Exercícios para tutoria. Semana 6.

Programação Funcional

Prof. Rodrigo Ribeiro

Introdução

Máquinas virtuais são um mecanismo amplamente utilizado por compiladores para execução de código. Em especial, destaca-se a Java Virtual Machine, utilizada para execução de programas da linguagem Java.

O objetivo deste trabalho é a implementação de um interpretador para uma máquina virtual simples, que chamaremos de *Small*. As próximas seções apresentam a sintaxe e a semântica de programas Small.

Sintaxe e semântica de programas Small

Para execução de programas, a máquina virtual small utiliza uma pilha, uma memória e um contador de instruções. O estado da máquina pode ser representado pelo seguinte tipo de dados.

```
newtype Name
  = Name String
    deriving (Eq, Ord, Show)

data VMState
  = VMState {
      pc     :: Int
    , stack  :: [Int]
    , memory :: [(Name, Int)]
    } deriving (Eq, Ord, Show)
```

O tipo Name representa nomes de variáveis e o tipo VMState a configuração atual da máquina. Os campos do tipo VMState possuem o seguinte significado:

- pc: Representa o contador de instruções, isto é, a próxima instrução a ser executada pela máquina.
- stack: Representa a pilha de valores utilizados durante a execução de operações.
- memory: Armazena os valores associados a variáveis presentes no programa.

Programas small consistem de uma lista de instruções. A máquina pode executar somente 9 tipos de instruções, a saber:

- push(n): insere n no topo da pilha.
- var(x): insere o valor da variável x (contida na memória) no topo da pilha.
- set(x): atribui a variável x o valor do topo da pilha.
- add: soma os valores do topo da pilha e empilha o resultado.
- sub: subtrai os valores do topo da pilha e empilha o resultado.
- jump(n): desvio incondicional.
- jumpeq(n): desvia, se os 2 valores do topo da pilha são iguais.
- jumpneq(n): desvia, se os 2 valores do topo da pilha são diferentes.

Implementação de small

Representamos instruções small pelo tipo `Instr` a seguir:

```
data Instr
  = IPush Int
  | IVar Name
  | ISet Name
  | IAdd
  | ISub
  | IJump Int
  | IJumpNeq Int
  | IJumpEq Int
  | IHalt
  deriving (Eq, Ord, Show)
```

1 - A instrução push(n) insere o valor inteiro n no topo da pilha presente no tipo de dados `VMState`. Para implementar a semântica dessa instrução, implemente a função:

```
push :: Int -> VMState -> VMState
push = undefined
```

que altera o estado atual da máquina empilhando o inteiro fornecido como primeiro parâmetro.

2 - A instrução var(x) armazena o valor associado a variável x no topo da pilha de execução. Implemente essa funcionalidade na função:

```
lookMemory :: Name -> VMState -> VMState
lookMemory = undefined
```

Note que essa função deve retornar o estado da máquina alterado após a modificação da pilha de execução contendo, em seu topo, o valor associado a variável x.

3 - A instrução set(x) atribui à variável x o valor atualmente contido no topo da pilha de execução. Implemente essa funcionalidade na função:

```
setMemory :: Name -> VMState -> VMState
setMemory = undefined
```

4 - As operações add e sub utilizam os dois elementos do topo da pilha e armazenam o resultado da operação na própria pilha. Ao invés de implementarmos duas operações para a execução dessas instruções, vamos utilizar uma função de ordem superior que recebe como parâmetro a operação a ser aplicada sobre os elementos do topo da pilha. Implemente a função:

```
stackOp :: (Int -> Int -> Int) -> VMState -> VMState
stackOp op st
  = undefined
```

que aplica a operação fornecida como primeiro parâmetro aos dois primeiros elementos da pilha e re-insere o resultado retornando o estado da máquina alterado.

5 - Após a execução de cada instrução, o contador de programa deve ser incrementado. Além disso, a máquina small possui instruções de desvio, cujo principal objetivo é alterar o contador de programa para alterar o fluxo de execução do código. Pode-se alterar o contador de programa adicionando um valor inteiro a este. Implemente a função:

```
addPC :: Int -> VMState -> VMState
addPC = undefined
```

que adiciona ao valor atual do contador de instrução a constante inteira fornecida como primeiro parâmetro.

6 - Desvios condicionais devem ser implementados analisando os primeiros dois elementos da pilha (caso esses existam). Novamente, vamos nos valer de funções de ordem superior para tratamento dos dois tipos de desvio. Implemente a função:

```
condJump :: (Int -> Int -> Bool) -> Int -> VMState -> VMState
condJump = undefined
```

que a partir de um teste (igualdade ou desigualdade), um deslocamento e um estado atual, retorna o novo estado da máquina contendo o contador de instrução devidamente alterado (usando o deslocamento caso o teste seja verdadeiro ou incrementado, caso contrário).

7 - De posse de todas as implementações anteriores, a definição da execução de uma instrução da máquina pode ser implementada pela seguinte função:

```
vmStep :: Instr -> VMState -> VMState
vmStep = undefined
```

que a partir de uma instrução a ser executada e do estado atual da máquina produz o um novo estado resultante.

8 - Uma etapa importante da máquina é a busca da próxima execução a ser executada. Implemente a função

```
nextInstr :: [Instr] -> VMState -> Maybe Instr
nextInstr = undefined
```

que a partir de um programa e o estado atual da máquina, retorna a instrução apontada pelo contador de instruções. Caso o contador indique uma posição inválida ou a instrução atual seja `halt`, o valor `Nothing` deve ser retornado.

9 - Utilizando a função `nextInstr`, podemos implementar o interpretador da máquina small utilizando a seguinte função:

```
exec :: [Instr] -> VMState -> VMState
exec = undefined
```

que deve obter a próxima instrução a ser executada e continuar a execução do programa sobre o estado da máquina alterado pela última instrução. A função deve parar apenas quando a função `nextInstr` retornar `Nothing`, quando o estado final da máquina é retornado como resultado.