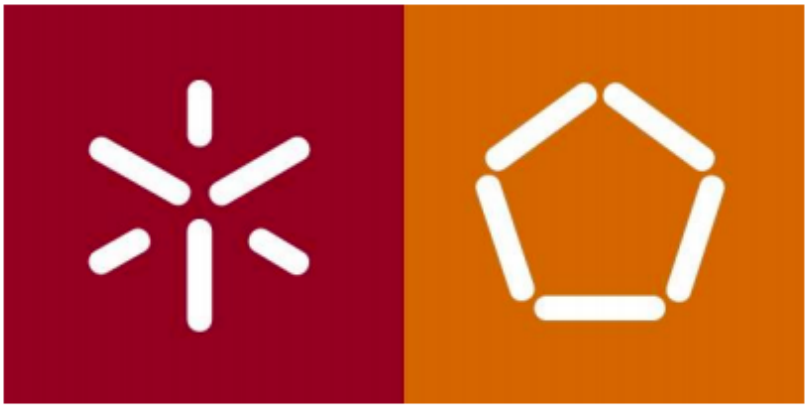

Universidade do Minho

Mestrado Integrado em Engenharia Informática

20/06/2021

Verificação Formal
Trabalho Prático 3

António Gonçalves (A85516)
Gonçalo Esteves (A85731)

Contents

1 Introdução

Para este trabalho foi-nos proposto que, utilizando *Coq* ou *Why3*, criássemos uma espécie de programa que se comporte como tradutor entre expressões algébricas e código máquina, mais especificamente, instruções de tratamento de uma *stack*. Posto isto, existem duas partes principais para este problema: uma primeira seria considerar que estas expressões apenas têm números inteiros, a operação do simétrico de um número, e as operações de soma, subtração e multiplicação; uma segunda versão deveria também aceitar expressões como variáveis. Para além disso deverá também lidar com a ideia de valoração de expressões.

Assim, a resolução é dividida em 3 partes: A primeira, onde estão os exercícios referentes às expressões, simples e com variáveis, a segunda, onde teremos o processamento referente ao funcionamento da nossa *stack*, e finalmente o compilador em si. Para além disso, ainda existem algumas funcionalidades extra, que funcionaram como valorização do trabalho final.

Iremos neste relatório explicar a nossa resolução para o trabalho, sendo que este foi resolvido usando *Why3*. De salientar que os excertos de código apresentados mais à frente são referentes a uma implementação final do código, pelo qual já têm informação referente às extensões ao problema inicial.

2 Parte 1 - Expressões com e sem Variáveis

Começando por explicar como é que iremos apresentar as expressões propriamente ditas, criamos dois tipos diferentes, *aexp*, que irá representar as expressões aritméticas, e *oper*, que irá representar as operações que poderemos utilizar dentro das expressões. Decidimos também adicionar outras opções, como a igualdade (*Igu*), e as ideias de maior e menor (*Sup* e *Inf*, respetivamente). Para além disso, permitimos também que existam conjunções e disjunções de expressões (*Econ* e *Edis*, respetivamente), usando à posteriori um novo tipo, *comb*, para o auxílio no cálculo do valor de expressões. Temos de seguida a definição destes tipos:

```
type oper = Adi
          | Sub
          | Mul
          | Igu
          | Sup
          | Inf

type comb = Conj
          | Disj

type aexp = Vcon int
          | Vsim aexp
          | Evar string
          | Ebin aexp oper aexp
          | Vneg aexp
          | Econ aexp aexp
          | Edis aexp aexp
```

Em relação às operações, temos então os 6 tipos de operações permitidas (adição, subtração, multiplicação, igualdade, maior e menor), bem como sete tipos de expressões: as constantes (*Vcon*); as simétricas (*Vsim*); as variáveis (*Evar*), que irão representar cada variável como sendo uma *String*; as binárias (*Ebin*), que irão relacionar duas expressões através de um operador do tipo *oper*; as negações (*Vneg*); as conjunções (*Econ*), que irão calcular a conjunção de duas expressões; e, por fim, as disjunções (*Edis*), que irão calcular a disjunção entre duas expressões. Os últimos três tipos, independentemente dos valores das expressões que recebem, irão retornar apenas 0 ou 1, consoante a expressão é falsa ou verdadeira (para feitos práticos, considera-se que se o valor associado a uma sub-expressão é 0, então a expressão é falsa, e caso contrário, a expressão é verdadeira).

