paleidimas skiriasi (tam yra sukurtas anksčiau aprašytas transakcijos paleidimo metodas). Nurodytas sprendimo būdas veikia lygiai taip pat kaip ir *Scala STM* biblioteka remiantis pateiktas sprendimas, tačiau nesaugių *IO* operacijų kvietimas čia yra apribotas: transakcinio bloko viduje vis tiek bus galima iškviesti nesaugias skaitymo ir rašymo operacijas išreikštinai jas įkėlus į monados vidų, tačiau laikantis nurodyto programavimo stiliaus ir naudojant tik aprašytas operacijas to bus išvengta.

4.5. Sukurtos specializuotosios kalbos ir Scala STM palyginimas

Šiame skyrelyje lyginamos sukurtos specializuotos kalbos ir Scala STM savybės remiantis tam tikrai iš anksto apibrėžtais kriterijais (1 lentelė). Kriterijai parinkti pagal pagrindines problemas kylančias kuriant STM mechanizmą, kurios buvo surastos jį nagrinėjant (1.3. poskyris).

1 lentelė. Specializuotos kalbos ir *Scala STM* palyginimas

Kriterijus	**Sukurta specializuotoji kalba**	***Scala STM***
Nesaugių I/O operacijų iškvietimas	Apribotas nesaugių operacijų kvietimas įveda programavimo stilių	Galima kviesti nesaugias operacijas
Plečiamumas	Plečiamos specializuotos kalbos operacijos (Numatytas tam tikras jų aprašymo būdas)	Plečiamos karkaso operacijos
Aprašymo ir interpretavimo atskyrimas	Atskirtas transakcinių blokų interpretavimas nuo aprašymo	Transakciniai blokai yra interpretuojami realiu laiku (aprašymo ir interpretavimo atskyrimo nėra)
Transakcinių kintamųjų aprašymas	Transakcinių kintamųjų aprašymas už transakcijos ribų	Transakcinių kintamųjų aprašymas už transakcijos ribų
Transakcijoje galimos operacijos	Yra konkretus transakcinių operacijų sąrašas	Transakcinio bloko viduje galima kviesti bet kokias operacijas

1 lentelėje pateiktas sukurtos transakcinės atminties specializuotosios kalbos Scala programavimo kalbai ir *Scala STM* karkaso palyginimas. Lyginami pagrindiniai šių bibliotekų ypatumai, norint pabrėžti privalumus ir trūkumus. Toliau šiame skyriuje detaliau aprašomi šių dviejų transakcinės atminties mechanizmą įgyvendinančių įrankių panašumai ir skirtumai.

Sukurta specializuotoji kalba turi daugelį *Scala STM* karkaso savybių, nes buvo kurta naudojant *Scala STM* bibliotekoje esančias konstrukcijas, jas uždengiant (angl. *wrap*) ir patobulinant. Taip pat vizualiai naujosios kalbos sintaksė ir kviečiamos operacijos yra labai panašios į jau *Scala STM* naudotas. Jeigu atidžiau pažvelgsime į *for-yield* konstrukciją ir anksčiau naudotus *atomic* blokus galime pastebėti, kad naujoji konstrukcija įveda tam tikrą stilių ir puikiai paslepia transakcinius *Txn* tipo kintamuosiuos reikalingus *Scala STM* atominiuose blokuose. Be to, naujojoje specializuotoje kalboje, dėka laisvųjų monadų, atskirtas interpretavimas ir užrašymas, kas mums leidžia daug lanksčiau užrašinėti transakcinius blokus, bei leisti juos nebūtinai kviečiant

funkciją su atominiu bloku. Tai reiškia, kad galima tiesiog grąžinti transakcijos bloką kaip rezultatą ir atėjus reikiamam momentui paleisti transakciją.

Didžiausias *Scala STM* trūkumas, kurį bandyta ištaisyti kuriant saugią transakcinės atminties specializuotąją kalbą atitinkančią visus tokį mechanizmą įgyvendinančių įrankių principus, buvo išorinį pasaulį veikiančių, nesaugių operacijų kvietimo transakcijose galimybė. Sukūrus naująją specializuotąją kalbą ir įvedus tam tikrą programavimo stilių, jos transakcijose apribojama tokių operacijų kvietimo galimybė. Visų galimų operacijų sąrašas buvo sudarytas bekuriant kalbą ir gali būti papildomas tik ją plečiant. Visos kitos operacijos sukurtos monados viduje negali būti kviečiamos.

