Compilação da CMTJava por transformação de programas[†]

Rafael de Leão Bandeira[1], André Rauber Du Bois[1], Maurício Lima Pilla[1]

[1]Centro de Desenvolvimento Tecnológico (CDTEC)
Universidade Federal de Pelotas (UFPEL)
Campus Universitário, s/nº
Caixa Postal 354 – 96010-900 – Pelotas – RS– Brasil

{rdlbandeira, dubois, pilla}@inf.ufpel.edu.br

1. Introdução

Com o crescimento da demanda por desempenho na execução de programas, a utilização de processos perdeu espaço para a multiprogramação leve (*multithreading*). Essa abordagem permite que diversos fluxos de execução (*threads*) sejam instanciados no interior de um processo. A área de memória do processo é acessível por todas as *threads* ativas, sendo necessário controlar a execução de conjuntos de instruções que acessam uma área de dados compartilhada, chamados de seções críticas [Costa et al. 2002]. Um mecanismo bastante usado para implementar a sincronização entre as *threads*, são os *locks*. No entanto, a utilização desse artifício dificulta bastante a tarefa de programação, se comparado à programação sequencial.

As Memórias Transacionais [Herlihy and Moss 1993] surgiram como um novo modelo para o controle de concorrência na programação de máquinas multicore que baseia-se no conceito de transação de banco de dados: englobar várias ações em uma operação atômica. 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 em relação a outras transações concorrentes. O programador apenas precisa identificar e delimitar regiões críticas, o controle do acesso dessas fica por conta do sistema transacional.

CMTJava [Du Bois and 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. A mônada STM é composta pelos métodos *bind*, *then* e *return*. No entanto, esses métodos oferecem uma baixa abstração para programação.

Este trabalho apresenta o bloco STMDO para composição de ações transacionais em CMTJava. O bloco STMDO oferece uma abstração de mais alto nível para criação e composição de transações e é traduzido para chamadas de *bind* e *then*. Além disso, descreve a automatização do processo de análise e geração de métodos de *get* e *set* para atributos de objetos transacionais (TObject), para compilação do código CMTJava.

[†]Este projeto foi financiado com uma bolsa ITI do projeto PRONEX FAPERGS/CNPq GREEN-GRID Computação de Alto Desempenho Sustentável

2. Blocos STMDO

A mônada para ações STM é implementada como uma mônada de passagem de estados, composta por operações de baixo nível, utilizadas para criar e compor transações. Essa mônada é usada para passar um estado pelas computações, sendo que cada computação retorna uma cópia alterada desse estado, onde esse estado é uma transação. Para uma mônada qualquer `m`, essas funções tem o seguinte tipo em Haskell:

```
bind ::  m a -> (a -> m b )-> m b

then ::  m a -> m b -> m b

return ::  a -> m a
```

O tipo `m a` representa uma computação dentro da mônada `m` que quando executada produzirá um valor do tipo `a`. As funções *bind* e *then* são usadas para combinar computações em uma mônada. O método *bind* é usado para compor ações transacionais, ele executa seu primeiro argumento e passa o resultado para seu segundo argumento (uma função) produzindo uma nova computação. 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. A função *return* cria uma nova computação para um simples valor.

O bloco SMTDO é uma abstração de alto nível para criação e composição de transações. A notação $STMDO\{a_1; \ldots; a_n\}$ constrói uma ação STM que une pequenas operações $a_1; \ldots; a_n$ em sequência. Blocos STMDO são traduzidos para chamadas de *bind* e *then* usando as regras de tradução mostradas na Tabela 1. As regras de tradução dos blocos STMDO são muito similares as regras da notação *do* disponível em Haskell.

Tabela 1. Esquema de tradução dos blocos STMDO

CMTJava	Java + *closures*
`STM{ type var <- e; s}`	`STMRTS.bind(e, { type var => STM{ s }})`
`STM{ e; s }`	`STMRTS.then(e, STM{ s })`
`STM{ e }`	`e`

3. Implementação

Muitos compiladores usam a técnica de compilação por transformação de código. Nessa abordagem, o máximo possível do processo de compilação é expressado como transformações que preservam a corretude e mantém a semântica do programa [Jones et al. 1994].

Para transformar o código CMTJava em bytecodes Java, as seguintes etapas são realizadas: análise léxica e análise sintática do código fonte em CMTJava, geração dos métodos *get* e *set* para todos os atributos das classes que implementam a interface TObject e a tradução dos blocos STMDO para chamadas de *bind* e *then*. Após essa fase, o código traduzido para Java puro + *closures* é compilado usando o compilador do BGGA *closures*. Uma visão geral da compilação de um programa CMTJava pode ser vista na Figura 1.

