Transformação de código CMTJava

BANDEIRA, Rafael de Leão
Universidade Federal de Pelotas

DU BOIS, André Rauber
Universidade Federal de Pelotas

1 INTRODUÇÃO

Nos últimos anos diversos fatores vem limitando o crescimento da capacidade de processamento dos microprocessadores e a tendência atual é a integração de dois ou mais núcleos de processamento em um mesmo chip. Nesta abordagem, diferentemente das arquiteturas monoprocessadas onde o aumento da capacidade de processamento implica um aumento proporcional no desempenho dos softwares utilizados, o processamento deve ser paralelizado entre os núcleos para explorar toda a capacidade da arquitetura.

Para tanto, a abordagem mais usada na programação de máquinas multi-core é a utilização de diversos fluxos de execução (*threads*), que se comunicam através de memória compartilhada e bloqueios (*locks*) para sincronização e controle de concorrência entre as *threads*. Porém, a utilização desses artifícios torna a tarefa de programação mais árdua e propensa a erros (DU BOIS, 2008; JONES, 2007).

Nesse contexto, surgiu um novo modelo para o controle de concorrência, as memórias transacionais (HERLIHY; MOSS, 1993). Esse modelo supera as dificuldades encontradas no uso de *locks* pois o acesso a memória compartilhada é feito em transações que executam de forma atômica (são executadas completamente ou não tem nenhum efeito) em relação a outras transações concorrentes.

Atualmente existe um grande interesse no assunto, devido ao fato das memórias transacionais oferecerem um alto nível de programação pois o programador apenas precisa identificar e delimitar regiões críticas mas não controlar o seu acesso, tal tarefa fica por conta do sistema transacional.

CMTJava (DU BOIS; ECHEVARRIA, 2009) é uma linguagem de domínio específico para programação de memórias transacionais em Java baseada em STM Haskell (HARRIS et al., 2005). CMTJava oferece vantagens como: abstração de objetos transacionais, composição de transações e suporte as construções *retry* e *orElse*.

A linguagem é implementada utilizando uma mônada de passagem de estados. Mônadas são uma forma de estruturar computações em termos de valores e sequências de computações usando esses valores. A mônada é usada para descrever computações que podem ser combinadas formando novas computações. Na mônada STM, há dois métodos muitos importantes: *bind* e *then*. O método *bind* é usado para compor ações transacionais e o método *then* é uma combinação sequencial: ele recebe como argumento duas ações STM e retorna uma ação que irá executá-las uma depois a outra. CMTJava utiliza BGGA *Closures*, uma extensão de Java para *closures* e regras de tradução são utilizadas para traduzir CMTJava para Java puro + *closures*.

Porém, atualmente essa tradução é feita de forma manual, sendo que mesmo para classes pequenas constitui uma tarefa cansativa e propensa a inserção de erros no código produzido, o que dificulta a utilização da linguagem.

Além da possibilidade de cometer-se erros, há outro problema. No processo de tradução manual, pode gerar-se um código sem erros sintáticos, mas semanticamente diferente do código original dada a falta de clareza característica do código dos *closures*. Esse tipo de erro é mais difícil de ser percebido e corrigido.

Este trabalho propõe o desenvolvimento de um compilador por transformação de código para automatizar a etapa de tradução da linguagem CMTJava.

2 METODOLOGIA

Para transformar o código CMTJava as seguintes etapas devem ser realizadas: análise sintática e análise léxica do código fonte em CMTJava, juntamente com a construção da árvore sintática para posterior transformação da mesma através de regras de reescrita e finalmente o *pretty-print* da árvore sintática para a linguagem alvo.

A ferramenta escolhida para auxiliar a realização desse trabalho foi o ANTLR. ANTLR (PARR; QUONG, 1994) é uma acrônimo para ANother Tool for Language Recognition, consistindo em um sofisticado gerador de *parsers* que pode ser usado para implementar interpretadores, compiladores e outros tradutores.

De uma definição formal de linguagem, ANTLR gera um programa que determina se aquela sentença pertence a linguagem. Adicionando alguns fragmentos de código à gramática, o reconhecedor torna-se um tradutor. Esses fragmentos de código computam as expressões de saída baseados nas expressões de entrada. ANTLR é adequado para o mais simples ou mais complicado reconhecimento de linguagem ou problema de tradução.