Dar viena svarbi naujosios specializuotos kalbos savybė yra ta, kad joje išlaikyti visi transakcinės atminties principai ir veiksmai su transakciniais kintamaisiais – aibėmis ar rodyklėmis į reikšmę – gali būti atliekami tik transakcijos viduje. Net jeigu kviestume kokią nors operaciją, keičiančią reikšmę (pvz. *writeTVal*) ne *for-yield* bloko viduje mes turėtume tik tam tikro atskiro bloko iš vienos operacijos aprašymą, bet realių veiksmų su atmintimi nebūtų atliekama. Tai vėlgi buvo pasiekta visiškai atskyrus interpretavimą nuo transakcinių blokų konstravimo.

Taip pat naujoji kalba naudoja labai panašų transakcinių kintamųjų inicializavimo ir aprašymo būdą kaip ir *Scala STM*. Kitas būdas inicializuoti kintamuosius būtų visų inicializavimo operacijų aprašymas transakcijose, tačiau to buvo atsisakyta. Vienas iš didžiausių transakcinės atminties kintamųjų inicializavimo ne transakcijoje privalumas yra tas, kad labai dažnai transkacinę atmintį naudojančiose programose yra reikalingi iš anksto inicializuoti globalūs kintamieji. Jeigu juos sukurti ir inicializuoti galime tik transakcijoje, tai visose tokiose programose bus reikalingas vienas inicializuojančios transakcijos blokas. Be to, inicializavimas ir kintamojo vietos išskyrimas iš principo nėra transakcinė operacija dėl to, kad kelios gijos neturėtų konkuruoti dėl dar neišskirtos atminties. Būtent dėl to yra daug logiškiau išskirti atmintį prieš vykdant bet kokias transakcijas, o galimybė prieš tai dar ką nors ten ir įrašyti suteikia daugiau galimybių. Aišku, tai gali kelti problemų dėl atminties kintamųjų perskyrinėjimo programos vykdymo metu, tačiau realiai tos pačios atminties ląstelės negalės būti paveiktos ne transakcijoje, jei į jas rodo transakcinis kintamasis.

Dar vienas sukurtos saugios transakcinės atminties specializuotos kalbos privalumas yra tas, kad yra aprašytas tam tikras naujų kalbos operacijų kūrimo principas. Naudojantis šiuo principu galima lengvai sukurti naujas transakcines operacijas panaudojant *Scala STM* jau esančias funkcijas ir įrankius arba kombinuojant jau esančius ir kuriant naujus transakcinius funkcionalumus.

Taigi, sukurta transakcinės atminties specializuotoji kalba sprendžia *Scala STM* karkaso trūkumus ir pateikia tam tikrą programavimo būdą leidžiantį išvengti nesaugių operacijų kvietimo.

Nors kuriant naujas operacijas ir galima aprašyti nesaugius išvedimo ir įvedimo veiksmus (tai leidžia *Scala* programavimo kalbos lankstumas), dabartinis sukurtas operacijų rinkinys gali būti naudojamas saugiai ir jeigu programuojant bus naudojamasi tik šiomis sukurtomis funkcijomis, programuotojui rūpintis transakcijos korektiškumu nereikės.

4.6. Scala transakcinės atminties specializuotosios kalbos tobulinimas

Sukurta transakcinės atminties specializuotoji kalba ištaiso vieną pagrindinių *Scala STM* trūkumų – nesaugių operacijų transakcijos viduje kvietimą – tačiau gali būti tobulinama didesniam naudojimo patogumui. Šiame skyriuje apibrėšime keletą pagrindinių sukurtos specializuotos kalbos trūkumų ir pasiūlysime galimus tolesnius bibliotekos vystymo kelius.

Vienas iš didžiausių sukurtos specializuotos kalbos trūkumų yra tas, kad labai griežtai apibrėždami transakcines operacijas prarandame lankstumą transakcijos viduje. Iš vienos pusės tas leidžia tik su transakcija susijusių operacijų kvietimus, tačiau iš kitos pusės apriboja programavimo lankstumą. Šio trūkumo pašalinimui reikėtų išanalizuoti ir apsibrėžti daugiau transakcijose naudingų specializuotos kalbos operacijų (pvz. transakcinių kintamųjų reikšmių aritmetinės operacijos). Apsibrėžus tokias operacijas, labai nesunku jas įgyvendinti panaudojant esančius naujų operacijų kūrimo principus.

Dar vienas tobulintinas sukurtos kalbos aspektas yra transakcinių kintamųjų tipai. Šiuo metu yra dvi galimos struktūros, kurių pilnai pakanka daugumai užduočių, tačiau sudėtingesniems ir daugiau logikos reikalaujantiems uždaviniams reikalingos ir sudėtingesnės struktūros. Naujas struktūras, tokias kaip eilė, stekas, sąrašas, galima sukurti pasinaudojant jau esančiais tipais, žymiai praplečiant specializuotosios kalbos galimybes.

