Segundo Trabalho Prático de PFL 2023

Âmbito

O presente documento diz respeito ao Segundo Trabalho Prático da Unidade Curricular de Programação Funcional e em Lógica, da Licenciatura em Engenharia Informática e Computação, no ano letivo 2023/24.

O trabalho representa a aplicação prática de conhecimentos adquiridos no âmbito do estudo do paradigma de programação funcional, através da linguagem Haskell.

Grupo e Distribuição de Trabalho

Grupo **T05_G01**:

- Pedro de Almeida Lima (up202108806@up.pt) - contributo de 50%
- Pedro Simão Januário Vieira (up202108768@up.pt) - contributo de 50%

Problema

O problema proposto para abordagem no descrito trabalho centra-se numa pequena linguagem de programação imperativa.

Num primeiro momento, foi implementado um *assembler* que, dadas instruções em código análogo ao Assembly para as operações oferecidas pela pequena linguagem, é capaz de simular a execução do código, recorrendo a estruturas de dados que fornecem abstrações para uma pilha (*stack*), um pequeno armazenamento (*state*) e a própria sequência de instruções.

Já numa segunda parte, desenvolveu-se um *parser* e um compilador responsáveis por converter código-fonte da pequena linguagem de programação para o referido código-máquina a ser interpretado pela solução desenvolvida para a primeira parte.

Detalhes da Implementação

Primeira parte — *Assembler*

Tendo em conta que o propósito do *assembler* é o de processar instruções em *assembly*, é fundamental uma estrutura de dados que represente as ditas instruções. Tal é assegurado por `data Inst`, como fornecido no modelo para o código do trabalho. Um conjunto de instruções é representado por `type Code = [Inst]`.

Dada a necessidade de suporte para variáveis com nome e conteúdo, recorreu-se a `type Var = (VarName, VarValue)`, que, para representar uma variável, consiste num par composto pelo nome (`type VarName = String`) e pelo conteúdo (`data VarValue`) dessa variável. Esta última estrutura distingue o conteúdo entre inteiros e booleanos.

Para responder à necessidade de ter uma pilha (*stack*) auxiliar à execução do código, bem como algo que cumpra o propósito de memória/armazenamento para os programas (*state*), foram criadas, respetivamente, `type Stack = [VarValue]` e `type State = [Var]`. Recordando que `Var` engloba o nome e o conteúdo de uma variável, o tipo apenas foi usado no âmbito do estado, dado que não é esperado nem desejável que a pilha contenha tal informação.

Foram, ainda, definidas funções que permitem visualizar os conteúdos da pilha e da memória de uma forma mais inteligível (`stack2Str :: Stack -> String` e `state2Str :: State -> String`, respetivamente), bem como funções que inicializam uma instância de cada uma das duas estruturas (`createEmptyStack :: Stack` e `createEmptyState :: State`, respetivamente).

Implementou-se funções de manipulação da pilha (`top` (obter o elemento no topo), `pop` (remover o elemento no topo) e `push` (colocar um elemento no topo da pilha)).

Concretizámos as funções que efetivamente cumprem o papel das instruções do código *assembly*, conforme definidas em `data Inst`, fazendo os cálculos necessários e manipulando a pilha e o estado. Para cada instrução, foi criada uma função com o mesmo nome, à exceção da instrução de `Loop`, uma vez que pode ser reescrita recursivamente como uma instrução `Branch` que encapsula o `Loop`.

Para o descrito efeito, algumas funções principais recorrem a funções auxiliares, de maneira a separar as preocupações e facilitar a leitura (e escrita) do código, permitindo que uma função que desempenhe o papel de uma instrução possa ser lida como uma abstração de mais alto nível tendo em conta a operação que se realiza, "escondendo" em funções auxiliares os meandros da operação.