De forma a conseguir representar o estado, criamos um novo tipo *estado* representado por *map string int*. Assim conseguimos associar o inteiro pretendido à *string* que representa uma variável. Para além disso, foram criadas duas funções, a *init* e a *update*, que têm como objetivo inicializar o estado do programa e atualizar o valor de uma dada variável de um estado.

```
type estado = map string int

function init : estado = const 0

function update (s: estado) (y: string) (n: int) : estado = set s y n
```

Com isto passamos então à definição de duas funções, sendo que a primeira recebe dois inteiros e um operador, e retorna o valor da aplicação do operador sobre os dois inteiros, e a segunda recebe dois inteiros e um combinador lógico, e retorna o valor da aplicação do combinador sobre os inteiros. Quanto à última função, numa primeira fase definimo-la de forma a executar o curto-circuito, como foi proposto no enunciado, no entanto acabamos por defini-la como se apresenta, de forma a facilitar a prova dos *lemmas* definidos posteriormente:

```
function aeval_ebin (n1: int) (op: oper) (n2: int) : int =
  match op with
    | Adi -> n1 + n2
    | Sub -> n1 - n2
    | Mul -> n1 * n2
    | Igu -> if (n1 = n2) then 1 else 0
    | Sup -> if (n1 > n2) then 1 else 0
    | Inf -> if (n1 < n2) then 1 else 0
  end

function leval_ebin (n1: int) (c: comb) (n2: int) : int =
  match c with
    | Conj -> if (n1 = 0) then 0
              else if (n2 = 0) then 0
                   else 1
    | Disj -> if (n1 <> 0) then 1
              else if (n2 <> 0) then 1
                   else 0
  end
```

Da mesma forma, criamos também as funções que retornam o valor simétrico de uma expressão, ou a negação:

```
function simet (n: int) : int = -n

function neg (n: int) : int = if (n <> 0) then 0 else 1
```

Posto isto, temos tudo o que necessitamos para criar a função *aeval*, que irá calcular o valor de uma expressão, sendo esta apresentada de seguida:

```
function aeval (e: aexp) (s: estado) : int =
  match e with
    | Vcon n         -> n
    | Vsim ex        -> simet (aeval ex s)
    | Evar x         -> get s x
    | Ebin e1 op e2 -> aeval_ebin (aeval e1 s) op (aeval e2 s)
    | Vneg ex        -> neg (aeval ex s)
    | Econ e1 e2     -> leval_ebin (aeval e1 s) Conj (aeval e2 s)
    | Edis e1 e2     -> leval_ebin (aeval e1 s) Disj (aeval e2 s)
  end
```

Para além disto, criamos também um predicado que determina se um dado valor é o resultado de uma expressão:

```
inductive aevalR aexp estado int =
  | vcon : forall s: estado, n: int.
    aevalR (Vcon n) s n
  | vsim : forall s: estado, e: aexp, n: int.
    (aevalR e s n) -> (aevalR (Vsim e) s (simet n))
  | evar : forall s: estado, x: string.
    aevalR (Evar x) s (get s x)
  | ebin : forall e1 e2: aexp, op: oper, s: estado, n1 n2: int.
    ((aevalR e1 s n1)/\(aevalR e2 s n2)) -> (aevalR (Ebin e1 op e2) s (aeval_ebin n1 op n2))
  | vneg : forall s: estado, e: aexp, n: int.
    (aevalR e s n) -> (aevalR (Vneg e) s (neg n))
  | econ : forall e1 e2: aexp, s: estado, n1 n2: int.
    ((aevalR e1 s n1)/\(aevalR e2 s n2)) -> (aevalR (Econ e1 e2) s (leval_ebin n1 Conj n2))
  | edis : forall e1 e2: aexp, s: estado, n1 n2: int.
    ((aevalR e1 s n1)/\(aevalR e2 s n2)) -> (aevalR (Edis e1 e2) s (leval_ebin n1 Disj n2))
```

