UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANDRÉ MARTINS FERREIRA

Retry-transaction: Uma nova função primitiva de bloqueio para a STM de Clojure.

Trabalho de Graduação.

Prof. Dr. Álvaro Moreira
Orientador

Porto Alegre, julho de 2012

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

André Martins Ferreira,

Retry-transaction: Uma nova função primitiva de bloqueio para a STM de Clojure. /

André Martins Ferreira. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2012.

42 f.: il.

Trabalho de Conclusão (bacharelado) – Universidade Federal do Rio Grande do Sul. Curso de Ciência da Computação, Porto Alegre, BR–RS, 2012. Orientador: Álvaro Moreira.

1. Memória transacional de software. 2. Clojure. 3. Programação funcional. I. Moreira, Álvaro. II. Título.

AGRADECIMENTOS

Agradeço meus pais, Marcos e Sandra, pelo suporte não só durante a produção deste trabalho, mas também durante toda minha vida.

Agradeço ao Prof. Álvaro Moreira pela sua orientação neste trabalho.

Agradeço a UFRGS por ter me proporcionado um ensino de qualidade, sem o qual eu não teria capacidade para entender os conceitos que foram necessários na construção deste trabalho.

Agradeço a minha namorada Alana pela sua compreensão e apoio.

E por fim agradeço aos meus amigos e aos meus colegas de curso, com os quais aprendi muitas coisas e fiz ligações que espero que durem para o resto da vida.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CLR	Common Language Runtime
CSP	Communicating Sequential Processes
JVM	Java Virtual Machine
JIT	Just in Time
MVCC	Multiversion concurrency control
STM	Software Transactional Memory

LISTA DE FIGURAS

RESUMO

Neste trabalho é proposta uma nova função primitiva, *retry-transaction*, para o sistema de Memória Transacional de Software (STM) de Clojure.

STM é vista por muitos pesquisadores como uma possível forma de simplificar programação concorrente. Porém, uma ausência em muitos sistemas de STM é uma forma de conciliar código a ser executado na STM com código que deve bloquear. Este trabalho se baseia numa proposta originalmente para a STM de Haskell.

São apresentados para contextualizar a proposta: a linguagem de programação Clojure; estruturas de dados puramente funcionais para facilitar a escrita de código funcional, que é um requisito da STM de Clojure; o conceito de STM; como é realizado STM em Clojure. Em seguida é apresentada a proposta central deste trabalho, que diz respeito a uma nova função para a STM, *retry-transaction*, que permite a escrita de código bloqueante utilizando a STM.

Palavras-chave: Memória transacional de software, clojure, programação funcional.

Retry-transaction: A new primitive function for blocking in the Clojure STM.

ABSTRACT

This paper proposes a new primitive function, *retry-transaction*, for Clojure's STM system.

Software Transactional Memory is seen by many researchers as a possible way to simplify concurrent programming. However, there is an absence in many STM systems of a mechanism for conciliating code to be executed inside the STM with code that has to block. This work is based on a proposal originally designed for Haskell's STM.

This proposal contains the following: the Clojure programming language; purely functional data structures to facilitate writing functional code, a requisite for Clojure's STM; the concept of STM; Clojure's STM. Following that, this paper's main proposal is described: a new primitive function for Clojure's STM,*retry-transaction*, which allows blocking code using STM.

Keywords: STM, Clojure.

1 INTRODUÇÃO

Escrever código concorrente é difícil. Atualmente o método mais utilizado para isso usa *locks* para sincronização. Uma das dificuldades é que não é fácil compor sistemas construídos com *locks* sem conhecer os funcionamentos internos dos sistemas sendo compostos, como quais *locks* devem ser adquiridas para acessar um dado recurso e de que forma deve ser feita essa aquisição.

Para responder a essas dificuldades muitos pesquisadores propõem uma alternativa: o uso de **memória transacional de software** (Shavit e Touitou 1995) (ou STM, para *Software Transactional Memory*), com a qual é possível marcar regiões do código para que as ações sobre a memória contidas na região marcada sejam feitas de forma atômica.

Clojure (Hickey 2012) é uma linguagem moderna que implementa um sistema de STM. Uma ausência em muitos sistemas de STM, inclusive Clojure, é de um mecanismo para escrever código bloqueante que utilize a STM.

Escrever código que bloqueia até que uma certa condição se torne verdade é útil em situações como E/S, ou sincronização entre *threads*. (Harris et al. 2005) propôs uma forma de escrever código que bloqueia dentro de STM em Concurrent Haskell (Jones et al. 1996), com a função *retry*. Este trabalho propõe a adaptação desse mecanismo de Haskell para a linguagem Clojure.

1.1 Objetivos

Este trabalho se propõe a:

- Estudar como é feito o uso de STM em Clojure.
- Implementar a função *retry* de (Harris et al. 2005) em uma versão modificada da linguagem Clojure.
- Testar seu funcionamento com exemplos de programas que utilizem a função.

1.2 Estrutura do trabalho

Este trabalho é organizado da seguinte forma: no capítulo 2 é descrita a linguagem de programação Clojure e é apresentado o conceito de estruturas de dados puramente funcionais e como elas são usadas em Clojure; no capítulo 3 é apresentado o conceito de Memória Transacional de Software e especificamente como ele é realizado em Clojure; no capítulo 4 é apresentada a proposta deste trabalho, a função *retry-transaction*; no capítulo 5 são apresentados experimentos para testar a flexibilidade da proposta, assim como verificar seu funcionamento; no capítulo 6 é apresentado detalhas da implementação da

linguagem Clojure, e como foi feita a implementação da função *retry-transaction*; no capítulo 7 são discutidas as conclusões sobre STM, conclusões da proposta e possíveis melhorias futuras.

1.3 Trabalhos Relacionados

O artigo (Harris et al. 2005) introduz a função *retry* para a STM em Concurrent Haskell, e foi a principal inspiração para este trabalho.

O artigo (Adl-Tabatabai et al. 2006) implementa as funções de (Harris et al. 2005) numa linguagem imperativa ao estilo de Java.

X10 (Charles et al. 2005) é uma linguagem com um operador condicional de blocos atômicos. Nela existem duas formas de realizar uma ação atômicamente: ações não condicionais na forma *atomic S*, onde *S* é uma expressão, e ações condicionais na forma *when (c) S*, no qual a execução é suspensa até a condição *c* se tornar verdade, e após isso a expressão *S* é executada atomicamente.

2 CLOJURE

Clojure é uma linguagem de programação voltada principalmente para a *JVM*, embora esteja também disponível no CLR e em Javascript. Clojure é um membro da família Lisp de linguagens de programação, com um foco em programação funcional e concorrência. Embora esteja na família de linguagens Lisp, ela não visa compatibilidade com nenhum Lisp anterior. Este capítulo apresenta uma breve introdução para a linguagem funcional Clojure. Ele discute o que são estruturas de dados puramente funcionais, qual é o seu uso em Clojure, como elas se relacionam com STM e quais estruturas são implementadas pela linguagem.

2.1 Motivação

Lisp e suas variações historicamente sempre tiveram forte uso como laboratórios de novas linguagens e idiomas (Steele Jr. e Gabriel 1993). A facilidade de expandir a linguagem seja pelo uso de macros ou pelo seu dinamismo (tipagem dinâmica, *garbage collection*, funções como valores de primeira classe) fez com que Lisp e seus dialetos sempre fizessem parte da vanguarda de exploração de novas adições a linguagens de programação.