Tomemos como exemplo as funções afetas à instrução de adição (`Add`):

```
addValues :: VarValue -> VarValue -> VarValue
addValues (IntegerValue a) (IntegerValue b) = IntegerValue (a + b)
addValues _ _ = error "Run-time error"

add :: Stack -> Stack
add stack = push result (pop (pop stack))
            where result = top stack `addValues` top (pop stack)
```

No caso, podemos constatar que o trabalho de verificação de verificação da viabilidade da operação (dado que, concretamente, apenas podemos adicionar dois inteiros e não um inteiro e um booleano, ou dois booleanos) e de efetuar a adição em si é deixado para `addValues`, ficando a função primária, `add`, com a responsabilidade de manipular a pilha recorrendo ao resultado da operação.

Por último, temos a função `run :: (Code, Stack, State) -> (Code, Stack, State)`, que garante o fluxo de processamento das instruções desejadas. É recursiva; para uma dada instrução, após invocar a função correspondente, chama-se a si própria com as instruções restantes, bem como a pilha e o estado resultantes da operação efetuada.

Segunda parte — Compilador e *Parser*

Compilador

Tal como pretendido no enunciado, definimos 3 estruturas fundamentais no âmbito da compilação dos programas: `data Aexp`, para representar expressões aritméticas, i.e. que produzem um inteiro como resultado, `data Bexp`, para representar expressões booleanas, i.e. que produzem um booleano como resultado, e `data Stm`, para representar aquilo que, numa linguagem de programação imperativa, não é expressão: *statements*, que podem conter expressões e outros *statements*. Definimos, ainda, `type Program = [Stm]`, que estabelece que um programa, no âmbito da segunda parte do trabalho, é uma sequência de

statements, dado que uma expressão, por si só, não tem efeito num programa escrito numa linguagem imperativa, i.e. apenas tem valor se enquadrada num *statement* que defina o que efetivamente se faz com o seu resultado.

Implementámos 3 funções basilares da compilação:

```
compA :: Aexp -> Code
compA (AddExp a1 a2) = compA a2 ++ compA a1 ++ [Add]
compA (MultExp a1 a2) = compA a2 ++ compA a1 ++ [Mult]
compA (SubExp a1 a2) = compA a2 ++ compA a1 ++ [Sub]
compA (Var x) = [Fetch x]
compA (Num n) = [Push n]

compB :: Bexp -> Code
compB (AndExp b1 b2) = compB b2 ++ compB b1 ++ [And]
compB (LeExp a1 a2) = compA a2 ++ compA a1 ++ [Le]
compB (AEquExp a1 a2) = compA a2 ++ compA a1 ++ [Equ]
compB (BEquExp b1 b2) = compB b2 ++ compB b1 ++ [Equ]
compB (NegExp b) = compB b ++ [Neg]
compB (Bool True) = [Tru]
compB (Bool False) = [Fals]

compStm :: Stm -> Code
compStm (Assign name a) = compA a ++ [Store name]
compStm (While b stmList) = [Loop (compB b) (compStm stmList)]
compStm (IfThenElse b thenStmList elseStmList) = compB b ++ [Branch (compStm
thenStmList) (compStm elseStmList)]
compStm (SequenceOfStatements stmList) = concatMap compStm stmList
compStm NoopStm = [Noop]
```

São responsáveis por transformar expressões aritméticas, booleanas e *statements*, respetivamente, em código *assembly* tal como definido na primeira parte do trabalho, assegurando a correta ordem deste código. Como pode ser observado, todas elas recorrem à recursividade, uma vez que a natureza tanto das expressões como dos *statements* pressupõe imbricação.

Por último, tal como especificado no enunciado, temos `compile :: [Stm] -> Code` que, através de uma chamada a `concatMap`, do prelúdio-padrão, obtém o código *assembly* correspondente a cada *statement* e concatena-o numa só lista de instruções *assembly* (recordando, da primeira parte, que `type Code = [Inst]`).

Parser

Lorem ipsum dolor sit amet.

Destaque

Lorem ipsum dolor sit amet.

Exemplos de utilização

Lorem ipsum dolor sit amet.

Conclusões

Lorem ipsum dolor sit amet.

Grupo T05_G01, 01/01/2024