Projeto e implementação de uma linguagem de programação para a execução de algoritmos adaptativos (18 Outubro 2010)

J. A. Sabaliauskas[8], R. A. Rocha[9]

***Resumo* - Este artigo descreve uma proposta de implementação de uma linguagem de programação de alto nível capaz de expressar adequadamente algoritmos adaptativos seguindo a proposta de linguagens adaptativas descritas em [1] e a proposição de um paradigma de programação adaptativa descrito em [2]. Ao invés de se especificar toda uma nova linguagem de programação, foi escolhida a linguagem de programação Oberon [5] e a ela adicionadas construções sintáticas para a expressão de algoritmos adaptativos junto à programação imperativa usual. Para a geração de código executável foi utilizada a LLVM [3] (Low Level Virtual Machine), uma biblioteca disponível para o estudo de compiladores e de algoritmos de otimização, capaz de gerar código executável para diversas arquiteturas existentes. Essa proposta foi realizada e exercitada, os resultados foram mostrados no anexo do artigo.**

***Palavras chave*— Programação, Compilação, Adaptatividade, Oberon, LLVM, recursão de cauda.**

I. Nomeclatura

LLVM *Low Level Virtual Machine* [3]. Conjunto de componentes para a implementação de compiladores e máquinas virtuais

DAG *Direct Acyclic Graph*. Tipo de grafo que não contém ciclos.

II. Introdução

A ADAPTATIVIDADE é a técnica na qual um programa altera a si próprio em tempo de execução para resolver determinado problema. Os algoritmos que utilizam essa técnica de modificação de código são chamados de algoritmos adaptativos.

Entre as décadas de 1950 e 1970 os programadores utilizavam essa técnica para minimizar a utilização de memória pelo programa pois as memórias possuíam baixa capacidade de armazenamento. As linguagens de programação utilizadas nessa época eram linguagens de baixo nível (linguagem de máquina ou linguagens de montagem) onde o código do programa era expresso em uma linguagem de fácil compreensão pela máquina e difícil compreensão pelo ser humano.

Com o aumento da complexidade dos sistemas de software essas técnicas de programação tornaram-se uma fonte de erros devido à dificuldade de leitura do código em linguagens de baixo nível por outros programadores, pouco poder de expressão da linguagem para expressar a adaptatividade e liberdade total à utilização de recursos do sistema computacional.

Para lidar com o aumento da complexidade dos sistemas, a engenharia de software passou a guiar o processo de desenvolvimento através de um conjunto de técnicas consideradas eficientes para a implementação de sistemas o que fez com que algumas técnicas caíssem em desuso, entre elas a alteração do próprio código durante a execução.

O surgimento das linguagens de alto nível contribuiu com o desuso das técnicas de modificação do próprio código durante execução pois não forneciam mecanismos de modificação de código em tempo de execução e as linguagens de baixo nível foram aos poucos abandonadas ficando com uso restrito à tarefas específicas.

Por um grande período a área de programas adaptativos ficou estagnada até o surgimento de novas aplicações nas áreas de proteção de programa, compressão de código, inteligência artificial, otimização de código, etc.

III. Motivação

As linguagens de alto nível atuais não foram projetadas para uma correta expressão de algoritmos adaptativos, devido aos problemas que o uso não estruturado dessa técnica pode trazer.

Para se implementar a adaptatividade atualmente pode-se

[8] Jorge Augusto Sabaliauskas <jaugustosaba@gmail.com> é aluno do ultimo ano de graduação de engenharia de computação da Escola Politécnica da Universidade de São Paulo

[9] Ricardo Luís Azevedo da Rocha <ricardoluis.rocha@gmail.com> é docente do Laboratório de Tecnologia Adaptativa (LTA) da Escola Politécnica da Universidade de São Paulo (www.pcs.usp.br/~lta).

recorrer ao uso de linguagens de baixo nível, linguagens interpretadas ou a adição de um interpretador ao programa.

