PROPOSTA DE COMMIT E ABORT HANDLERS PARA STM HASKELL

JONATHAS AUGUSTO DE OLIVEIRA CONCEIÇÃO[1]; ANDRÉ RAUBER DU BOIS[2]

[1]*Universidade Federal de Pelotas – jadoliveira@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas – dubois@inf.ufpel.edu.br*

1. INTRODUÇÃO

Software Transactional Memory (STM) é uma alternativa de alto nível ao sistema de sincronização por *locks*. Nela todo acesso à memória compartilhada é agrupado como transações que podem executar de maneira concorrente. Se não houve conflito no acesso à memória compartilhada, ao fim da transação um *commit* é feito, tornando assim o conteúdo do endereço de memória público para o sistema. Caso ocorra algum conflito um *abort* é executado descartando qualquer alteração ao conteúdo da memória e a transação é recomeçada. Diferente da sincronização por *locks*, transações podem ser facilmente compostas e são livres de *deadlocks* [Harris at al. 2008].

Memórias Transacionais funcionam através da criação de blocos atômicos onde alterações de dados são registradas para detecção de conflitos. Os conflitos ocorrem quando duas ou mais ações de escrita ou leitura são feitas no mesmo endereço de memória; entretanto, essa forma de detecção conflitos pode, em alguns casos gerar falsos conflitos, levando a uma perda de desempenho. Um exemplo seria quando duas transações modificam partes diferentes de uma lista encadeada [Sulzmann at al. 2009, Herlihy at al. 2008]. Embora essas ações não conflitem, o sistema detecta um conflito já que uma transação modifica uma área de memória lida por outra transação. Este tipo de detecção de conflito pode ter grande impacto na performance quando se utiliza certos tipos de estruturas encadeadas. Por outro lado, utilizando sincronização por *locks*, ou mesmo algoritmos *lock-free*, programadores experientes podem alcançar um alto nível de concorrência ao custo de complexidade no código.

Funções linearmente concorrentes (locks e afins) seguem uma semântica de execução diferente de transações. Enquanto funções linearmente concorrentes executam de maneira sequencial em relação ao tempo, as funções transacionalmente concorrentes podem ser re-executadas inúmeras vezes até que sejam de fato efetivadas. Em Haskell, por essa diferença em sua natureza de execução, ações linearmente concorrentes são modeladas na mônada de IO, enquanto funções transacionalmente concorrentes são modeladas na mônada de STM. Assim, o forte sistema de tipos da linguagem garante que essas ações não possam ser livremente compostas entre si.

Apesar dessa diferença em seu modelo de execução, compor ações linearmente concorrentes aos blocos transacionais de maneira apropriada já mostrou-se ser uma maneira viável de resolver perdas de desempenho e evitar *redesign* de implementações quando se lida com blocos transacionais [Herlihy et al. 2008, Du Bois at al. 2014]. Apesar disso, a STM Haskell não oferece, seja de maneira nativa ou por meio de bibliotecas, uma maneira segura de compor ações linearmente concorrentes com ações transacionalmente concorrentes.

STM Haskell [Harris at al. 2008] é uma biblioteca do *Glasgow Haskell Compiler* (GHC) que provê primitivas para o uso de memórias transacionais em Haskell. O programador define ações transacionais que podem ser combinadas para gerar novas transações como valores de primeira ordem. O sistema de tipos da linguagem só permite acesso à memória compartilhada dentro das transações e as mesmas não podem ser executadas fora de uma chamada ao *atomically*, garantindo assim que a atomicidade (o efeito da transação se torna visível todo de uma vez) e isolamento (durante a execução, uma transação não é afetada por outra) são sempre mantidos.

O objetivo deste trabalho então é de estender a biblioteca STM Haskell com duas novas primitivas, um *commit handler* e um *abort handler*, similar à bibliotecas como a DSTM2 [Harris at al. 2006], possibilitando que o programador especifique ações para serem tomadas em caso de *commit* ou *abort*. Visamos então oferecer uma maneira segura e modular de compor ações linearmente concorrentes aos blocos transacionais.

2. METODOLOGIA

O STM Haskell define um conjunto de primitivas para utilização de Memórias Transacionais em Haskell (Figura 1). Nela o acesso a memória compartilhada é feito através de variáveis transacionais, as *TVar*s, que são variáveis acessíveis apenas dentro de transações. Ao fim da execução de um bloco transacional, um registro de acesso às *TVars* é analisado para determinar se a transação foi bem-sucedida ou não, para assim realizar o *commit* ou *abort* da transação.

