Programação Funcional e em Lógica

Identificação

Esse projeto foi desenvolvido por:
Luiz Henrique Mamede Queiroz (up202102362@fe.up.pt), Turma 11 Grupo 12, com contribuição de 100%

Descrição do Problema

Este é um projeto feito em Haskell para a disciplina de Programação Funcional e em Lógica onde é dividido em duas partes. Na parte 1 é considerado uma lower-level machine que recebe uma lista de instruções e a executa, podendo assim alterar a evaluation stack ou o storage. Na parte 2 é considerado uma pequena linguagem de programação imperativa que tem expressões aritméticas e booleanas e statements que precisam ser traduzidas para a lista de instruções definida na parte 1.

Descrição da Implementação

Na parte 1, foi definido os tipos Value, Stack e State (em 3 módulos diferentes) além dos já definidos Inst e Code. O tipo Value é um tipo que pode ser tanto inteiro quanto booleano, o tipo Stack é um tipo formado por uma lista de Value e o tipo State é um tipo formado por uma lista de tuple (String, Value).

As funções implementadas para o módulo Stack foram: push, pop, top, calculate, compare, neg, empty e isEmpty, onde push insere um valor na stack, pop retira o valor no topo da stack, top obtém o valor do topo da stack, calculate calcula, utilizando os dois valores no topo da stack, as operações de adição, subtração ou multiplicação, compare compara, utilizando os dois valores no topo da stack, para igualdade, menor ou igual ou and, neg funciona como a negação de um valor booleano que esteja no topo da stack, empty cria uma stack vazia e isEmpty verifica se a stack está vazia. Enquanto as funções implementadas para o módulo State foram: store, fetch, empty, isEmpty e stateSort, onde store guarda um tuple (String, Value) no State ou altera o valor de um tuple que possua a mesma String, fetch obtém o valor associado a uma String, empty cria um state vazio, isEmpty verifica se o state está vazio, stateSort ordena o state por ordem alfabética das Strings.

A partir disso, a primeira parte se resolve com a função run que vai verificar qual a instrução atual a partir da lista de instruções fornecida e aplicar as funções que permitem alcançar o objetivo dessa instrução. Assim, se a instrução for: Push -> será usado a função push do módulo Stack, Add -> será usado a função calculate do módulo Stack, Mult -> será usado a função calculate do módulo Stack, Sub -> será usado a função calculate do módulo Stack, Tru -> será usado a função push do módulo Stack, Fals -> será usado a função push do módulo Stack, Equ -> será usado a função compare do módulo Stack, Le -> será usado a função compare do módulo Stack, And -> será usado a função compare do módulo Stack, Neg -> será usado a função neg do módulo Stack, Fetch -> será usado a função fetch do módulo State com a função push do módulo Stack, Store -> será usado a função pop do módulo Stack e a função top do módulo Stack com a

função store do módulo State, Noop -> nada é feito, só se passa para a próxima instrução, Branch -> será usado a função top e pop do módulo Stack para obter o valor Booleano no topo da stack e saber qual condição será realizada no branch (se a primeira ou a segunda), Loop -> é feita utilizando as instruções branch, noop e a própria loop.

Para que sejam printados os resultados na forma desejada, são usadas as funções stack2Str que utiliza uma instância Show customizada para o tipo Stack e state2Str que utiliza uma instância Show customizada para o tipo State que será primeiramente ordenado.

Na parte 2, foram criados os tipos Aexp, Bexp e Stm para representar as expressões aritméticas (adição, subtração e multiplicação), booleanas (negação, igual, menor ou igual e and) e os statements (do tipo assignment, sequência de dois statements, if then else e while loops) respectivamente e o sinônimo de tipo program para representar a lista de statements. Foram criadas, também, as funções compA, compB e compile onde a função compA será responsável por fazer a transformação de alguma expressão artimética (Aexp) para o formato de lista de instruções da parte 1 que realize essa expressão aritmética, onde a função compB será responsável por fazer a transformação de alguma expressão booleana (Bexp) para o formato de lista de instruções da parte 1 que realize essa expressão booleana e onde a função compile será responsável por fazer a transformação de uma lista de statements (program) para o formato de lista de instruções da parte 1 que consiga realizar os objetivos de cada statement, sendo que para isso, a função compile poderá utilizar as funções compA e compB, a depender do statement em questão, para auxiliar na tarefa.

Também seria necessário criar a função parse para fazer a transformação da String que representa um programa imperativo na sua correspondente representação em lista de statements. Dessa forma, será possível a partir de um código de programa, fazer a transformação para a correspondente lista de statements utilizando a função parse, depois utilizar a função compile para transformar essa lista de statements numa lista de instruções e assim conseguir utilizar a função run para realizar essas instruções.

