Segundo trabalho - Máquinas de Pilha

Linguagens e Ambientes de Programação
NOVA FCT

Versão de 7 de Maio de 2024

- 6 de Maio de 2024 - Versão inicial do enunciado.
- 7 de Maio de 2024 - Adição do Erro "Division by zero".

O objectivo deste trabalho é implementar um fragmento da funcionalidade de uma máquina de pilha muito parecida com a JVM.

1 Introdução

A contribuição de Peter Landin em 1964 com a construção de uma máquina virtual (SECD[1]) foi fundamental para o desenvolvimento das linguagens de programação funcionais, como o LISP, e, sem qualquer dúvida, para o aparecimento de linguagens de programação em que a principal vantagem era a portabilidade, como o Java, o JavaScript ou o C♯. Tendo como foco neste trabalho a máquina virtual JVM, que suporta todo o ecosistema Java, vamos neste trabalho explorar um fragmento do seu funcionamento: a execução de código dentro de uma única função utilizando uma pilha para a execução de expressões e controlo de fluxo de execução.

Considere como exemplo o seguinte programa em Java na Figura 1 que calcula o valor de um polinómio de grau 2: Quando compilado e analisado pelos

[1] O Acrónimo SECD representa a utilização de 4 registos para conter estruturas de dados de apoio à execução de um programa: **S**tack, **E**nvironment, **C**ontrol and **D**ump. Pode ler mais em `https://en.wikipedia.org/wiki/SECD_machine`

```
1 public class Polynomial {
2   private int a,b,c;
3
4   public Polynomial(int a, int b, int c)
5     { this.a = a; this.b = b; this.c = c; }
6
7   public int evaluate(int x)
8     { return a*x*x + b*x + c; }
9 }
```

Figura 1: Programa em Java que calcula o valor de um polinómio

```
1  public class Polynomial {
2    public Polynomial(int, int, int);
3      Code:
4         0: aload_0
5         1: invokespecial #1
   // Method java/lang/Object."<init>":()V
6         4: aload_0
7         5: iload_1
8         6: putfield      #7                  // Field a:I
9         9: aload_0
10       10: iload_2
11       11: putfield      #13                 // Field b:I
12       14: aload_0
13       15: iload_3
14       16: putfield      #16                 // Field c:I
15       19: return
16
17   public int evaluate(int);
18     Code:
19        0: aload_0
20        1: getfield      #7                  // Field a:I
21        4: iload_1
22        5: imul
23        6: iload_1
24        7: imul
25        8: aload_0
26        9: getfield      #13                 // Field b:I
27       12: iload_1
28       13: imul
29       14: iadd
30       15: aload_0
31       16: getfield      #16                 // Field c:I
32       19: iadd
33       20: ireturn
34 }
```

Figura 2: Código intermédio gerado pelo comando `javap -c Polynomial.class`

comandos `javac` e `javap -c` obtemos o programa em código intermédio (*bytecode*) que pode ver na Figura 2. Como pode observar, a estrutura de um ficheiro *bytecode* corresponde à estrutura de classes e métodos de um programa Java, mas o interior de cada método é reescrito numa linguagem, chamada intermédia, de uma máquina de pilha. Na realidade, há ainda um passo de compilação em tempo de carregamento de código que torna a execução dos programas Java mais eficiente. Vamos neste contexto ignorar esse facto e analisar apenas a interpretação destas instruções.

Ao contrário de uma máquina de registos, uma máquina de pilha caracteriza-se por um conjunto de instruções muito simples, com poucos (um) ou nenhuns parâmetros. Os argumentos para cada uma das instruções são previamente colocados numa pilha de execução. A execução de cada instrução retira os argumentos da pilha, executa a operação e coloca o resultado de volta na pilha. Por exemplo, partindo de uma pilha vazia, o conjunto de instruções:

```
1 Push 1, Push 2, Add, Push 3, Mul
```

corresponde à avaliação da expressão $(1 + 2) \times 3$ e resulta numa pilha com um único valor, 9. Analisando o código da Figura 2 vemos algumas instruções cujo significado não é tão claro. Por exemplo, a instrução `aload.0` (linha 4) carrega o parâmetro 0 do método ou método estático corrente para o topo da pilha, num método o parâmetro 0 corresponde ao objecto `this`. As instruções `aload` carregam parâmetros e variáveis locais de tipo referência para objecto ou array. A instrução `iload.1` carrega o primeiro parâmetro do método (`x`) para o topo da pilha. As instruções `iload` carregam parâmetros e variáveis locais de tipo inteiro. Note agora a instrução `invokespecial` (linha 5). Esta trata de invocar o construtor por omissão da superclasse `Object`, com o objecto `this` no topo da pilha. Após estas duas instruções, a pilha, que começa vazia, encontra-se novamente vazia. Note agora a instrução `putfield #7` (linha 6). Esta instrução coloca o valor que está no topo da pilha no campo `a` do objecto `this` (também no topo da pilha). A pilha, depois de ter o valor do objecto `this` e o valor para `a` passado no primeiro parâmetro no topo da pilha, encontra-se de novo vazia. O construtor repete este padrão até à instrução `return`.