Para lidar com as *TVars* temos três primitivas principais. (1) **newTVar** é utilizado para criar uma *TVar* que pode conter valores de um tipo **a** qualquer; (2) **readTVar** retorna o conteúdo da *TVar*; e (3) **writeTVar** que escreve um valor **a** em uma *TVar* de mesmo tipo. As transações acontecem dentro da mônada STM e essas ações podem ser compostas para gerar novas ações através dos operadores monádicos (*bind* (**>>=**), *then* (**>>**), e **return**), ou com a utilização da notação **do**.

```
-- Execution control
atomically :: STM a -> IO ()
retry :: STM a
orElse :: STM a -> STM a -> STM a

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

Figura 1. Interface do STM Haskell.

O **retry** e o **orElse** são primitivas que controlam a execução do bloco. **retry** é utilizada para abortar uma transação e colocá-la em espera até que alguma de suas *TVars* seja alterada por outra transação. **orElse** é uma primitiva de composição alternativa, ela recebe duas ações transacionais e apenas uma será considerada; se a primeira ação chamar **retry** ela é abandonada, sem efeito, e a segunda ação transacional é executada; se a segunda ação também chamar **retry** todo o bloco é re-executado.

As novas primitivas sendo desenvolvidas podem ser vistas na Figura 2. Com essas primitivas o programador poderá definir *handlers* (do tipo **IO ()**) para serem executados. No momento que a primitiva é chamada a ação é posta na respectiva pilha de *commit handlers* ou de *abort handlers.* Ao fim da transação ações extras são tomadas para o tratamento dos handlers; em caso de *commit*, as ações associadas às pilha de *commit handlers* são desempilhadas e executadas uma a uma; em caso de *abort*, os *handlers* são desempilhados e executadas e, por fim, as ações de *commit* associadas a transação atual são descartados antes de que a transação volte a ser re-executada.

```
addCommitHandler :: IO () -> STM ()
addAbortHandler :: IO () -> STM ()
```

Figura 2. Novas primitivas para handlers definidos pelo usuário.

A implementação das primitivas envolve três passos principais, (1) estender as estruturas no RunTime System do GHC para dar suporte aos novos *handlers*; (2) implementar as funções para permitir que o usuário defina novos *handlers*; (3) estender o RunTime System do GHC para que as ações extras sejam executadas quando uma transação realizar um *commit* ou um *abort*. Com as primitivas devidamente implementadas, o programador poderá designar livremente ações linearmente concorrentes para serem executadas caso a transação atual realize um *commit* ou um *abort*.

3. RESULTADOS E DISCUSSÃO

Até o presente momento o RunTime System da linguagem foi estendido para armazenar os *handlers* de *commit* e *abort*. Registros de transações agora, além de armazenar o estado atual de uma transação e a lista de *TVar*s acessadas por ela, também armazena duas pilhas de ações de *commit* e *abort*. Também foi finalizado o desenvolvimento das duas novas primitivas apresentadas na Figura 2, que podem ser utilizadas para associar novos handlers a transação atual.

Estender o RunTime System do GHC para que as transações executem os *handlers* em transações aninhadas é o próximo passo do trabalho. Em Haskell as primitivas **retry** e **orElse** são implementadas usando a ideia de *Nested Transaction* como explicado em [Le et al. 2016], onde transações podem ser aninhadas para que nem todo o bloco sofra um *abort* ou *commit*. O estado atual desta parte da implementação ainda não dá suporte apropriado ao uso de *handlers* dentro destas *Nested Transactions*.

4. CONCLUSÕES

Com a implementação descrita neste resumo e em trabalhos anteriores, espera-se compor um conjunto de primitivas para o STM Haskell que possibilitem maior controle do programador sobre a execução da transação. Embora possam adicionar mais complexidade de código e possibilitar problemas como *Deadlocks*, se utilizadas por programadores experientes, tais primitivas devem possibilitar o design de funções e estruturas transacionais mais eficientes mas ainda seguramente componíveis à outras ações transacionais.

5. REFERÊNCIAS BIBLIOGRÁFICAS

Du Bois, A., Pilla, M., Duarte, R. (2014). Transactional boosting for haskell. Quintão Pereira, F., editor, **Programming Languages, volume 8771 of Lecture Notes in Computer Science**, 145–159. Springer International Publishing.

Harris, T., Marlow, S., Jones, S. P., Herlihy, M. (2008). Composable memory transactions. **Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming**. ACM, 51(8):91–100.

Herlihy, M., Koskinen, E. (2008). Transactional boosting: A methodology for highly-concurrent transactional objects. **Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**, PPoPP ’08, 207–216, New York, NY, USA. ACM.

Herlihy, M., Luchangco, V., Moir, M. (2006). A flexible framework for implementing software transactional memory. **Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications**, 253-262, Portland, OR, USA. ACM.

Le, M., Yates, R., Fluet, M., (2016). Revisiting Software Transactional Memory in Haskell. **Proceedings of the 9th International Symposium on Haskell**, 105–113, Nara, Japan, ACM.

Sulzmann, M., Lam, E. S., and Marlow, S. (2009). Comparing the performance of concurrent linked-list implementations in haskell. **Proceedings of the 4th workshop on Declarative aspects of multicore programming**, pages 37–46. ACM.