Usando o ANTLR, deve-se construir uma gramática para a linguagem CMTJava de forma que as regras dessa gramática contenham ações que construam uma árvore sintática. Como CMTJava é uma extensão da linguagem Java, a gramática não precisa ser construída do zero, podendo ser estendida partindo-se de uma gramática de Java, para reconhecer as construções inerentes a CMTJava.

Essa extensão se dá basicamente pela adição do bloco STM à gramática de Java. O blocos STM em CMTJava são uma implementação da notação *do* disponível em Haskell. A notação *STM{a1; . . . ; an}* constrói uma ação STM que une pequenas operações *a1;. . . ;an* em sequência. Os blocos STM são usados para compor transações e são traduzidos para chamadas de *bind* e *then* usando as regras de tradução mostradas na Figura 1.

```
STM{ type var <- e; s} = STMRTS.bind( e, { type var => STM { s }})

STM{ e ; s }           = STMRTS.then( e,  STM{ S })

STM{ e }               = e
```

Figura 1: Esquema básico de tradução para blocos STM

3 RESULTADOS E DISCUSSÕES

A partir de uma gramática da linguagem Java para ANTLR foi construída a árvore sintática do código fonte e, em uma etapa posterior, realizado o *pretty-print* da mesma. Para realizar traduções simples, poderia-se simplesmente emitir a saída já na fase de *parsing*. No caso de CMTJava, como a tradução é mais complicada, o analisador sintático apenas constrói a árvore sintática e não gera nenhuma saída. Com a árvore gerada, essa é passada para uma outra fase, o *parsing* da árvore (*tree parsing*). O *tree parser* construído percorre a árvore gerada na fase anterior imprimindo o código fonte nela contido.

Em resumo, o tradutor até aqui implementado recebe código Java, faz sua análise, constrói a árvore sintática e finalmente varre essa árvore gerando código semanticamente equivalente ao que foi fornecido na entrada.

Como trabalho posterior pode-se citar a extensão dessa gramática de Java para aceitar as construções da linguagem CMTJava. Além disso, será implementada a geração dos nós da árvore sintática dessas novas construções inseridas.

Posteriormente, deverá ser implementada a transformação das construções de CMTJava para Java + *closures*. Também é necessário gerar código para os atributos dos objetos transacionais (Tobject), para então utilizar-se o processo tradicional de compilação de um programa Java.

4 CONCLUSÕES

O trabalho proposto visa a elaboração de uma ferramenta capaz de automatizar a etapa de tradução do código fonte da linguagem de domínio específico CMTJava, que até então é feito de forma manual. Como foi discutido, esse processo é propenso a erros, sem falar o fato de ser extremamente enfadonho.

Com a tradução implementada via software, será possível utilizar-se de forma mais fácil e rápida a linguagem, permitindo assim que cresça a sua aceitação por parte da comunidade científica. Outra importante contribuição desse trabalho será a definição rígida de uma sintaxe para CMTJava, fato que até então não é observado, já que não é aplicada uma verificação sintática rigorosa como por exemplo a realizada por um *parser*.

5 REFERÊNCIAS

DU BOIS, A. R. Memórias Transacionais e Troca de Mensagens: duas alternativas para a programação de máquinas multi-core. **Escola Regional de Alto Desempenho**, [S.l.], p.43–76, 2008.

DU BOIS, A. R.; ECHEVARRIA, M. A Domain Specific Language for Composable Memory Transactions in Java. In: DSL ’09: PROCEEDINGS OF THE IFIP TC 2 WORKING CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES, 2009, Berlin, Heidelberg. **Anais. . .** Springer-Verlag, 2009. p.170–186.

HARRIS, T.; MARLOW, S.; PEYTON, S.; HERLIHY, J. M. Composable memory

transactions. In: 2005. **Anais. . .** ACM Press, 2005. p.48–60.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.21, n.2, p.289–300, 1993.

JONES, S. P. **Beautiful concurrency**. [S.l.]: O'Reilly Media, Inc., 2007. 385–406p.

PARR, T. J.; QUONG, R. W. ANTLR: a predicated-ll(k) parser generator. **Software Practice and Experience**, [S.l.], v.25, p.789–810, 1994.