Por fim, criamos vários exemplos de avaliação utilizando a nossa resolução, que poderão ser analisados no ficheiro enviado com o trabalho, bem como a prova de que o predicado *aevalR* e a função *aeval* estão em concordância, ou seja, se n é o valor retornado por *aeval* para uma dada expressão, então *aevalR* deverá validar a associação dessa mesma expressão a n.

```
lemma func_rela_conc:
  forall e: aexp, s: estado, n: int. aevalR e s n <-> aeval e s = n
```

3 Parte 2 - Máquina de *Stack*

Iremos agora criar a linguagem que será utilizada para exprimir as nossas expressões em formato máquina. Uma vez que há vários tipos de operações diferentes, tornou-se necessário definir uma cláusula para cada tipo de operação. Assim sendo, *SPush* irá representar a inserção de um elemento na *stack*, *SLoad* irá representar a inserção do valor de uma variável na *stack*, *SSim* irá representar a substituição do valor no topo da *stack* pelo seu simétrico, *SPlus* irá representar a substituição dos dois primeiros valores na *stack* pela sua soma, *SMinus* irá representar a substituição dos dois primeiros valores na *stack* pela sua diferença, *SMult* irá representar a substituição dos dois primeiros valores na *stack* pelo seu produto, *SIgu* irá representar a substituição dos dois primeiros valores na *stack* pela sua comparação (1 caso sejam iguais, 0 o contrário), *SSup* irá representar a substituição dos dois primeiros valores na *stack* pela sua comparação (1 caso o primeiro seja superior ao segundo, 0 o contrário), *SInf* irá representar a substituição dos dois primeiros valores na *stack* pela sua comparação (1 caso o primeiro seja inferior ao segundo, 0 o contrário), *SNeg* irá representar a substituição do valor no topo da *stack* pelo seu oposto, *SConj* irá representar a substituição dos dois primeiros valores na *stack* pela sua conjunção, *SDisj* irá representar a substituição dos dois primeiros valores na *stack* pela sua disjunção.

Começamos então por criar um tipo novo, *opstack*, que irá representa as várias operações possíveis.

```
type opstack = SPush int
             | SLoad string
             | SSim
             | SPlus
             | SMinus
             | SMult
             | SIgu
             | SSup
             | SInf
             | SNeg
             | SConj
             | SDisj
```

Tendo os tipos de operações diferentes definidos, passamos à definição da função *execute*. Esta irá receber um estado, uma *stack*, que será representada por uma lista de números sendo a cabeça da lista considerada como o topo, e uma lista de instruções, representada por uma lista de *opstack*, que irá formar o programa pretendido. Esta função irá no fim retornar a *stack* passada como parâmetro tendo em conta as alterações feitas pela execução do programa.

```
function execute (s: estado) (st: list int) (p: list opstack) : (list int) =
  match p with
    | Nil      -> st
    | Cons h t -> match h with
      | SPush n -> execute s (Cons n st) t
      | SLoad x -> execute s (Cons (get s x) st) t
      | SSim    -> execute s (Cons (simethd st) (tl st)) t
      | SPlus   -> execute s (Cons (aeval_ebin (hd st) Adi (hdtl st)) (tltl st)) t
      | SMinus  -> execute s (Cons (aeval_ebin (hd st) Sub (hdtl st)) (tltl st)) t
      | SMult   -> execute s (Cons (aeval_ebin (hd st) Mul (hdtl st)) (tltl st)) t
      | SIgu    -> execute s (Cons (aeval_ebin (hd st) Igu (hdtl st)) (tltl st)) t
      | SSup    -> execute s (Cons (aeval_ebin (hd st) Sup (hdtl st)) (tltl st)) t
      | SInf    -> execute s (Cons (aeval_ebin (hd st) Inf (hdtl st)) (tltl st)) t
      | SNeg    -> execute s (Cons (neghd st) (tl st)) t
      | SConj   -> execute s (Cons (leval_ebin (hd st) Conj (hdtl st)) (tltl st)) t
      | SDisj   -> execute s (Cons (leval_ebin (hd st) Disj (hdtl st)) (tltl st)) t
    end
  end
```

Assim, passamos a conseguir consumir recursivamente as operações passadas na lista de *opstack*, adicionando o resultado da execução das mesmas na cabeça da *stack* em si. Podemos também constatar que não definimos forma de impedir que as operações executem sobre uma *stack* inválida, no entanto, considerando que esta função irá ser utilizada posteriormente sobre expressões convertidas em linguagem de máquina de *stack*, desde que essa conversão seja bem definida e garanta, por exemplo, que uma adição apenas é efetuada após a inserção de dois elementos na *stack*, o correto funcionamento da função *execute* é também garantido.