Analise agora o código do método `evaluate` (linhas 19 a 33). Corresponde de forma aproximada à sequência de instruções que vimos acima para fazer adições e multiplicações. Note que no final do método, a instrução `ireturn` que usa o valor no topo da pilha para devolver como resultado do método. A pilha apenas pode ter um valor, neste caso inteiro, no topo quando a instrução `ireturn` é executada, ou estar vazia quando a instrução `return` é executada. No trabalho que vamos descrever agora, iremos usar uma linguagem simplificada, e só valores de tipo inteiro, em relação ao real da JVM, mas sem qualquer perda de generalidade.

2 Descrição do Trabalho

O trabalho consiste em implementar um interpretador para um conjunto de instruções semelhantes às instruções da JVM. Começamos pode definir o conjunto de instruções que deve implementar, a forma de as organizar e a forma de as executar.

2.1 Instruções

Considere o seguinte conjunto de instruções, numa representação em ASCII, que envolvem apenas valores inteiros colocados numa pilha de execução:

1. PUSH n: Coloca o valor n (de tipo inteiro) no topo da pilha.
2. POP: Retira (e descarta) o valor do topo da pilha.
3. DUP: Duplica o valor do topo da pilha. Uma pilha com um único valor n passa a ter dois valores n.
4. SWP: Troca os dois valores do topo da pilha. Uma pilha com dois valores n e m (n no topo) passa a ter dois valores n e m (m no topo).
5. OVER: Duplica o segundo valor do topo da pilha. Uma pilha com dois valores n e m (n no topo) passa a ter três valores m, n, m.
6. ADD: Retira os dois valores do topo da pilha, soma-os e coloca o resultado no topo da pilha.
7. SUB: Retira os dois valores do topo da pilha, subtrai os dois valores do topo da pilha e coloca o resultado no topo da pilha. O valor que estava no topo da pilha é o subtraendo e o valor que estava imediatamente abaixo é o minuendo.
8. MUL: Retira os dois valores do topo da pilha, multiplica-os e coloca o resultado no topo da pilha.
9. DIV: Retira os dois valores do topo da pilha, divide-os e coloca o resultado no topo da pilha. O valor que estava no topo da pilha é o divisor e o valor que estava imediatamente abaixo é o dividendo.
10. CMP: Retira os dois valores do topo da pilha, compara-os e coloca o resultado da comparação no topo da pilha. O resultado da comparação é um inteiro que indica se os valores são iguais (1) ou diferentes (0).
11. JMP *label*: Salto incondicional para a instrução com a etiqueta *label*.
12. JZ *label*: Salto condicional para a instrução com a etiqueta *label* se o valor no topo da pilha for zero. Retira o valor do topo da pilha. Se não saltar para a instrução com a etiqueta *label*, continua para a instrução seguinte.
13. JNZ *label*: Salto condicional para a instrução com a etiqueta *label* se o valor no topo da pilha for diferente de zero. Retira o valor do topo da pilha. Se não saltar para a instrução com a etiqueta *label*, continua para a instrução seguinte.
14. RETURN: Termina a execução do programa e devolve o valor no topo da pilha como resultado. A pilha só pode ter um valor no topo quando esta instrução é executada ou o interpretador termina em erro.

Este conjunto de instruções é semelhante àquelas visíveis na Figura 2 mas acrescentamos aqui as instruções de salto incondicional e salto condicional. Isto quer dizer ainda que as instruções são etiquetadas com labels (strings) que permitem referenciá-las nas instruções de salto.

2.2 Blocos básicos

Uma organização típica de um programa em linguagem intermédia consiste em organizar as instruções num conjunto de blocos associados a labels que contendo uma sequência de instruções (diferentes de JMP) e terminam com uma instrução de salto incondicional (JMP) ou de terminação (RETURN).