Em primeiro lugar, Clojure visa se modernizar rompendo com os Lisps anteriores, e assim não precisar manter o legado e idiossincrasias passadas. Até mesmo a função *cons*, presente desde a definição original da linguagem Lisp (McCarthy 1960) tem seu significado alterado, só podendo ser usada em listas e sequências, e o uso de células *cons* para construção de pares e listas impróprias não é mais possível. Os nomes tradicionais de Lisp *car* e *cdr* como primitivas para o acesso a listas tambem foram abandonados, sendo usados apenas os nomes *first* e *rest*.

Clojure evolui rapidamente graças a uma comunidade online ativa, uma implementação principal definida, e um *Benevolent Dictator for Life* (Raymond 2000), Rich Hikey, definindo a visão e direção do futuro da linguagem e resolvendo disputas e argumentos técnicos na comunidade.

As vantagens de Clojure ser implementada na *JVM* são diversas. É possível utilizar bibliotecas de Java diretamente de Clojure, vencendo assim a inercia inicial de não haver bibliotecas para uma linguagem nova. A *JVM* é uma plataforma madura, com um *garbage collector* altamente otimizado, compilador *JIT*, etc. Todos esses mecanismos podem ser utilizados por Clojure, poupando assim tempo de desenvolvimento. Isso acelerou sua adoção, sendo hoje em dia a vigésima segunda linguagem mais popular no GitHub (GitHub Inc. 2012), a frente de Common Lisp (vigésima sétima) e Scheme (vigésima oitava). Clojure já está em uso na indústria, por empresas como Citigroup e Tianya. Uma lista não exaustiva de empresas que utilizaram Clojure com sucesso está disponível em

(Redinger 2012).

A abstração de interfaces comuns é um grande foco da linguagem. Por exemplo, as funções *first* e *rest*, que normalmente são funções de manipulação de listas, em Clojure atuam sobre sequências, uma interface de diversas estruturas de dados que podem ser acessadas sequencialmente, com listas sendo uma das várias estruturas de dados que implementam essa interface comum. O uso de interfaces comuns entre diferentes estruturas de dados facilita o reuso de algoritmos.

2.2 Estruturas de dados puramente funcionais

Estruturas de dados puramente funcionais são aquelas cujas operações disponíveis na sua interface não fazem atualizações destrutivas. Há dois benefícios para seu uso em programas paralelos: O primeiro é que código funcional é mais facilmente paralelizável, visto que ele nunca causa efeitos colaterais; o segundo benefício é que elas são *persistentes* (Driscoll et al. 1986). As estruturas de dados não funcionais são *efêmeras* no sentido que fazer uma mudança na estrutura destrói a versão antiga, deixando apenas a nova versão disponível. Uma estrutura de dados *persistente* permite acesso às versões antigas da estrutura. Para aumentar a eficiência das operações, as versões novas e antigas compartilham memória, o que é possível graças a segurança de que sua memória compartilhada nunca será modificada.

Clojure aproveita essas características de estruturas persistentes no seu idioma de uso da STM, na qual o código a ser executado dentro de uma transação pode ter que ser executado mais de uma vez caso um conflito ocorra. Uma *thread* que manipula uma estrutura de dados não interfere com outra que está manipulando a mesma estrutura, devido a ausência de atualizações destrutivas. E caso seja necessário repetir uma transação, não é necessário tentar desfazer operações feitas em uma estrutura, pois sua versão antiga ainda está disponível.

2.3 Visão Global das estruturas de Clojure

As principais estruturas de Clojure são *vectors*, *maps*, *sets* e *lists*. Outra estrutura importante que também é implementada por Clojure são *queues*, embora essa não possua suporte de criação de instâncias na sintaxe da linguagem.

As estruturas são comumente manipuladas através de funções genéricas. Na chamada *conj coleção elemento* por exemplo, é retornada uma nova coleção que é a coleção dada com o elemento inserido nela, embora onde, ou como essa inserção é feita, depende da estrutura usada. A chamada da função *assoc coleção chave valor* retorna uma nova coleção com a chave associada ao valor, e pode ser usada tanto em coleções como *maps* e *vectors*.

2.3.1 Lists

Listas permitem adição de elementos na frente em tempo constante, e acesso aos elementos internos em complexidade de tempo linear.

A Figura 2.1 demonstra um exemplo de seu uso. A função *conj* insere elementos no início da lista como mostra a linha 3. As funções *pop* e *rest* retornam a lista sem seu primeiro elemento como mostram as linhas 6 e 7. As funções *peek* e *first* retornam a lista sem seu primeiro elemento como mostra as linhas 10 e 11. Todas essas operações possuem complexidade $O(1)$. A função *nth*, demonstrada na linha 14, retorna o enésimo elemento da lista, com uma complexidade $O(n)$.

```
1  (def l (list 4 3 2))
2
3  (conj l 8)
4  ;retornam (8 4 3 2)
5
6  (rest l)
7  (pop l)
8  ;retornam (3 2)
9
10 (first l)
11 (peek l)
12 ;retornam 4
13
14 (nth l 1)
15 ;retorna 3
```

Figura 2.1: Exemplo de uso listas em Clojure.

2.3.2 Vectors

Um *vector* é uma coleção de valores indexados por inteiros em um intervalo continuo. A Figura 2.2 demonstra um exemplo de seu uso. Na linha 1 um *vector v* é definido com uma *string* "a string", o número 3, e uma *keyword :keyword*. Inserções de elementos no fim de um *vector* são feitas com a função *conj* como mostra a linha 10 do exemplo. A complexidade da operação é $O(1)$. Acessos a um elemento são feitos com as funções *get* ou *nth*, demonstrados nas linhas 6 a 8 do exemplo. Atualizações não destrutivas de um elemento são feitas com a função *assoc*, como mostra a linha 3. É possível remover o último elemento do *vector* com a função *pop*, como mostra a linha 13. Acessos e atualizações possuem complexidade de $\log_{32} N$.

```
1  (def v ["a string" 3 :keyword])
2
3  (assoc v 1 "another string")
4  ;retorna ["a string" "another string" :keyword]
5
6  (nth v 2)
7  (get v 2)
8  ;retornam :keyword
9
10 (conj v 42)
11 ;retorna ["a string" 3 :keyword 42]
12
13 (pop v)
14 ;retorna ["a string" 3]
```

Figura 2.2: Exemplo de uso vectors em Clojure.

2.3.3 Maps

Um *map* é uma coleção para associar elementos com chaves. A Figura 2.3 demonstra o uso de *maps*. Na linha 1 é definido um *map m*, com a chave *:name* associada ao string "João", e a chave *:age* associada ao número 23. A função *assoc* associa uma chave a um elemento no *map*, como mostra a linha 3. Repare que quando a estrutura é impressa pela linguagem ou iterada, a ordem das associações pode ser diferente da ordem em que

elas foram inseridas. A função *conj* também associa uma chave a um elemento no *map*, porém eles devem estar contidos em uma estrutura como demonstra a linha 6. A função *get* extrai o elemento associado a chave no *map*, como mostra a linha 9. A função *dissoc* retorna o *map* com a chave removida, como mostra a linha 12.