Os mesmos problemas que foram encontrados no passado serão obtidos ao se utilizar alguma linguagem de baixo nível para expressar um algoritmo adaptativo. Esses códigos são dependentes da arquitetura do computador em uso e determinarão a portabilidade do programa. A técnica de alteração do próprio código de máquina é considerado ilegal pelos sistemas operacionais modernos. Para se utilizar essa técnica será necessário utilizar-se de chamadas de sistema a fim de remover essa proteção.

As linguagens interpretadas fornecem uma alternativa para a implementação de adaptatividade. A maioria das linguagens de programação destinadas à execução através de um interpretador possuem uma função chamada *eval*. A função *eval* faz com que um fragmento de código seja executado em determinado contexto. A usual presença dessa função em linguagens interpretadas é devido a reaproveitamento de componentes do próprio interpretador para implementar essa função. Em uma linguagem interpretada qualquer pode-se simular a adaptatividade através do uso da função *eval*, gerando durante a execução do programa um trecho de código que será executado.

As linguagens interpretadas não podem ser utilizadas em todas as situações por possuírem um menor desempenho quando comparadas à linguagens compiladas. Essa perda de desempenho é acentuada ainda mais em programas adaptativos devido ao uso excessivo da função *eval* durante a execução.

Para tentar agilizar a execução do programa, ou caso a linguagem de programação não seja interpretada (não possui o comando *eval*), pode-se acrescentar ao programa um interpretador que ficará responsável exclusivamente por executar os algoritmos adaptativos. Pode-se construir um interpretador dedicado aos algoritmos adaptativos ou então adotar alguma linguagem de extensão disponível (a maioria das linguagens de programação interpretadas permitem que sejam usadas também como linguagens de extensão). A utilização do interpretador dedicado a execução de algoritmos adaptativos pode não ter um ganho de performance significativo caso o programa utilize intensamente algoritmos adaptativos.

Nas três formas de implementar adaptatividade o algoritmos adaptativo não fica expresso explícitamente, necessitando de documentação adicional (comentários ou documentos) para informar a presença de um algoritmo adaptativo. A responsabilidade de uma correta documentação fica sobre o programador. O programador também fica responsável pelo correto teste do programa, pois o compilador (ou interpretador) não poderá auxiliar na detecção de erros devido ao seu desconhecimento particular sobre algoritmos adaptativos.

Para expressar de maneira coerente um algoritmo adaptativo faz-se necessário a especificação da linguagem de programação e do compilador estarem cientes sobre a existência de algoritmos adaptativos a fim de auxiliar o programador no processo de desenvolvimento e padronização do código.

Através da proposta de uma linguagem de programação com características adaptativas em [1] e [2] tornou-se possível a implementação de algoritmos adaptativos sem que haja necessidade de alterar o código de máquina durante tempo de execução, servindo também como base para a proposição de compilador ao invés de um interpretador.

IV. Definição da linguagem

A maioria dos programadores atuais estão acostumados ao uso de linguagens de programação que seguem o paradigma imperativo. Para facilitar a adoção da linguagem pelos programadores, a linguagem possuirá todos os comandos e declarações presentes em linguagens imperativas e a adaptatividade será expressa através de uma extensão que poderá ser utilizada opcionalmente pelo programador.

Porderia-se propor uma extensão a uma linguagem não necessariamente imperativa como por exemplo as linguagens que seguem paradigmas declarativos como o funcional e o lógico. Esses outros tipos de linguagem necessitam de técnicas de compilação mais complexas que uma linguagem imperativa pois as linguagens imperativas refletem a arquitetura dos computadores atuais, que são baseados na arquitetura de Von Neumann. Portanto adotou-se o uso do paradigma de programação imperativo por simplicidade.

Ao invés de definir uma linguagem de programação do zero, optou-se por utilizar a linguagem de programação Oberon [5], por possuir uma sintaxe simples, representar praticamente todas as construções presentes em linguagens imperativas e ao seu sistema de módulos.