Considere o seguinte exemplo com um programa que calcula a soma dos números de 0 a 99, composto por três blocos básicos começados pelas labels Start, Loop e Exit. Considere ainda que o comentário contendo o conteúdo da stack que se segue a cada instrução é apenas para ajudar a perceber o que cada instrução faz e não está presente no input do interpretador. O nome acc denota o acumulador que guarda o resultado da soma e n o cursor e número que está a ser somado.

```
1  Start: PUSH 0    -- acc
2         PUSH 99   -- n, acc
3         JMP Loop  -- n, acc
4
5  Loop:  DUP       -- n, n, acc
6         PUSH 0    -- 0 , n , n, acc
7         CMP       -- B, n, acc
8         JNZ Exit  -- n, acc
9         SWP       -- acc, n
10        OVER      -- n, acc, n
11 L0:    ADD       -- acc+n, n
12        SWP       -- n, acc+n
13        PUSH 1    -- 1, n, acc
14        SUB       -- n-1, acc
15        JMP Loop  -- n-1, acc
16
17 Exit:  POP       -- n, acc
18        RETURN    -- acc, OK!
```

A estrutura de blocos básicos é essencial para a implementação eficiente de um interpretador. Note que todas as linhas podem ter *labels* associadas, a condição para formar um bloco é terminar com uma instrução JMP. Por razões de simplicidade não vamos considerar a possibilidade de instruções de salto para instruções que não sejam o início de um bloco.

2.3 Execução de um programa

A execução de um programa deve ser feita por uma função que começa com uma pilha, uma *label* e que recursivamente vai percorrendo o conjunto dos blocos e a lista de instruções, aplicando o efeito de cada instrução à pilha que é passada para a instrução seguinte. A função deve terminar quando a instrução RETURN é executada e devolver o valor no topo da pilha. Se não houver instrução RETURN no bloco, a função deve terminar (failwith) com o erro "No return instruction" e se a pilha não estiver vazia "Stack not empty".

Como estamos a falar da execução de programas, que podem até não terminar, é essencial usar uma função *tail recursive* para evitar o uso excessivo de memória na *stack* e erros de *stack overflow*.

2.4 Erros possíveis

Durante o processamento de um programa, podem ocorrer erros de execução que são atribuíveis ao utilizador (*input*). Os erros possíveis são:

- "Invalid line": Se a linha não tiver o formato certo.
- "Expecting label": Se a linha devia ter uma label e não tem.
- "Invalid instruction": Se a instrução que consta do programa não for uma das instruções válidas.
- "Division by zero": Se a instrução DIV for executada com o divisor a zero.
- "Unexpected empty stack": Se a pilha estiver vazia quando uma instrução que necessita de mais argumentos do que os existentes na pilha é executada.
- "Stack not empty": Um programa termina com a pilha não vazia.
- "No return instruction": Um programa termina sem encontrar uma instrução RETURN.
- "Label not found: *label*": Se uma instrução de salto para uma label que não existe é executada.

3 Especificação do interpretador

O interpretador deve ser implementado por uma função que aceita como argumento uma lista de strings que contêm instruções com labels e devolve o valor no topo da pilha quando a instrução RETURN é executada. A função deve ser implementada em OCaml e ter a seguinte assinatura:

```
val run : string list -> string -> int
```

por exemplo, o programa anterior seria representado por uma lista de strings:

```
let program = [
  "Start:-PUSH-0";
  "PUSH-99";
  "JMP-Loop";
  "Loop:-DUP";
  "PUSH-0";
  "CMP";
  "JNZ-Exit";
  "SWP";
  "OVER";
  "L0:-ADD";
  "SWP";
  "PUSH-1";
  "SUB";
  "JMP-Loop";
  "Exit:-POP";
  "RETURN";
]
```

e a execução e teste do programa deve ser feito com a chamada:

```
let _ = assert (run program "Start" = 4950)
```

Num mundo ideal, a resposta seria 42, mas não estamos num mundo ideal.

4 Critérios de avaliação

Para além da correção do programa, obtida através da execução de testes automáticos, a avaliação do trabalho terá em conta a qualidade do código, a clareza da implementação e a documentação do código.

Em relação à qualidade do código é de notar os fatores da representação das instruções como um tipo de dados algébrico, a representação dos blocos básicos numa estrutura de dados apropriada, a utilização de módulos para isolar partes do interpretador (e.g.a pilha de execução) e a estrutura *tail recursive* da função de execução.

5 Testes e Entrega

No repositório estão um conjunto de testes que exercitam o interpretador com programas de diferentes complexidades. Para executar os testes, basta correr o comando `dune runtest`. É importante usarem o repositório *git* como meio de trabalho de forma a garantirem que todos os testes passam antes da data de submissão do trabalho.

6 Data de entrega

O prazo de entrega é o dia 17 de Maio às 23:59. A entrega deve ser feita através do github classroom, com um push no repositório que for criado para o efeito quando aceitar o *assignment* disponível online em link a disponibilizar brevemente.