A ferramenta escolhida para auxiliar a realização desse trabalho foi o ANTLR. ANTLR [Parr and Quong 1994] é um acrônimo para A*Nother Tool for Language*

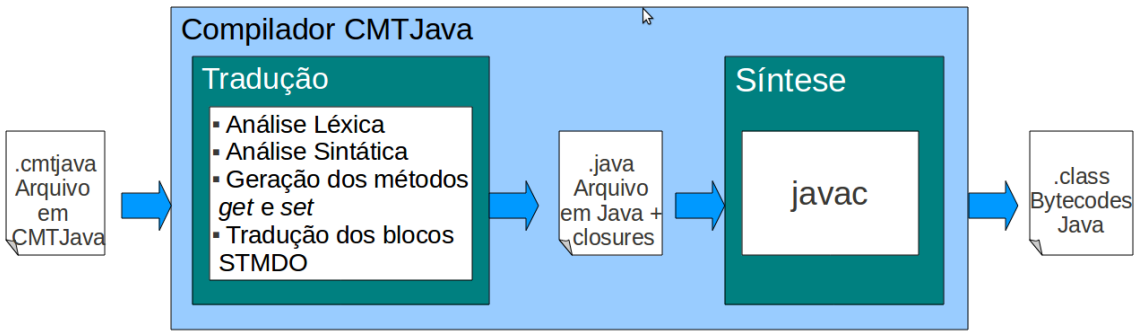

Figura 1. *Pipeline* do processo de compilação de um programa CMTJava

Recognition, consistindo em um sofisticado gerador de *parsers* que pode ser usado para implementar interpretadores, compiladores e outros tradutores. Para realizar as transformações e geração de código, foi utilizado o String Template, uma linguagem para gerar texto estruturado integrada ao ANTLR.

Usando o ANTLR foi implementada uma gramática para reconhecer as construções inerentes a CMTJava. Após reconhecer os programas na linguagem, para cada classe que implementa a interface TObject, são gerados pelo compilador métodos *get* e *set* para cada um de seus atributos, sendo este o único meio de modificar o conteúdo de tais atributos. Essa restrição garante que os atributos da classe não poderão serem acessados fora de transações, assim garantindo que as propriedades de atomicidade e isolamento das transações não sejam violadas.

A tradução dos bloco STMDO é feita recursivamente de acordo com o esquema apresentado na Tabela 1. No mecanismo de tradução implementado, basicamente as regras pertencentes aos nós mais inferiores da árvore de derivação retornam sua tradução para os nós superiores, que vão montando a tradução de acordo com as regras definidas.

O código devidamente transformado, pode então ser compilado utilizando o compilador do BGGA closures, que estende a linguagem Java com o suporte a *closures*.

4. Resultados

A Figura 2 apresenta um exemplo de código escrito em CMTJava e sua tradução. Em particular, é apresentado um método para deleção de um elemento em uma lista. Na esquerda, o código na linguagem CMTJava e na direita, o código transformado automaticamente, mantendo a semântica, para java + *closures*.

Essa transformação levaria um tempo considerável se feita a mão, principalmente por um usuário inexperiente, e também seria sucetível a erros. Porém, quando automatizado, o processo é realizado em um tempo praticamente desprezível e de forma correta. Além disso, a análise realizada sobre o código facilita a identificação e correção de erros. Após a etapa de tradução, o código é passado ao compilador do BGGA closures e então são gerados os *bytecodes* para serem executados pela máquina virtual Java.

Código em CMTJava

```
public STM<Boolean> delete(Node prevNode, Integer i) {

  STM<Boolean> r = new STMDO{
    Node curNode <- prevNode.getNext();
    if(curNode==null) {
       STMRTS.stmReturn(false)
    } else {
      Integer cv <- curNode.getVal();
      if(cv==null) {
        STMRTS.stmReturn(false)
      } else {
        if(!cv.equals(i)) {
          delete(curNode, i)
        } else {
          Node next <- curNode.getNext();
          prevNode.setNext(next);
          STMRTS.stmReturn(true)
        }
      }
     }
   };
   return r;
  }
}
```

Código após a etapa de tradução

```
public STM<Boolean> delete(Node prevNode, Integer i) {

  STM<Boolean> r = STMRTS.bind( prevNode.getNext() ,{
   Node curNode => curNode==null
    ? STMRTS.stmReturn(false)
    : STMRTS.bind( curNode.getVal() , {Integer cv =>
    cv==null ? STMRTS.stmReturn(false)
    : !cv.equals(i) ? delete(curNode, i) : STMRTS.bind(
        curNode.getNext() ,
    { Node next => STMRTS.then( prevNode.setNext(next),
    STMRTS.stmReturn(true))})})});
  return r;
  }
}
```

Figura 2. Método delete: exemplo de tradução do bloco STMDO realizada pelo compilador

5. Conclusões

Este artigo relatou a elaboração de uma ferramenta para automatizar a etapa de compilação do código fonte em CMTJava. Essa automação facilita a utilização da linguagem e garante a corretude no processo.

Como trabalho futuro, pode-se citar a realização de algumas otimizações do algoritmo usado na implementação do sistema transacional da linguagem, visto que a implementação corrente é bastante simples e passível de otimizações. Também é necessária a implementação de mais aplicações para a realização de testes de desempenho do sistema, comparando-as com códigos baseados em bloqueios.

Referências

Costa, C. M., Stringhini, D., and Cavalheiro, G. G. H. (2002). Programação concorrente: Threads, MPI e PVM. *2a Escola Regional de Alto Desempenho ERAD*, pages 31–65.

Du Bois, A. R. and Echevarria, M. (2009). 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*, pages 170–186, Berlin, Heidelberg. Springer-Verlag.

Harris, T., Marlow, S., Peyton, S., and Herlihy, J. M. (2005). Composable memory transactions. pages 48–60. ACM Press.

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

Jones, S. P., Santos, A., and Santos, J. A. (1994). Compilation by transformation in the glasgow haskell compiler. *Glasgow Workshop on Functional Programming*, pages 184–204.

Parr, T. J. and Quong, R. W. (1994). Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810.