Taigi, pagrindiniai kalbos tobulinimo aspektai yra susiję su naujų operacijų ir struktūrų kūrimu. Tai padėtų pagerinti specializuotos kalbos panaudojamumą, išlaikant pagrindinius transakcinės atminties principus ir užtikrinti kalbos pilnumą.

Rezultatai ir išvados

Šiame darbe buvo nagrinėjami transakcinės atminties mechanizmo pritaikymo principai ir aprašomi pagrindiniai tokio mechanizmo sukūrimo ir panaudojimo ypatumai. Išnagrinėjus transakcinės atminties mechanizmą ir Scala funkcinio programavimo kalbos *Scala STM* biblioteką, buvo įvardinti pagrindiniai jos trūkumai ir pasiūlyti jų sprendimo variantai. Darbo metu buvo sukurta saugios transakcinės atminties ir konkurencinio programavimo su bendrų resursų naudojimu bazinius reikalavimus atitinkanti Scala programavimo kalbos transakcinės atminties specializuotoji kalba. Sukurta kalba užtikrina, kad nebūtų galima transakcijos viduje iškviesti operacijų, kurios veikia išorinį pasaulį ir apibrėžia tam tikrų transakcijoje galimų naudoti funkcijų aibę.

Darbo rezultatai:

- sukurta laisvoji monada, apribojanti nesaugių operacijų kvietimą STM transakcijose,
- aprašyti laisvosios monados kūrimo ypatumai, kuriuos galima panaudoti toliau kuriant laisvąsias monadas *Scala* kalboje,
- įrodytas sukurtos monados korektiškumas,
- sukurta transakcinės atminties specializuotoji kalbą naudojanti sukurtą monadą.

Išanalizavus pagrindinius principus ir sukūrus juos įgyvendinančią laisvąją monadą *Scala* programavimo kalboje, buvo prieita išvados, kad sukurta laisvoji monada yra korektiška ir apriboja nesaugių operacijų kvietimą transakcijose. Be to darbe pateikiama pagrindiniai sukurtos specializuotos kalbos privalumai ir trūkumai ir aprašomos pagrindinės tobulinimo kryptys.

ŠALTINIAI

[Bja12] BJARNASON, Rúnar Oli. Stackless Scala With Free Monads. 2012. Prieiga per internetą: <http://days2012.scala-lang.org/sites/days2012/files/bjarnason_trampolines.pdf>

[BW90] BARR, Michael; WELLS, Charles. Category theory for computing science. New York: Prentice Hall, 1990. Prieiga per internetą: < http://www.math.mcgill.ca/triples/Barr-Wells-ctcs.pdf>

[Cla99] CLAESSEN, Koen. A poor man's concurrency monad. Journal of Functional Programming, 1999, 9.03: 313-323. Prieiga per internetą: <http://www.di.ens.fr/~pouzet/cours/systeme/bib/koen-concurrency-poor-man-jpf93.pdf >

[HM93] HERLIHY, Maurice; MOSS, J. Eliot B. Transactional memory: Architectural support for lock-free data structures. ACM, 1993. Prieiga per internetą: <http://www.cs.utexas.edu/~pingali/CS395T/2009fa/lectures/herlihy93transactional.pdf>

[Jon07] JONES, Simon Peyton. Beautiful concurrency. Beautiful Code: Leading Programmers Explain How They Think, 2007, 385-406. Prieiga per internetą: <http://research.microsoft.com/en-us/um/people/simonpj/papers/stm/beautiful.pdf>

[Mar05] MARSHALL, Casey. Software Transactional Memory. Computer Science, 2005, 203. Prieiga per internetą: < http://metastatic.org/Grad/Software%20Transactional%20Memory.pdf>

[MK99] MAGEE, Jeff; KRAMER, Jeff. State models and java programs. wiley, 1999. Prieiga per internetą: < http://flylib.com/books/en/2.752.1.48/1/>

[Sco06] SCOTT, Michael L. Sequential specification of transactional memory semantics. In: Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing, Ottawa, ON, Canada. 2006.
Prieiga per internetą:
< http://192.5.53.208/u/scott/papers/2006_TRANSACT_formal_STM.pdf>

[ST97] SHAVIT, Nir; TOUITOU, Dan. Software transactional memory. Distributed Computing, 1997, 10.2: 99-116.
Prieiga per internetą:
<http://link.springer.com/article/10.1007/s004460050028#page-1>

[Wad95] WADLER, Philip. Monads for functional programming. In: Advanced Functional Programming. Springer Berlin Heidelberg, 1995. p. 24-52..
Prieiga per internetą:
< http://flint.cs.yale.edu/trifonov/cs629/WadlerMonadsForFP.pdf >

[Wei08] WEIMERSKIRCH, Michel. Software Transactional Memory. 2008.
Prieiga per internetą:
<http://michel.weimerskirch.net/wp-content/uploads/2008/02/software_transactional_memory.pdf>