V. Extensão da linguagem

A adaptatividade será expressada na linguagem de programação através de blocos de código e conexões entre eles ao invés de se trabalhar com a alteração de código de máquina durante a execução. A alteração do código do programa durante a execução será feita através da modificação da ligação entre cada bloco de código, onde cada uma dessas ligações indicam qual é o próximo bloco de código a ser executado. Para o prosseguimento de um trecho do programa pode ser necessário com que as ligações entre os blocos sejam alteradas e, para determinado bloco, ao invés de simplesmente transferir o controle, é executada uma ação de rearranjo das ligações entre todos blocos.

A extensão adaptativa foi inserida na linguagem através de um tipo especial de procedimento que é chamado como se fosse um procedimento comum da linguagem Oberon (no momento da chamada é impossível diferenciar um procedimento adaptativo de um procedimento comum).

Dentro do procedimento adaptativo é possível declarar de um procedimento convencional, um procedimento adaptativo, declarar fragmentos de código e declarar ações. Em cada procedimento adaptativo é obrigatório declarar o estado inicial das conexões entre os fragmentos.

```
DeclarationSequence = ["CONST" {ConstDeclaration ";"}]
```

```
                    ["TYPE" {TypeDeclaration ";"}]
                                ["VAR"
{VariableDeclaration ";"}]
                                        {ProcedureDeclaration
";"}

{AdaptProcedureDeclaration ";"}.

AdaptProcedureDeclaration = AdaptProcedureHeading ";"
                            AdaptProcedureBody
                                                    "END"
ident.

AdaptProcedureHeading = "ADAPTIVE" identdef
                        [FormalParameters].

AdaptProcedureBody = DeclarationSequence
                                [AdaptFragment {";"
AdaptFragment}]
                                    ["ACTIONS"
                    [AdaptAction {";"AdaptAction}]]
                        "CONNECTIONS"
                    AdaptStmts.
```

O algoritmo adaptativo é executado através dos fragmentos e cuja lógica de controle é definida nos blocos de ação.

A. Declaração de fragmentos

Os fragmentos constituem os blocos de código imperativo atômicos. Dentro de cada fragmento podem conter qualquer comando da linguagem Oberon (comandos não adaptativos), referência a variáveis do procedimento, variáveis globais, procedimento locais, procedimentos globais e acesso aos parâmetros do procedimento.

Cada fragmento é referenciado a partir do seu nome, são visíveis apenas no escopo do procedimento adaptativo no qual foi declarado e cada nome só pode ser associado a um único fragmento dentro desse escopo.

```
AdaptFragment = "FRAGMENT" ident ";"
                                StatementSequence
                        "END" ident ";".
```

B. Declaração de ações

São blocos de definições das conexões entre os fragmentos dentro de um procedimento adaptativo.

```
AdaptAction = "ACTION" ident ";"
                AdaptStmts
                    "END" ident ";".
```

C. Comandos adaptativos

Os dois comandos adaptativos tem como objetivo selecionar qual ação será executada para um determinado fragmento de origem. Um deles permite que uma entre diversas ações de controle seja selecionada para a execução a partir de uma condição e o outro comando executa uma ação de controle apenas informando qual é o fragmento de origem.

```
AdaptStmts = [AdaptStmt {";" AdaptStmt}].
AdaptStmt = ConditionalAdaptStmt | InconditionalAdaptStmt.
```

1) Seleção de comando adaptativo

Seleciona uma ação de controle através de uma condição para determinado fragmento de origem. A ação de controle que será executada será a primeira em que sua condição estiver verdadeira e deverá ser informada a ação de controle que deverá ser executada caso nenhuma das condições seja verdadeira.

```
ContionalAdaptStmt = "AFTER" ident
                                    AdaptControl
"CASE" expression ","
                            {AdaptControl    "CASE"
expression ","}
                            AdaptControl
"OTHERWISE".
```

