PFL 2023/24 - Haskell practical assignment Grupo 9 Turma 6

Contribuidores

João Pedro Sousa (up202106996). Contribuição: 50%

Emanuel Maia (up202107486). Contribuição: 50%

Descrição do Problema

> O objetivo da primeira parte do projeto é criar uma máquina que simule o comportamento de uma máquina de baixo nível, proporcionando uma plataforma para explorar a execução de programas e o funcionamento de estruturas de dados básicas. Essa compreensão foi vital para a segunda parte do projeto, onde desenvolvemos um compilador para traduzir programas de alto nível para o conjunto de instruções que esta máquina pode executar.
>
> A segunda parte do projeto envolve a construção de um compilador. Este compilador é responsável por traduzir programas escritos nesta linguagem de alto nível para um conjunto de instruções que podem ser executadas pela máquina de baixo nível desenvolvida na primeira parte do projeto. Essencialmente, a linguagem imperativa inclui expressões aritméticas e booleanas, bem como instruções de controlo, como atribuições, sequências de instruções, condicionais e loops. A tarefa aqui é mapear cada elemento da linguagem imperativa para as instruções específicas da máquina de baixo nível, garantindo que o programa traduzido se comporte da mesma forma que o programa original. Isso inclui entender como as expressões são avaliadas e como as instruções de controlo alteram o fluxo do programa. Além disso, a parte inclui o desenvolvimento de um parser que transforma a representação textual de um programa na linguagem imperativa para a sua forma estruturada em Haskell, facilitando assim a sua compilação para o conjunto de instruções da máquina.

Descrição da Implementação

Parte 1

> Esta implementação oferece um interpetador para uma máquina de baixo nível, conforme descrito no enunciado do projeto. A máquina processa uma série de instruções que manipulam uma stack e um state. Esta abordagem é delineada por estruturas de dados específicas e uma série de funções auxiliares, culminando na função `run`, que é o núcleo do interpetador.
>
> O tipo `StackData` é usado para elementos na stack, capaz de conter um inteiro (Value Integer), um booleano (Boolean Bool) ou uma expressão (Expression String). A Stack, uma lista de `StackData`, representa a evaluation stack da máquina, enquanto o State representa o armazenamento da máquina como uma lista de pares (String, `StackData`), onde a string é o nome da variável.
>
> As funções auxiliares incluem `createEmptyStack` e `createEmptyState`, que retornam, respectivamente, stack e state vazios. `StackDataToStr` converte um valor `StackData` para a sua representação em string, sendo integral para exibir a stack num formato legível. As funções

> `stack2Str` e `state2Str` convertem toda a stack e o state em representações de string.
>
> A função `run` é o núcleo deste interpetador. Processa recursivamente cada instrução na lista de códigos fornecida, modificando a stack e o state conforme as semânticas de cada instrução. Cada tipo de instrução é tratado de acordo com a sua definição. Operações de manipulação de stack como `Push`, `Fetch` e `Store` manipulam diretamente a stack e o state. Operações aritméticas como `Add`, `Sub` e `Mult` realizam operações aritméticas sobre os valores inteiros na stack. Operações booleanas como `Tru`, `Fals`, `Equ`, `Le`, `And` e `Neg` realizam operações ou comparações booleanas. `Noop` não faz nada, `Branch` toma decisões baseadas no valor superior da stack e `Loop` implementa o comportamento de loop.
>
> O interpetador inclui verificações de erro, como underflow da stack ou incompatibilidades de tipo, garantindo robustez.

Parte 2

> A 2.ª parte do trabalho prático envolve a implementação de um compilador para uma simples linguagem de programação imperativa. Esta linguagem inclui expressões aritméticas, expressões booleanas, atribuições de variáveis, instruções condicionais (`if`-`then`-`else`) e ciclos `while`.
> A compilação do programa passa inicialmente pelo *parsing* do código fonte. Para isto serve a função `parse`, que por sua vez coordena as funções `aexpParse`, para expressões aritméticas, `bexpParse` para expressões booleanas e `stmParse` para instruções próprias da linguagem, como atribuições de variáveis (`:=`) ou então estruturas `while`, `if`-`then`-`else`, etc.
> O *parsing* gera código com que o compilador consegue trabalhar, através das seguintes estruturas:
>
> - `Aexp` - para representar expressões aritméticas (`ANum`, `AMul`, etc)
> - `Bexp` - para representar expressões booleanas (`BTrue`, `BNot`, etc)
> - `Stm` - para representar *statements* (`Assign`, `If`, etc)
> - `Program` - lista de *statements* (`Stm`)
>
> Uma vez compilado o programa, consegui-mos executálo através da função `run`, referida na 1.ª parte do projeto.
> Esta implementação faz uso de combinadores de analisadores e funções de ordem superior para a construção de analisadores.

Conclusões

> Este projeto proporcionou-nos uma compreensão profunda e detalhada sobre a programação em níveis mais baixos, especialmente no que toca à manipulação direta da stack e do armazenamento. Ao longo do desenvolvimento, deparamo-nos com a complexidade de traduzir conceitos e operações de alto nível em instruções de máquina de baixo nível.