Maps são uma implementação de *Hash Array Mapped Trie* de (Bagwell 2001).

```
1  (def m {:name "João", :age 23})
2
3  (assoc m :name "Pedro")
4  ;retorna {:age 23, :name "Pedro"}
5
6  (conj m ["chave" "valor"])
7  ;retorna {:age 23, :name "João", "chave" "valor"}
8
9  (get m :age)
10 ;retorna 23
11
12 (dissoc m :age)
13 ;retorna {:name "João"}
```

Figura 2.3: Exemplo de uso maps em Clojure.

2.3.4 Sets

Um *set* é uma coleção para definir um conjunto de elementos. A Figura 2.4 demonstra o uso de *sets*. Na linha 1 um *set* é definido com os elementos 2, 5, 3 e 7. A função *get* verifica se um elemento está contido pelo conjunto, retornando ele mesmo caso ele esteja contido como mostra a linha 3, e nil caso ele não esteja contido como mostra a linha 6. A função *conj* insere um novo elemento no conjunto como mostra a linha 9. A função *disj* remove um elemento do conjunto como mostra a linha 12.

Sets são implementados com *maps* na qual os objetos são chaves deles mesmos.

```
1  (def s #{2 5 3 7})
2
3  (get s 2)
4  ;retorna 2
5
6  (get s 4)
7  ;retorna nil
8
9  (conj s 42)
10 ;retorna #{2 3 5 7 42}
11
12 (disj s 3)
13 ;retorna #{2 5 7}
```

Figura 2.4: Exemplo de uso sets em Clojure.

2.4 Transientes

O uso de estruturas funcionais torna interfaces mais robustas, ainda mais em um cenário paralelo. Porém, às vezes é necessário para aumentar o desempenho realizar

operações que modificam estruturas de dados de forma destrutiva, e o programador consegue garantir que suas modificações são seguras pois são feitas em um contexto local, sem a modificação de estado global. Clojure utiliza o conceito de transientes para realizar isso (Hickey 2012).

A Figura 2.5 mostra um exemplo de como transientes são usados. O exemplo mostra duas funções que realizam o mesmo trabalho, construir um *vector* com elementos de 0 a n. A função *vrange* definida nas linhas 1 a 5 utiliza as funções de *vectors* de Clojure para realizar essa tarefa, enquanto a função *vrange2* definida nas linhas 7 a 11 demonstra uma versão otimizada com transientes.

```
1  (defn vrange [n]
2    (loop [i 0 v []]
3      (if (< i n)
4        (recur (inc i) (conj v i))
5        v)))
6
7  (defn vrange2 [n]
8    (loop [i 0 v (transient [])]
9      (if (< i n)
10       (recur (inc i) (conj! v i))
11       (persistent! v))))
12
13 (time (def v (vrange 1000000)))
14 "Elapsed time: 297.444 msecs"
15
16 (time (def v2 (vrange2 1000000)))
17 "Elapsed time: 34.428 msecs"
```

Figura 2.5: Exemplo de uso de transientes em Clojure.

É possível obter uma versão transiente de uma estrutura de dados persistente usando a função *transient*. As operações de leitura sobre a estrutura continuam as mesmas. Para as operações de "modificação" se deve inserir um ! no final do nome de cada função (*conj!* ao invés de *conj*, compare a linha 10 com a linha 4). Seu uso continua parecido, com a nova estrutura sendo o valor retornado da função, com a restrição agora que o código não deve acessar as cópias antigas da estrutura após a sua modificação. O seu uso deve ser isolado a uma única *thread*. Para evitar que seja feito uso das formas transientes entre *threads*, o que poderia levar a erros já que elas não são sincronizadas, Clojure salva na estrutura persistente qual *thread* que criou ela, e se outra *thread* tenta acessá-la, uma exceção é gerada.

Quando as operações internas são terminadas, a função *persistent!* retorna uma "cópia" persistente da estrutura transiente. Diferente da função *transient*, a estrutura antiga não pode mais ser usada após a chamada de *persistent!*. Ambas as funções geram as cópias com complexidade $O(1)$.

Como todas as modificações a uma estrutura transiente devem ser feitas localmente e não são visíveis as outras *threads*, o uso de estruturas transientes é seguro e recomendado junto à STM.

3 MEMÓRIA TRANSACIONAL DE SOFTWARE

O capítulo anterior apresentou Clojure com ênfase no conceito de estruturas de dados puramente funcionais. Este capítulo inicia apresentando como programação concorrente com memória compartilhada é feita tradicionalmente, com o uso de *locks*. Ele então demonstra fraquezas da abordagem, e apresenta uma alternativa, Memória Transacional de Software. O uso de STM em Clojure é então explicado e exemplificado.

3.1 Motivação

Programação concorrente é notoriamente difícil, e está se tornando cada vez mais relevante. A abstração mais comumente usada para sistemas de memória compartilhada são *locks*.

Para uma *thread* acessar um dado recurso compartilhado com outras *threads* é primeiramente necessário obter a *lock* associada a ele. Se a *lock* já está em uso por outra *thread*, ocorre um bloqueio até que a *lock* seja liberada e então adquirida. Caso seja logicamente impossível o programa progredir devido a um ciclo de dependências de *locks*, é dito que o programa entrou em *deadlock* (*e.g.* uma *thread* que tem a *lock* A quer a *lock* B, que está na posse de outra *thread* que quer a *lock A*). Diversas estratégias existem para evitar *deadlocks*, como adquirir todas as *locks* necessárias no início da operação seguindo uma ordem fixa. Mas essas estratégias devem ser definidas pelo programador, e em geral não há suporte do compilador ou *runtime system* de que não ocorram de concorrência. Quando se vai além de problemas simples, fica difícil lidar com a complexidade. Assim é difícil de atingir escalabilidade e robustez, pois, ou o programa utiliza uma granularidade muito grande de *locks* (*locks* para sistemas inteiros) ou usa algoritmos complexos para evitar *deadlocks*. Uma das fontes de complexidade é que mesmo quando se tem abstrações corretamente implementadas com *locks*, não se pode compor abstrações maiores sem acesso aos mecanismos internos do sistema (as *locks* utilizadas internamente e o algoritmo de prevenção de *deadlock*).