Definimos mais alguns testes simples, para comprovar que esta implementação se encontra correta, podendo estes ser mais uma vez consultados no ficheiro com a implementação final.

4 Parte 3 - O Compilador

Nesta última parte do trabalho iremos dotar o programa da capacidade de compilar expressões para linguagem de máquina de *stack* por nós definida anteriormente.

Desta forma, criamos a função *compile* que recebe uma expressão e a converte na respetiva lista de operações de *stack*. Nesta, as principais questões a ter em conta relacionam-se com a forma como são organizadas as instruções para a *stack*. Assim, para expressões que envolvem uma outra sub-expressão, calcula-se primeiro as *opstack* relativas à sub-expressão, e só de seguida se adiciona ao fim da lista a operação relativa à expressão principal; no caso de expressões que envolvam duas sub-expressões, determina-se primeiro as operações relativas à expressão da esquerda, em seguida concatena-se as operações relativas à expressão da direita, e só no fim se adiciona a operação relativa à expressão principal, mais uma vez, no fim da lista.

Tendo tudo isto em consideração, chegamos à seguinte definição para a função *compile*:

```
function compile (e: aexp) : (list opstack) =
  match e with
    | Vcon n        -> Cons (SPush n) Nil
    | Vsim ex       -> (compile ex) ++ (Cons SSim Nil)
    | Evar x        -> Cons (SLoad x) Nil
    | Ebin e1 op e2 -> (compile e1) ++ ((compile e2) ++ (Cons (convert op) Nil))
    | Vneg ex       -> (compile ex) ++ (Cons SNeg Nil)
    | Econ e1 e2    -> (compile e1) ++ ((compile e2) ++ (Cons SConj Nil))
    | Edis e1 e2    -> (compile e1) ++ ((compile e2) ++ (Cons SDisj Nil))
    end
```

Posto isto, criamos algumas pequenas provas que serviram para comprovar a validade da função em exemplos concretos, bem como dois outros *lemmas*, apresentados em seguida, que servem para validar a construção da função:

```
lemma comp_corr_aux :
  forall s: estado, e: aexp, st: list int, p: list opstack.
    execute s st ((compile e) ++ p) = execute s (Cons (aeval e s) st) p

lemma comp_corr :
  forall s: estado, e: aexp. execute s Nil (compile e) = Cons (aeval e s) Nil
```

O primeiro *lemma* consiste na garantia de que a execução de um programa, cuja primeira parte se refere a uma expressão, é a mesma coisa que executar apenas o resto do programa, considerando que a *stack* já tem inserida à cabeça o valor da expressão. Este *lemma* torna-se essencial para assegurar a provar do segundo, que nos permite concluir que as funções *execute* e *compile* estão bem construídas e podem ser utilizadas em conjunto.

5 Conclusão

Acreditamos ter conseguido atingir o objetivo deste trabalho, passando este pela criação de um programa capaz de compilar expressões para linguagem máquina e em seguida executar as operações geradas, de forma a controlar uma *stack* e utilizando tipos por nós definidos.

Conseguimos responder a todas as alíneas principais enumeradas no enunciado, não tendo apenas sido capazes de comprovar a validade do *lemma* auxiliar construído e utilizado na última parte. Implementamos também uma das extensões propostas no enunciado, permitindo ao programa a interpretação de expressões booleanas bem como de operadores relacionais.

Por fim, fomos capazes de aprofundar os nossos conhecimentos sobre *Why3*, não só quanto ao uso da ferramenta em si, mas também quanto à utilidade da mesma, tendo analisado com um exemplo concreto a relativa facilidade com que a mesma garante a validade de programas, auxiliando assim numa construção mais segura dos mesmos.