PRIEDAI

1 Priedas. Vienos juostos tilto problemos sprendimas su Scala STM

```
case class Car(direction: Direction, number: Int)

sealed trait Direction
object Right extends Direction
object Left extends Direction

object SingleLaneBridge {
  private val carsOnBridge: TSet[Car] = TSet();
  private val bridgeDirection: Ref[Option[Direction]] = Ref(None);

  private def arriveOnBridge(car: Car) {
    atomic { implicit Txn =>
      if ((carsOnBridge.size == 3) ||
        (carsOnBridge.size != 0 && !(car.direction == bridgeDirection))) {
        retry;
      }

      if (carsOnBridge.size == 0) {
        bridgeDirection() = Option[Direction](car.direction);
      }

      carsOnBridge += car;
    }
  }

  private def exitBridge(car: Car) {
    atomic { implicit Txn =>
      carsOnBridge.-=(car);
    }
  }
```

2 Priedas. Saugi transakcinės atminties specializuotoji kalba

```
object ScalaFreeSTM {

  final class SafeSet[A]() {
    private[ScalaFreeSTM] val tSet: TSet[A] = TSet()

    private[ScalaFreeSTM] def add(value: A)(implicit txn: InTxn): Unit = {
      tSet.+=(value)
    }

    private[ScalaFreeSTM] def remove(value: A)(implicit txn: InTxn): Unit = {
      tSet.-=(value)
    }
  }

  final class SafeVal[A](a: A) {
    private[ScalaFreeSTM] val tRef: Ref[A] = Ref(a)
  }

  sealed trait STMOp[A]
  private object STMOp {
    case class AddTSet[A](value: A, set: SafeSet[A]) extends STMOp[Unit]
    case class RemoveTSet[A](value: A, set: SafeSet[A]) extends STMOp[Unit]
    case class CountTSet[A](set: SafeSet[A]) extends STMOp[Int]

    case class ReadTVal[A](tVal: SafeVal[A]) extends STMOp[A]
    case class WriteTVal[A](value: A, tVal: SafeVal[A]) extends STMOp[Unit]

    case object Retry extends STMOp[Unit]
    case object NoOp extends STMOp[Unit]
  }
  import STMOp._

  type STMCoyoneda[A] = Coyoneda[STMOp, A]
  type FreeSTM[A] = Free[STMCoyoneda, A]



  def addTSet[A](value: A, set: SafeSet[A]): FreeSTM[Unit] = liftFC[STMOp,
Unit](AddTSet(value, set))
  def removeTSet[A](value: A, set: SafeSet[A]): FreeSTM[Unit] = liftFC[STMOp,
Unit](RemoveTSet(value, set))
  def countTSet[A](set: SafeSet[A]): FreeSTM[Int] = liftFC[STMOp,
Int](CountTSet(set))



  def readTVal[A](tVal: SafeVal[A]): FreeSTM[A] = liftFC[STMOp,
A](ReadTVal(tVal))
  def writeTVal[A](value: A, tVal: SafeVal[A]): FreeSTM[Unit] = liftFC[STMOp,
Unit](WriteTVal(value, tVal))
  def noOp[A]: FreeSTM[Unit] = liftFC[STMOp, Unit](NoOp)



  implicit class FreeSTMOps[A](freeSTM: FreeSTM[A]) {
    def filter(f: Unit => Boolean): FreeSTM[Unit] = {
      f() ? liftFC[STMOp, Unit](Retry) | noOp
    }
  }
```

3 Priedas. Vienos juostos tilto problemos sprendimas su sukurta specializuota kalba

```
case class Car(direction: Direction, number: Int)

sealed trait Direction
object Right extends Direction
object Left extends Direction

object SingleLaneBridge {
  private val carsOnBridge: SafeSet[Car] = new SafeSet();
  private val direction: SafeVal[Option[Direction]] = new SafeVal(None);

  private def arriveOnBridge(car: Car) {
    val atomicArrive = {
      for {
        carsCount <- countTSet(carsOnBridge)
        currentDirection <- readTVal(direction)

        if((carsCount == 3) || (carsCount != 0 &&
           !(car.direction == currentDirection)))

        _ <- if(carsCount == 0)
              writeTVal(Option[Direction](car.direction), direction)
             else
              noOp

        _ <- addTSet(car, carsOnBridge)
      } yield ()
    }
    runTransaction(atomicArrive)
  }

  private def exitBridge(car: Car) {
    var atomicExit = {
      for {
        _ <- removeTSet(car, carsOnBridge)
      } yield ()
    }
    runTransaction(atomicExit)
  }

  def crossBridge(car: Car) {
    arriveOnBridge(car);
    Thread.sleep(1000);
    exitBridge(car);
  }
}
```