A Figura 3.1 mostra um sistema simplificado de contas bancárias escrito em Java utilizando *locks*. As operações de debitar e creditar contas nas linhas 14 a 24 são simples e funcionam em ambientes concorrentes. Porém, ao tentar se compor ambas operações para que seja feita uma transferência atomicamente entre contas como mostram as linhas 41 a 56 é necessário conhecimento interno sobre as *locks*, e o uso de um mecanismo (nesse caso a ordenação de *locks*) para evitar *deadlocks*. O código realmente necessário da operação de transferência se resume a 5 linhas, da linha 48 a 53, enquanto o código para que o sistema não entre em *deadlock* ocupa 7 linhas da função, além de uma classe extra para definir a ordem entre *locks*.

```java
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Arrays;

public class Account {
        private int money;
        private SortableReentrantLock lock = new SortableReentrantLock();;
        private final Condition moreMoney  = lock.newCondition();

        public Account(int money) {
                this.money = money;
        }
        void debit(int amount) {
                lock.lock();
                money -= amount;
                lock.unlock();
        }
        void credit(int amount) {
                lock.lock();
                money += amount;
                moreMoney.signalAll();
                lock.unlock();
        }

        static final AtomicInteger lockOrderCounter = new AtomicInteger(0);
        class SortableReentrantLock extends ReentrantLock implements Comparable {

                Integer order;
                public SortableReentrantLock() {
                        order = lockOrderCounter.incrementAndGet();
                }

                @Override
                public int compareTo(Object o) {
                        return order.compareTo(((SortableReentrantLock)o).order);
                }

        }

        void transfer(Account toAccount, int amount) throws Exception {
                SortableReentrantLock locks[] = new SortableReentrantLock[2];
                locks[0] = this.lock;
                locks[1] = toAccount.lock;
                Arrays.sort(locks);
                for (SortableReentrantLock l : locks)
                        l.lock();
                if (money >= amount) {
                        debit(amount);
                        toAccount.credit(amount);
                }
                else
                        throw new Exception();
                for (SortableReentrantLock l : locks)
                        l.unlock();
        }
}
```

Figura 3.1: Exemplo de código com locks em Java.

Memória Transacional de Software é um mecanismo para gerenciar a complexidade causada pelo paralelismo em sistemas de memória compartilhada. Ela permite a escrita de código na quais várias escritas e leituras de memória compartilhada ocorrem logicamente em um único momento no tempo. Uma vantagem é que se torna possível compor diferentes operações atômicas em operações atômicas maiores, de forma automática e correta, com a possibilidade de introduzir *deadlocks* sendo eliminada por construção (Shavit e Touitou 1995). O mecanismo que permite que as transações sejam logicamente atômicas é a detecção automática de conflitos entre transações que estejam sendo executadas paralelamente, com a reexecução do código da transação para resolver os conflitos.

A maior crítica contra STM é de que ela causa um baixo desempenho. Por exemplo, em (Cascaval et al. 2008) é testado o desempenho de duas diferentes implementações STMs. Foi concluído que elas sofreram de *overheads* muito grandes em relação à versão sequencial do código. O motivo que a conclusão de (Cascaval et al. 2008) não necessariamente se aplica a Clojure é que ela é baseada nas seguintes premissas: que o compilador de um sistema com STM deve tornar todos acessos a variáveis acessos transacionais (acessos as variáveis gerenciados pela STM), a não ser que consiga provar que os acessos são seguros, ou que o programador deve indicar que variáveis terão acesso transacional, o que ele alega ser um peso muito grande para o programador.

Clojure segue a opção de que o programador deve marcar todas variáveis que terão acesso transacional com o uso de *refs*, que serão explicadas na próxima seção, porém isso não se torna um peso devido ao uso forte de programação funcional, o que garante que as variáveis não transacionais não causem conflitos, devido a sua imutabilidade. Para poder tornar isso uma alternativa viável na prática, é essencial uma biblioteca padrão com estruturas de dados puramente funcionais, o que Clojure prove.

3.2 STM em Clojure

A STM de Clojure utiliza o método de controle de concorrência multiversionado (*MVCC*) (Hickey 2012). Quando uma *thread* começa uma transação, ela ganha sua própria versão do mundo, na qual ela é a única *thread* existente.

O código em Clojure é em geral funcional. Todas variáveis locais são imutáveis. Uma *ref* aponta para um valor imutável e pode apontar para valores diferentes com o passar do tempo. O importante é que somente a *ref* muda, os valores apontados por ela devem ser imutáveis. Os valores dos tipos primitivos (*int, long, double, char, string*) são imutáveis assim como as estruturas de dados puramente funcionais vistas no capítulo 2.

No exemplo da Figura 3.2 é possível ver como STM é usada em Clojure. O exemplo programa a mesma funcionalidade que o exemplo de *locks* em Java, um sistema concorrente de contas bancárias. A função *make-account* na linha 1 recebe um valor e cria uma *ref* que aponta para ele. As funções *debit* e *credit* nas linhas 4 e 8 recebem uma conta (uma *ref* que aponta para um inteiro) e um valor, e com o uso de *ref-set* alteram o conteúdo da conta. A função *transfer* nas linhas 12 a 18 faz uma transferência entre duas contas caso a conta a ser debitada possua créditos suficientes. Sua implementação é feita com a composição das funções *debit* e *credit* sendo feita de forma atômica. Não importa a ordem de operações executadas, uma *thread* que estivesse observando duas contas que estão fazendo uma transferência em outra *thread* sempre teria uma visão consistente, ou seja, a soma do saldo das duas contas sempre seria constante. O código para debitar e creditar contas continua simples como o código do exemplo em Java da Figura 3.1,

a maior diferença sendo que em Clojure não é necessário escrever código para previnir *deadlocks* na função *transfer*.

```
(defn make-account [money]
  (ref money))

(defn debit [account money]
  (dosync
    (ref-set account (- @account money))))

(defn credit [account money]
  (dosync
    (ref-set account (+ @account money))))

(defn transfer [from-account to-account money]
  (dosync
    (if (>= @from-account money)
      (do
        (debit from-account money)
        (credit to-account money))
      (throw (Exception.)))))
```

Figura 3.2: Exemplo de uso da STM em Clojure.

A modificação de uma *ref* só pode ser feita dentro de uma transação, que é iniciada com o comando *dosync*. Todo código envolto em um bloco *dosync* é executado de forma atômica, ou seja, todas as leituras e escritas feitas pelo código podem ser vistas como acontecendo em um único instante de tempo. Caso seja feita a tentativa de modificar uma *ref* fora de uma transação uma exceção é gerada e a *ref* não é modificada.

Diferentes *threads* podem ter acesso compartilhado a mesma conta. Enquanto uma *thread* executa uma transação, mudanças em *refs* feitas em outras *threads* não são visíveis.

Quando chega o final da transação, o sistema verifica se não há conflitos entre as mudanças que a *thread* fez com mudanças feitas por outras *threads* desde o início da transação. Se tudo estiver bem, suas mudanças locais se tornam os novos valores globais para as *refs*. Caso ocorra um conflito, todas mudanças feitas são descartadas e a transação é refeita.

O exemplo apresentado de contas mostra as vantagens da STM de Clojure quando comparando com o uso de *locks* em Java. O código em Clojure da Figura 3.2 é muito mais simples do que o código em Java da Figura 3.1. Observe que a função *transfer* gera uma exceção caso não exista créditos suficientes na conta a ser debitada. Mas o que acontece caso decidirmos mudar o problema para que ao invés de lançar uma exceção caso não existam fundos para a transferência nós decidirmos bloquear a *thread*? O capítulo seguinte abordará essa mudança, e para torná-la possível em Clojure será necessário introduzir uma nova função a STM, o que é a proposta principal deste trabalho.

4 RETRY TRANSACTION

O capítulo anterior apresentou o sistema de STM, que permite a escrita de código concorrente que pode ser facilmente composto. Neste capítulo será analisado um problema que ocorre ocasionalmente quando é escrito código paralelo, a necessidade de uma *thread* ter que bloquear até que uma condição ocorra e será visto como esse problema é tratado se utilizando *locks*. Será relatado que com o uso de STM em Clojure normalmente esse problema se torna impossível de solucionar. Será analisado como Concurrent Haskell lidou com esse problema, e será criada uma nova função, *retry-transaction* para viabilizar a solução do problema em Clojure.