2) Execução de comando adaptativo incondicional

Executa uma ação de controle para determinado fragmento de origem.

```
InconditionalAdaptStmt = "AFTER" ident AdaptCommand.
```

3) Comandos de ação de controle

São comandos que irão indicar qual é o próximo fragmento de código que será executado, qual deve ser a ação usada para continuar a execução do altorimo adaptativo ou terminar a execução do algoritmo adaptativo e opcionalmente retornar um valor.

```
AdaptControl =          GotoControl
               |    PerformControl
               |    ReturnControl.
```

a) Comando de salto para próximo fragmento

Indica o próximo fragmento a ser executado. O próximo fragmento é identificado a partir de seu nome.

```
GotoControl = "GOTO" ident.
```

b) Comando de redefinição de conexões

É um comando que transfere a lógica de conexão entre os fragmentos para um outro bloco de ações. O controle da execução do algoritmo adaptativo passa para o bloco de ações identificado até que ocorra uma outra chamada de redefinição de conexões (o controle só volta para essa ação caso ocorra um outro comando de redefinição de conexões tendo como alvo a ação atual).

```
PerformControl = "PERFORM" ident.
```

c) Comando de retorno de procedimento

É o comando que encerra a execução do procedimento

adaptativo, pode ou não devolver um resultado. O resultado devolvido por esse comando deve ter o mesmo tipo de retorno definido na declaração do procedimento adaptativo.

ReturnControl = "RETURN" [expression].

VI. FUNÇÃO FATORIAL

```
MODULE module;
    VAR fat10:INTEGER;

        (* procedimento adaptativo para calcular fatorial *)
        ADAPTIVE fat(n:INTEGER):INTEGER;
            VAR result:INTEGER;

    (* a inicialização do algoritmo
       começa pelo primeiro
       fragmento *)
            FRAGMENT init;
                result := 1
            END init;

            (* passo de cálculo *)
            FRAGMENT calc;
                result := result * n;
                n := n - 1
            END calc;
        ACTIONS
            (* conexões executadas caso n > 0 *)
            ACTION not0;
                AFTER calc RETURN result CASE n = 0,
                    GOTO calc OTHERWISE
            END not0;
        CONNECTIONS
            (* conexões iniciais *)
        AFTER init RETURN result CASE n = 0,
            PERFORM not0 OTHERWISE
        END fat;
BEGIN
    fat10 := fat(10)
END module.
```

VII. COMPILAÇÃO

A compilação da linguagem de programação é feita com o auxílio da LLVM [3] para gerar o código de máquina. O compilador, ao invés de gerar o código de máquina diretamente, gera um texto que contém uma representação intermediária do programa e a LLVM fica responsável pela transformação da representação intermediária em código de máquina.

A representação intermediária da LLVM utiliza o formato de atribuição única e é muito parecida com linguagens de montagem (porém utiliza um sistema de verificação de tipos). A LLVM recebe como entrada um texto em notação intermediária, realiza as otimizações e gera um código chamado de *bytecode* que pode ser transformado em código de máquina para alguma arquitetura ou executado diretamente através de um interpretador.

Para cada comando da linguagem Oberon foi mapeado um conjunto de instruções na representação intermediária da LLVM. Nesse mapeamento foi utilizado o conceito de blocos básicos de instruções que também é utilizado pela LLVM.

O protótipo de um compilador ficou dividido em analisador léxico, analisador sintático, gerador de código intermediário, gerador de código LLVM e emissão de código LLVM.

A. Analisador léxico e sintático

O analisador léxico e sintático é feito utilizando-se as técnicas tradicionais de compiladores. Em ambos os casos foram utilizadas ferramentas para a geração do analisador léxico e analisador sintático. O analisador sintático possui como saída a representação do programa em forma de árvore.

B. Gerador de código intermediário

O gerador de código intermediário recebe como entrada a árvore do programa, a valida e gera o código intermediário. O código intermediário é representado através de um grafo direto acíclico (DAG - *Direct Acyclic Graph*). Cada comando do código fonte do programa é substituído por um comando mais simples como por exemplo os comandos *IF* e WHILE que são substituídos por um simples comando de salto condicional. Essa representação mais simples dos comandos visa facilitar a sua tradução para código de máquina nos passos posteriores.