Exemplos

Para a parte 1, temos como exemplo a seguinte lista de instruções [Fals, Store "var", Fetch "var"] que deve gerar como resultado na stack e no state os respectivos valores em formato string ("False", "var=False").

Isso porque a instrução Fals adiciona o valor False no topo da stack (ou seja, stack ”False” e state “”), a instrução Store “var” pega o valor do topo da stack (que no caso é False), remove esse valor do topo da stack, atribui à variável var esse valor e salva no storage (ou seja, stack “” e state “var=False”) e, por fim, a instrução Fetch “var” procura a variável var no storage e coloca seu valor no topo da stack (ou seja, stack ”False” e state “var=False”).

Outro exemplo é a seguinte lista de instruções [Push 5, Store "x", Push 1, Fetch "x", Sub, Store "x"] que deve gerar como resultado na stack e no state os respectivos valores em formato string ("","x=4").

Isso porque a instrução Push 5 adiciona o valor 5 no topo da stack (ou seja, stack “5” e state “”), a instrução Store “x” pega o valor do topo da stack (que no caso é 5), remove esse valor do topo da stack, atribui à variável x esse valor e salva no storage (ou seja, stack “” e state “x=5”), a instrução Push 1 adiciona o valor 1 no topo da stack (ou seja, stack “1” e state “x=5”), a instrução Fetch “x” procura a variável x no storage e coloca seu valor no topo da stack (ou seja, stack ”5,1” e state “x=5”), a instrução Sub subtrai os dois valores do topo da stack (5-1 = 4) e substitui esses valores pelo resultado da subtração (ou seja, stack “4” e state “x=5”) e, por fim, a instrução Store “x” pega o valor do topo da stack (que no caso é 4), remove esse valor do topo da stack, atribui à variável x esse valor e salva no storage (ou seja, stack “” e state “x=4”)

Esses exemplos foram utilizados como testes e os resultados ocorreram conforme esperado.

Para a parte 2, temos como exemplo a string representando o código “x := 5; x := x - 1;” que deve gerar como resultado na stack e no state os respectivos valores em formato string (“”, ”x=4”).

Isso porque a primeira parte (x := 5) depois da função parse será um statement que fará a atribuição do valor 5 para a variável x e depois da função compile, esse statement gerará a lista de instruções para fazer essa atribuição [Push 5, Store “x”] (ou seja, stack “” e state “x=5”), a segunda parte (x := x – 1) depois da função parse será um statement que fará a atribuição do valor x-1 para a variável x e depois da função compile, esse statement gerará a lista de instruções para fazer essa atribuição [Push 1, Fetch “x”, Sub, Store “x”] (ou seja, stack “” e state “x=4).

Outro exemplo é a string representando o código "if (not True and 2 <= 5 = 3 == 4) then x :=1; else y := 2;" que deve gerar como resultado na stack e no state os respectivos valores em formato string (“”, “y=2”).

Isso porque depois da função parse será um statement if then else e depois da função compile, esse statement gerará a lista de instruções para fazer o condicional [Push 4, Push 3, Equ, Push 5, Push 2, Le, Equ, Tru, Neg, And, Branch ([Push 1, Store “x”]) ([Push 2, Store “y”])] (ou seja, stack “” e state “y=2”).

Esses exemplos foram utilizados como testes e os resultados ocorreram conforme esperado.

Conclusão

O projeto desenvolvido em Haskell para a disciplina de Programação Funcional e em Lógica envolveu a implementação de uma lower-level machine na Parte 1, capaz de executar uma lista de instruções, manipulando a evaluation stack e o storage. Além disso, a Parte 2 abordou uma pequena linguagem de programação imperativa com expressões aritméticas e booleanas e possíveis statements, que utilizam essas expressões, e podem ser traduzidos para a lista de instruções da Parte 1.

Na Parte 1, foram definidos tipos como Value, Stack, e State em módulos separados, juntamente com as funções específicas para manipulação de stacks e estados. A função run foi implementada para executar a lista de instruções, utilizando as operações correspondentes do módulo Stack e

State. A customização das instâncias Show para Stack e State permitiu uma representação amigável dos resultados.

Na Parte 2, os tipos Aexp, Bexp, e Stm foram criados para representar expressões aritméticas, booleanas e statements, respectivamente. Funções como compA, compB, e compile foram desenvolvidas para traduzir essas representações para lista de instruções da Parte 1. A função parse possibilita a conversão de programas imperativos representados como strings para a lista de statements.

Dessa forma, o projeto abrangeu desde a definição dos tipos até a implementação de funções específicas para obtenção dos resultados esperados, proporcionando aprofundar os conhecimentos da programação funcional.