4.1 Motivação

Um problema da estratégia atual da STM de Clojure é que não é possível escrever código condicional bloqueante. Utilizando o exemplo anterior das contas, digamos que queremos um método para realizar uma transferência entre contas, e caso não exista fundos suficientes na conta debitada, ao invés de indicar um erro quisermos esperar até que os fundos estejam presentes. Em Java, é possível utilizar *Conditions* para implementar esse código, como mostra a Figura 4.1, que entre as linhas 15 e 21 utiliza o método *await* de uma *Condition* para bloquear até que seja creditado mais dinheiro na conta.

Usando a STM de Clojure, não há como fazer isso facilmente. Um modo seria utilizar um *loop* que repetisse o código de transferência apresentado anteriormente na Figura 3.2 até obter um retorno com sucesso, uma forma de espera ativa, mas isso desperdiçaria recursos do sistema como CPU em um trabalho ocioso. O ideal seria ter uma forma de fazer a *thread* atual bloquear no caso de uma condição ser falsa até que ocorra uma mudança que possibilitaria a condição se tornar verdadeira.

(Harris et al. 2005) definiu um sistema de STM para a linguagem Concurrent Haskell, e apresentou uma nova forma de escrever código bloqueante utilizando a função *retry*. O significado de seu uso é: bloqueie até que alguma das *TVars* (equivalente em Haskell das *refs* de Clojure) lidas por essa transação até sua chamada seja modificada.

A Figura 4.2 demonstra na linha 11 o uso da função em Haskell para solucionar o novo problema de uma transferência que bloqueia caso não exista crédito suficiente.

```
void blockingTransfer(Account toAccount, int amount) {
        SortableReentrantLock locks[] = new SortableReentrantLock[2];
        locks[0] = this.lock;
        locks[1] = toAccount.lock;
        Arrays.sort(locks);
        boolean done = false;
        while (!done) {
                for (SortableReentrantLock l : locks)
                        l.lock();
                if (money >= amount) {
                        debit(amount);
                        toAccount.credit(amount);
                        done = true;
                }
                else {
                        toAccount.lock.unlock();
                        try {
                                moreMoney.await();
                        } catch (InterruptedException e) {
                        }
                        this.lock.unlock();
                }
        }
        this.lock.unlock();
        toAccount.lock.unlock();
}
```

Figura 4.1: Exemplo de código com locks em Java que bloqueia.

```
type Account = TVar Int
debit :: Account -> Int -> STM ()
debit account money = do { writeTVar account readTVar account - money}

credit :: Account -> Int -> STM ()
credit account money = do { writeTVar account readTVar account + money}

blockingtransfer :: Account -> Account -> Int -> STM ()
blockingtransfer fromAccount toAccount money =
        do { v <- readTVar account
           ; if v < money then retry
                 else debit fromAccount money; credit toAccount money }
```

Figura 4.2: Exemplo de código em Haskell que bloqueia.

4.2 Solução Proposta - retry-transaction

Este trabalho propõe disponibilizar em Clojure uma nova primitiva originalmente definida para Haskell em (Harris et al. 2005), denominada neste trabalho de **retry-transaction**. A função *retry-transaction* bloqueia a *thread* que a chama até que uma das *refs* lidas pela transação, que a *thread* está executando, seja modificada, e então reinicia a transação. Essa função é uma função primitiva pois ela não pode ser construída em Clojure em cima do sistema de STM atual, mas o sistema de STM em si precisa ser modificado para suportar a função.

O problema apresentado no início desse capítulo, de fazer uma transferência entre duas contas e bloquear caso não existam fundos suficientes se torna possível de ser resolvido em Clojure com o uso da nova função *retry-transaction*. A Figura 4.3 mostra a solução.

```
1 (defn blocking-transfer [from-account to-account money]
2   (dosync
3     (if (>= @from-account money 0)
4       (do
5         (debit from-account money)
6         (credit to-account money))
7       (retry-transaction))))
```

Figura 4.3: Exemplo de código em Clojure que bloqueia.

Caso uma *thread* chame a função *blocking-transfer* e não exista créditos suficientes em *from-account*, a função *retry-transaction* é chamada, como mostra a linha 7, e a *thread* é bloqueada. Quando outra *thread* modificar a conta *from-account*, a *thread* bloqueada irá acordar, e irá repetir a transação, agora com o novo valor em *from-account*. Se dessa vez existirem créditos suficientes, as contas são debitas e creditadas, e a transação irá terminar.

O interessante é que a capacidade de composição de transações continua presente mesmo com o uso de *retry-transaction*. Se surgir um novo problema que precise realizar atomicamente transferências entre dois pares de contas, bloqueando caso qualquer uma das transferências não seja possível, pode-se chamar a função de transferência duas vezes dentro de um bloco *dosync*, como mostra a Figura 4.4.

```
1 (dosync
2   (blocking-transfer accountA accountB amount1)
3   (blocking-transfer accountC accountD amount2))
```

Figura 4.4: Exemplo de código em Clojure mostrando composição de bloqueios.

No próximo capítulo serão apresentados experimentos para demonstrar a flexibilidade que essa função provê, e para validar seu funcionamento.

5 EXPERIMENTOS

O capítulo anterior apresentou o problema de realizar um bloqueio condicional, e a função *retry-transaction* para solucionar este problema na linguagem Clojure.

Neste capítulo são apresentados experimentos com 2 objetivos: demonstrar a flexibilidade para desenvolver novas funções utilizando a função *retry-transaction*, e verificar o funcionamento da função *retry-transaction*. A primeira seção deste capítulo apresenta uma biblioteca de canais de comunicação desenvolvida com o uso de *retry-transaction*, e a segunda seção apresenta dois problemas clássicos de programação concorrente para testar a função *retry-transaction*.

5.1 Experimento - Biblioteca para comunicação

Para demonstrar o uso e as possibilidades que essa abordagem permite, foi construída uma pequena biblioteca de comunicação assíncrona entre *threads* via canais, emprestados da linguagem Go (Google Inc. 2012) e baseados em *Communicating Sequential Processes* (CSP) (Hoare 1978).

A biblioteca define um protocolo para canais e programa dois tipos de canais: os com *buffer* de capacidade máxima limitada e os de *buffer* com capacidade máxima ilimitada. A diferença entre eles é que canais de capacidade ilimitada nunca ficam cheios.

O interessante dessa biblioteca é que a implementação de um comportamento tão complexo quanto o de canais se torna simples. Internamente, cada canal mantém uma *ref* que aponta para uma fila persistente, e só precisa suportar as seguintes operações do protocolo:

- Atomicamente realizar um *send*, adicionando o objeto a ser enviado na fila e chamando *retry-transaction* caso não seja possível.
- Atomicamente realizar um *receive*, removendo um objeto da fila e chamando *retry-transaction* caso não seja possível.
- Indicar se o canal está cheio ou vazio (*is-full?* e *is-empty?*).