C. Gerador de código LLVM

O gerador de código LLVM recebe como entrada o grafo do programa e o transforma em uma estrutura de dados representando o programa em linguagem de montagem na LLVM. Cada instrução simples do programa é mapeada em uma ou mais instruções em linguagem de montagem constituindo um bloco básico no código LLVM.

D. Emissão de código LLVM

A emissão de código LLVM é feita percorrendo a estrutura de dados contendo o código LLVM. É representado o código de montagem LLVM em estrutura de dados antes para facilitar a sua manipulação pelo gerador de código intermediário.

VIII. COMPILAÇÃO DA EXTENSÃO ADAPTATIVA

A compilação da extensão adaptativa foi projetada para utilizar intensamente a otimização de chamadas de cauda realizada em chamada de funções (consule [4] para maiores

detalhes sobre chamadas de cauda).

A otimização de chamada de cauda é feita quando uma função é chamada em posição de cauda. Uma chamada de função é dita em posição de cauda quando a função que realiza a chamada não realiza nenhum processamento após a chamada, permitindo com que a nova chamada seja feita descartando o contexto da função atual. Ao não ter que armazenar o contexto da função permite que um número arbitrário de chamadas de função em posição de cauda sejam realizadas sem que ocorra estouro de pilha.

Tanto fragmentos quanto ações foram mapeadas em funções em código de montagem LLVM.

Os fragmentos recebem como parâmetro a função de ação que será executada depois do fragmento o que permite com que a função de ação seja chamada em posição de cauda.

As ações recebem como parâmetro a função de fragmento que acabou de executar a fim de identificar o fragmento de origem para selecionar a ação de controle que deverá ser executada. As duas únicas chamadas de função executadas dentro de um bloco de ação também permitem a otimização de chamada de cauda: a redefinição de conexões que é implementada através da chamada de uma segunda função ação e a transferência de controle para fragmento através da chamada de uma função que representa um fragmento. Em ambas as chamadas de função o bloco de ação terminou o seu processamento.

IX. Conclusão

A extensão proposta para a implementação de algoritmos adaptativo possui como vantagem manipular de maneira organizada o próprio código fonte. A adaptatividade fica claramente exposta no código fonte de um programa através da existência de um procedimento dedicado para a implementação de algoritmos adaptativos ao contrário das técnicas de auto-modificação de código de máquina e uso da função *eval*.

A implementação em baixo nível do procedimento adaptativo é feito unicamente através de funções não necessitando armazenar as informações sobre as conexões em variáveis globais. Dessa maneira o algoritmo adaptativo pode ser executado no mesmo instante por múltiplas linhas de execução sem que haja a necessidade de sincronização.

Os procedimentos adaptativos não possuem memória, ou seja, o procedimento adaptativo não armazena as conexões de uma chamada anterior. Se esse for um efeito desejado, pode-se implementar esse efeito através da definição do uso de variáveis ou estruturas de dados para serem passados como parâmetro para o procedimento adaptativo, ficando ele responsável pelo armazenamento do estado atual no final de sua execução e leitura e a execução de ação de redefinição de conexões no início de sua execução.

Referencias

[1] S. B. da Silva e J. J. Neto “Projeto de uma linguagem para programação adaptativa“, 2010.

[2] S. B. da Silva e J. J. Neto “Um método para programação adaptativa” XVI Congreso Argentino de Ciencias de la Computacion, 2010.

[3] Low Level Virtual Machine, llvm.org acessado em 18/10/10.

[4] Daniel P. Friedman e Mitchel Wand, “Essentials of Programming Languages”, MIT Press, 2008.

[5] Niklaus Wirth, “The Programming Language Oberon”, revisão 01/11/2008.