A Figura 5.1 demonstra o código da biblioteca que implementa canais de capacidade limitada. O tipo *BoundedChannel* é definido para representar os canais. Ele possui dois membros, uma *ref* que aponta para uma fila e um valor *n* que define sua capacidade. O tipo *BoundedChannel* implementa o protocolo *Channel*, que precisa das funções *send, receive, is-empty?* e *is-full*. Repare que na implementação de *send* (linhas 7 a 10) e *receive* (linhas 11 a 17) é utilizado uma função auxiliar *retry-if*, que é definida nas linhas 1 a 3 com a função *retry-transaction*.

```
1  (defn retry-if [condition]
2    (if condition
3      (retry-transaction)))
4
5  (deftype BoundedChannel [queue n]
6    Channel
7    (send [this obj]
8          (dosync
9            (retry-if (>= (count @queue) n))
10           (ref-set queue (conj @queue obj))))
11   (receive [this]
12            (dosync
13              (retry-if (empty? @queue))
14              (let
15                [ret (peek @queue)]
16                (alter queue pop)
17                ret)))
18   (is-empty? [this]
19              (empty? @queue))
20   (is-full? [this]
21              (>= (count @queue) n)))
```

Figura 5.1: Implementação de canais de capacidade máxima limitada.

As possíveis operações que o usuário de um canal pode usar são:

- *send*: Enviar um objeto por um canal, bloquear se o canal estiver cheio.
- *receive*: Receber um objeto de um canal, bloquear se o canal estiver vazio.

A Figura 5.2 mostra o uso das operações *send* e *receive*. A macro *future* utilizada nas linhas 13 e 14 cria duas *threads*, que irão se comunicar com o canal criado com a função *make-channel* na linha 12. O valor 10 passado de parâmetro para a função *make-channel* indica que o canal a ser criado deve ter capacidade máxima do seu *buffer* de 10 elementos. Uma das *threads* criadas executa a função *send-n* (linhas 8 a 10), que utiliza a macro *doseq* provida pela linguagem para iterar pelos números naturais e chama a função *send* para enviar o número atual pelo canal. A outra *thread* executa a função *channel-printer* definida na linha 4, recebe os números enviados pelo canal com a função *receive*, imprime eles na tela com a função *prn* (linha 5) e se chama recursivamente com o uso de *recur* (linha 6).

A biblioteca também provê a macro *select* para dado um conjunto de operações possíveis sobre canais escolher uma para prosseguir. Ela possui uma sintaxe parecida com uma expressão *switch* de outras linguagens. Seu comportamento especificamente: É executado o código do lado esquerdo de todo *select*. Atomicamente é feito: Das operações que são capazes de executar imediatamente (*receives* no qual já há elementos no *buffer*, ou *sends* para canais ilimitados ou com espaço no *buffer*) é selecionada uma aleatoriamente. Se não houver nenhuma operação pronta para ser executada, e houver uma cláusula *default*, essa é selecionada. Se não, o programa bloqueia com o uso da função *retry-transaction* até que uma cláusula se torne pronta para ser executada. A operação selecionada é executada, assim como o código que a segue, com este fora do bloco atômico (caso se queira que ele seja executado atomicamente junto com a seleção basta colocar um *dosync* ao redor do *select*).

```
;Uma thread gera naturais de 0 a 199 e envia-os pelo canal
;A outra consome do canal, e imprime-os na tela.

(defn channel-printer [channel]
  (prn (receive channel))
  (recur channel))

(defn send-n [channel n]
  (doseq [i (range n)]
    (send channel i)))

(def a (make-channel 10))
(def p (future (channel-printer a)))
(def s (future (send-n a 200)))
```

Figura 5.2: Exemplo de uso da biblioteca de canais.

A Figura 5.3 mostra o uso da operação *select* sem uma cláusula *default*. Duas *threads* são criadas, uma delas deve fazer uma computação possivelmente muito demorada, representada no exemplo como um *sleep* de 15 segundos e envia seu resultado por um canal (linhas 4 a 8). A outra *thread* espera 10 segundos e envia um sinal de *timeout* por um canal (linhas 10 a 12). A *thread* original do programa funciona como consumidor (linhas 14 a 18) e utiliza a macro *select* para fazer um *receive* na primeira operação que tiver sucesso.

```
(def long-operation-channel (make-channel))
(def timeout-channel (make-channel))

(future
  ;an operation that takes too long
  ;...
  (Thread/sleep 15000)
  (send long-operation-channel [1 2]))

(future
  (Thread/sleep 5000) ;wait 5 seconds then timeout
  (send timeout-channel :timeout))

(select
  [:letreceive [x y] long-operation-channel
   (println "Result was " (+ x y))]
  [:receive timeout-channel
   (println "Operation timeouted")])
```

Figura 5.3: Exemplo de uso do *select* da biblioteca de canais.

A Figura 5.4 mostra uma possível implementação de uma versão alternativa de *receive* que jamais bloqueia. A função *non-blocking-receive* recebe dois valores, um canal e um valor para retornar caso o canal esteja vazio (se for chamada com apenas o canal utiliza *nil* como o valor a ser retornado caso o canal esteja vazio).

```
;Usando select para fazer um receive que nunca bloqueia

(defn non-blocking-receive
  ([channel]
    (non-blocking-receive channel nil))
  ([channel if-empty]
    (select
      [:letreceive ret channel
        ret]
      [:default
        if-empty])))
```

Figura 5.4: Exemplo de uso de *default* da biblioteca de canais.

A operação *select* apenas precisa compor seu comportamento em cima das primitivas acima, com garantia de que a composição dessas primitivas também será atômica graças a STM. De fato, a parte mais difícil do código da bibloteca é o macro para tornar o uso do *select* mais legível, já o código para tratar do possível paralelismo da operação é trivial.

O uso de STM para a implementação dessa biblioteca em Clojure possibilita algo que não é possível em Go, a composição de operações concorrentes. A Figura 5.5 mostra múltiplas operações feitas em um único momento do tempo.

```
;atomically takes 2 items of a and send them to b
(dosync
  (send b (+ (receive a) (receive a))))
```

Figura 5.5: Exemplo de múltiplas operações sendo compostas.

5.2 Experimento - Problemas clássicos

Foram feitos dois testes para validar o funcionamento da função *retry-transaction*. Eles foram o problema do jantar dos cinco filósofos com sua formulação atual baseada em (Hoare 1978) e o problema do Papai Noel, definido por (Trono 1994). Ambos são problemas de concorrência que visam atacar possíveis falhas em algoritmos concorrentes.

5.2.1 O Jantar dos Filósofos

O problema dos filósofos é o seguinte. Cinco filósofos estão sentados ao redor de uma mesa circular, cada um com um prato de comida a sua frente. Cinco garfos estão dispostos na mesa, com cada filósofo tendo um garfo a sua esquerda e um garfo a sua direita. Cada filósofo alterna entra comer e pensar. Um filósofo só pode comer se ele estiver segurando o garfo a sua esquerda e o garfo a sua direita. Os filósofos não podem se comunicar entre si. O problema é construir um algoritmo para determinar o comportamento de cada filósofo no qual não ocorra inanição, ou seja, que os filósofos consigam alterar entre comer e pensar para sempre.

A solução testada utiliza uma *thread* para cada filósofo, com cada garfo sendo representado por uma *ref* que aponta para um valor booleano que define se o garfo está livre ou sendo utilizado. O algoritmo de cada filósofo é simples, tentar atomicamente pegar o garfo a esquerda e depois o a direita, e caso um dos garfos não esteja presente chamar a função *retry-transaction*. O sistema da STM garante que não haverá *deadlocks*.

A Figura 5.6 demonstra o código que cada filósofo executa. É importante reparar na linha 4 o uso da função *retry-transaction* que permite que a *thread* de um filósofo bloqueie até que um garfo seja devolvido à mesa.

```
1  (defn pick-fork [n]
2    (dosync
3      (if-not @(forks n)
4        (retry-transaction))
5      (ref-set (forks n)
6               false)))
7
8  (defn drop-fork [n]
9    (dosync
10     (ref-set (forks n)
11              true)))
12
13 (defn eat [status id]
14   (Thread/sleep (* (Math/random) 5000))
15   (dosync
16           (drop-fork (left-fork id))
17           (drop-fork (right-fork id))
18           (ref-set (philosophers-status id) :thinking))
19   (send-off (philosophers id) think id)
20   nil)
21
22 (defn think [status id]
23   (Thread/sleep (* (Math/random) 5000))
24   (dosync
25     (pick-fork (left-fork id))
26     (pick-fork (right-fork id)))
27   (send-off (philosophers id) eat id)
28   nil)
```

Figura 5.6: Código de cada filósofo.

5.2.2 Papai Noel

O problema do Papai Noel é o seguinte. Nove renas de Papai Noel alternam entre tirar férias e ir ao polo norte para entregar presentes junto com Papai Noel. Dez duendes alternam entre trabalhar na fábrica e se reunir com Papai Noel para resolver dúvidas. Papai Noel alterna entre entregar presentes com todas as nove renas, auxiliar grupos de três duendes com suas perguntas, e caso não haja nem nove renas esperando por ele nem três ou mais duendes esperando por ele, dormindo. Caso tanto as renas quanto um grupo de duendes estejam esperando por Papai Noel simultaneamente, Papai Noel deve dar prioridade as renas.

A solução testada utiliza uma *thread* para Papai Noel, uma para cada rena e uma para cada duende. Quando *threads* de renas e duendes estão esperando para serem atendindas elas utilizam a função *retry-transaction* para bloquear até que a *thread* de Papai Noel as libere. A *thread* de Papai Noel consulta a sua fila de espera para decidir quem atender, e caso não tenha nenhum grupo pronto, chama *retry-transaction* para bloquear até que ocorra uma mudança.

5.3 Resultados

A função *retry-transaction* se mostrou flexível para desenvolver novas formas de comunicação entre *threads*. O experimento com a biblioteca de canais comprova que é possível definir operações de comportamento complexo de forma simples.

O experimento de testar problemas clássicos também se mostrou valioso. Ao se executar os testes foi detectado um erro na implementação da função *retry-transaction* com uma exceção de acesso a um ponteiro nulo sendo gerada ocasionalmente. A causa foi rastreada para uma condição de corrida entre o código de *retry-transaction* e o código original de Clojure. A implementação foi modificada para que esse bug não ocorra mais.

Na forma atual do código, os testes foram executados por mais de uma hora e nenhuma *thread* entrou em *deadlock* e nenhuma invariante do problema foi violada. Os testes foram realizados em uma maquina com quatro CPUs, com paralelismo real entre as diferentes *threads*.

6 IMPLEMENTAÇÃO

Este capítulo visa apresentar como foi feita a implementação da função *retry-transaction*, e com esse intuito primeiro apresenta alguns detalhes da implementação da linguagem Clojure, principalmente de seu sistema de STM, que são necessários para a implementação da função *retry-transaction*.

6.1 Clojure

Como este trabalho modifica o funcionamento da STM de Clojure, foi necessário primeiro entender como ela é implementada, para reutilizar seus mecanismos para implementar a função *retry-transaction*. O principal para esse trabalho foi entender a forma com que a STM de Clojure realiza um *rollback* de uma transação em caso de um conflito, já que a função *retry-transaction* realiza um *rollback*.

O código fonte da linguagem Clojure está disponível em https://github.com/clojure/clojure/ e pode ser adquirido através da ferramenta de controle de código fonte *git*. Ele pode ser compilado com as ferramentas *ant* ou *maven*.

O compilador da linguagem Clojure é escrito em Clojure e Java. Em Java estão programados o compilador, as estruturas de dados que a linguagem prôve, e as facilidades para programação concorrente (entre elas o sistema de STM). O código em Java contém 116 classes. Em Clojure estão programados a biblioteca padrão, as funções que interajem com a parte programada em Java, e diversas macros necessárias para utilizar a linguagem. O código em Clojure contém 63 arquivos, sendo que o mais importante é o arquivo *core.clj* que realiza o *bootstrap* de diversas facilidades da linguagem.

A STM de Clojure está contida principalmente na parte escrita em Java. Suas principais classes são *LockingTransaction* e *Ref*. A classe *LockingTransaction* é responsável para gerenciar as transações da STM, e a classe *Ref* para prover as *refs*. Fazem parte ainda do sistema de STM as funções e macros definidas em Clojure para interagir com essas classes, que se encontram no arquivo *core.clj*.

6.2 Implementação de retry-transaction

Toda transação começa com o uso da macro *dosync* ao redor de um código cliente que deve ser transacionado. O macro utiliza a interface de comunicação entre Clojure e Java para chamar o método estático *runInTransaction* da classe *LockingTransaction*, passando de parâmetro o código que deve ser transacionado envolto em uma função anônima.

O método *runInTransaction* verifica se já está sendo executada uma transação, nesse caso apenas precisa chamar a função anônima com o código cliente. Caso não exista uma

transação, é criada uma instância da classe *LockingTransaction* e chamado o método *run*.

O método *run* é o mais importante do sistema da STM. Ele contém um laço, que repete até que a transação atual consiga realizar um *commit* com sucesso. Dentro do laço, é realizado um *try*, e o código cliente é executado se chamando a função anônima que o contém. Após a execução do código cliente, é feita a tentativa de cometer a transação.

Conflitos podem ser detectados em dois lugares, na tentativa de cometer, ou durante a execução do código cliente. Se durante a execução do código cliente for detectado um conflito (exemplo: estamos fazendo um *ref-set* em uma *ref* cujo valor já foi modificado) é gerada uma exceção do tipo *RetryEx*. O bloco *try* do método *run* captura essas exceções geradas por conflitos, e utiliza o laço para tentar a transação novamente. Um campo da transação chamado de *info* guarda o *status* da transação, que pode ser *RUNNING, COMMITTING, RETRY, KILLED ou COMMITTED*.

Para programar a função *retry-transaction* é necessário ainda uma forma de notificação caso uma *ref* seja modificada. Clojure prove essa forma através de *watchers*, funções que são atreladas a uma *ref* e executadas logo após a fase de *commit* quando o valor da *ref* é modificado.