ANEXO: REPRESENTAÇÃO INTERMEDIÁRIA PARA A FUNÇÃO FATORIAL

```
;********************
; apelidos para tipos            *
;********************
%l1 = type i32 (%l2, i32*, i32*)* ; função tipo conexão
%l2 = type i32 (%l1, i32*, i32*)* ; função tipo fragmento

;***********************
; VAR fat10:INTEGER;        *
;***********************
@fat10 = internal global i32 0

;******************
; FRAGMENT init; *
;******************
define i32 @init(%l2 %l4, i32* %l0, i32* %result)
{
l8:                                ;
  store i32 1, i32* %result  ;  result := 1
  br label %l6                     ;
l6:
  %l7 = tail call i32 (%l1, i32*, i32*)* %l4 (%l1 @init, i32* %l0,
i32* %result)
  ret i32 %l7
}

;******************
; FARGMENT calc; *
;******************
define i32 @calc(%l2 %l4, i32* %l0, i32* %result)
{
l17:
  %l9 = load i32* %result
  %l10 = load i32* %l0                    ;
  %l11 = mul i32 %l9, %l10          ;  result := result * n;
  store i32 %l11, i32* %result      ;
  br label %l16
l16:
  %l12 = load i32* %l0              ;
  %l13 = sub i32 %l12, 1            ;  n := n - 1
  store i32 %l13, i32* %l0          ;
  br label %l14
l14:
  %l15 = tail call i32 (%l1, i32*, i32*)* %l4 (%l1 @calc, i32* %l0,
i32* %result)
  ret i32 %l15
}

;********************
; FRAGMENT not0;            *
;********************
define i32 @not0(%l1 %l3, i32* %l0, i32* %result)
{
l26:
          ;
  %l27 = icmp eq %l1 %l3, @calc          ; AFTER calc
  br i1 %l27, label %l24, label %l25     ;
l24:
  %l18 = load i32* %l0                   ;
  %l19 = icmp eq i32 %l18, 0             ; CASE n = 0
  br i1 %l19, label %l21, label %l22
l21:
                                         ;
  %l20 = load i32* %result                    ; RETURN
result
  ret i32 %l20                                ;
l22:
  ;
  ; GOTO calc OTHERWISE
  ;
  %l23 = tail call %l1 @calc (%l2 @not0, i32* %l0, i32* %result)
  ret i32 %l23
l25:
   ; Aqui poderia ser informado um erro por falta de ação adaptativa
ret i32 0
}

;*****************************
; ações de conexão iniciais *
;*****************************
define i32 @l5(%l1 %l3, i32* %l0, i32* %result)
{
l36:                    ;
  %l37 = icmp eq %l1 %l3, @init      ; AFTER init
  br i1 %l37, label %l34, label %l35  ;
l34:
  %l28 = load i32* %l0                          ;
  %l29 = icmp eq i32 %l28, 0                    ; CASE n = 0
  br i1 %l29, label %l31, label %l32    ;
l31:
  %l30 = load i32* %result          ; RETURN result
  ret i32 %l30                                  ;
l32:
   ; PERFORM not0 OTHERWISE
   %l33 = tail call %l2 @not0 (%l1 %l3, i32* %l0, i32* %result)
  ret i32 %l33
l35:
  ; Aqui poderia ser informado um erro por falta de ação adaptativa
  ret i32 0
}

;************************************
; ADAPTIVE fat(n:INTEGER):INTEGER; *
;************************************
define i32 @fat(i32 %n)
{
l38:
  %l0 = alloca i32
  store i32 %n, i32* %l0
  %result = alloca i32
  %l39 = tail call %l1 @init (%l2 @l5, i32* %l0, i32* %result)
  ret i32 %l39
}

;***********************
; Inicialização do módulo *
;***********************
define void @module()
{
l42:
  %l40 = call i32 @fat (i32 10)          ;
  store i32 %l40, i32* @fat10            ; fat10 := fat(10)
  br label %l41                                  ;
l41:
  ret void
}
```