Essa é uma visão superficial da implementação da STM de Clojure, mas suficiente para poder ser implementada a função *retry-transaction*. Foi modificado o método de leitura de uma *ref* dentro de uma transação para podermos rastrear todas as *refs* que foram lidas. Quando a função *retry-transaction* é chamada, ela lança uma exceção do tipo *BlockRetryException*. O método *run* da classe *LockingTransaction* foi modificado para incluir um *catch* para a nossa exceção. O *status* da transação, guardado no campo *info*, é modificado para um novo estado criado por este trabalho, *BLOCKED*. É utilizado um bloco *synchronized* no campo *info* da transação atual, pois utilizaremos seu monitor para podermos bloquear. É adicionado um *watcher*, *WakeWhenChangeFn*, a cada *ref* que foi lida durante a transação. Enquanto o *status* continua sendo *BLOCKED*, bloqueamos a thread atual chamando o método *wait()* no campo *info*, que ao mesmo tempo bloqueia e libera o monitor que foi adquirido com o uso do *syncronized*. Quando o status for modificado, removemos os *watchers* que adicionamos, e utilizamos do laço do método run para tentarmos o código cliente novamente.

Quando uma *ref* lida for modificada por uma outra transação, a função *watcher* criada, *WakeWhenChangeFn*, será executada. *WakeWhenChangeFn* obtém o monitor do campo *info* da transação bloqueada, verifica que o status da transação é *BLOCKED*, modifica ele para *RUNNING* e notifica a transação bloqueada chamando o método *notify* do *info*. O uso de monitores garante acesso exclusivo sobre a função, evitando a condição de corrida de duas threads chamarem o *watcher* ao mesmo tempo.

6.3 Limitações

Como o código do início da transação até o momento que a transação bloqueou precisa ser re-executado quando ela acorda, é aconselhável manter as transações pequenas. Como todas as *refs* lidas por uma *thread* bloqueada podem reativar a *thread* quando modificadas, é aconselhável somente ler *refs* que conduzem ao progresso do programa antes de chamar *retry-transaction* (exemplo: um programa lê dentro de um *dosync* as *refs* *a*, *b* e *c*, e caso *c* seja maior que 50 o programa chama *retry-transaction*. Se outras *threads* modificarem *a* ou *b*, o programa tentara novamente a transação, que novamente irá falhar, desperdiçando tempo do processador).

7 CONCLUSÕES

7.1 Análise do trabalho

O código para implementar a nova função na STM de Clojure foi implementado e testado com sucesso. Sua corretude não está garantida, já que ele opera com primitivas de concorrência de um nível mais baixo, e é possível que devido a bugs ele apresente condições de corrida ou *deadlocks*.

O uso da função *retry-transaction* parece promissor, com iterações que são complexas de serem realizadas com monitores ou *watchers* se tornando mais simples. A facilidade de compor operações provida pela STM continua presente mesmo quando *retry-transaction* é utilizado múltiplas vezes.

7.2 Melhorias futuras

- Implementar o operador *OrElse* (Harris et al. 2005). O operador recebe duas transações, executa a primeira transação, e caso ela chame a função *retry*, ao invés de bloquear o operador executa a segunda transação. Se está também chama *retry*, a *thread* bloqueia até que *refs* de qualquer uma das transações sejam alteradas. Implementar esse operador *OrElse* necessitaria de mudanças significativas na STM de Clojure, pois durante seu uso é possível de ocorrer o *rollback* de transações internas a uma transação. O código atual de Clojure pode ignorar *dosyncs* internos de uma transação, o que impossibilita *rollbacks* internos.

- Testar desempenho do código. Deve ser feito com cuidado, devido a se tratar de código para lidar com paralelismo, além de ser muito fácil de cometer erros ao se realizar *benchmarks* em Java (Goetz 2005).

7.3 Reflexões sobre Memória Transacional

Memória transacional é um assunto na qual muita pesquisa está sendo feita atualmente. É possível traçar analogias entre STM e *garbage collectors* (Grossman 2007). Talvez sua adoção pelas linguagens mais populares demore, e assim como até hoje existem aplicações onde *garbage collection* não é viável, é possível que sempre continue a existir aplicações no qual o uso de STM não seja viável.

REFERÊNCIAS

[Adl-Tabatabai et al. 2006]ADL-TABATABAI, A.-R. et al. Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 41, n. 6, p. 26–37, jun. 2006. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1133255.1133985>.

[Bagwell 2001]BAGWELL, P. Ideal hash trees. *Es Grands Champs*, v. 1195, 2001.

[Cascaval et al. 2008]CASCAVAL, C. et al. Software transactional memory: Why is it only a research toy? *Queue*, ACM, New York, NY, USA, v. 6, n. 5, p. 46–58, set. 2008. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/1454456.1454466>.

[Charles et al. 2005]CHARLES, P. et al. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 10, p. 519–538, out. 2005. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1103845.1094852>.

[Driscoll et al. 1986]DRISCOLL, J. R. et al. Making data structures persistent. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1986. (STOC '86), p. 109–121. ISBN 0-89791-193-8. Disponível em: <http://doi.acm.org/10.1145/12130.12142>.

[GitHub Inc. 2012]GitHub Inc. *Clojure*. maio 2012. Disponível em: <https://github.com/languages/Clojure>.

[Goetz 2005]GOETZ, B. *Java theory and practice: Anatomy of a flawed microbenchmark*. 2005.

[Google Inc. 2012]Google Inc. *The Go Programming Language Specification*. mar. 2012. Disponível em: <http://golang.org/ref/spec>.

[Grossman 2007]GROSSMAN, D. The transactional memory / garbage collection analogy. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 42, n. 10, p. 695–706, out. 2007. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1297105.1297080>.

[Harris et al. 2005]HARRIS, T. et al. Composable memory transactions. In: . [S.l.]: ACM Press, 2005. p. 48–60.

[Hickey 2012]HICKEY, R. *Clojure's Website*. 2012. Disponível em: <http://clojure.org/>.

[Hickey 2012]HICKEY, R. *Refs and Transactions*. 2012. Disponível em: <http://clojure.org/refs>.

[Hickey 2012]HICKEY, R. *Transient Data Structures*. 2012. Disponível em: <http://clojure.org/transients>.

[Hoare 1978]HOARE, C. A. R. Communicating sequential processes. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 8, p. 666–677, ago. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359576.359585>.

[Jones et al. 1996]JONES, S. P. et al. Concurrent haskell. In: *Annual Symposium on Principles of Programming Languages*. [S.l.]: ACM, 1996. p. 295–308.

[McCarthy 1960]MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, ACM, New York, NY, USA, v. 3, n. 4, p. 184–195, abr. 1960. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/367177.367199>.

[Raymond 2000]RAYMOND, E. S. *Homesteading the Noosphere*. 2000. Disponível em: <http://catb.org/ esr/writings/homesteading/homesteading/index.html>.

[Redinger 2012]REDINGER, C. *Clojure Success Stories*. 2012. Disponível em: <http://dev.clojure.org/display/community/Clojure+Success+Stories>.

[Shavit e Touitou 1995]SHAVIT, N.; TOUITOU, D. Software transactional memory. In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1995. (PODC '95), p. 204–213. ISBN 0-89791-710-3. Disponível em: <http://doi.acm.org/10.1145/224964.224987>.

[Steele Jr. e Gabriel 1993]STEELE JR., G. L.; GABRIEL, R. P. The evolution of lisp. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 28, n. 3, p. 231–270, mar. 1993. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/155360.155373>.

[Trono 1994]TRONO, J. A. A new exercise in concurrency. *SIGCSE Bull.*, ACM, New York, NY, USA, v. 26, n. 3, p. 8–10, set. 1994. ISSN 0097-8418. Disponível em: <http://doi.acm.org/10.1145/187387.187391>.