

Verificação Formal (2019/20) - Projecto Opcional

Este projecto poderá ser desenvolvido em duas modalidades:

- individualmente, em **Why3** ou em **Coq**;
- em grupos de dois, neste caso, em **Why3** e em **Coq**.

No final do projecto deverão entregar os ficheiros com a sua resolução e um pequeno relatório explicativo da sua solução.

O projecto deverá ser entregue até **15 de Julho**, via Blackboard. Na semana seguinte, entre os dias **20 e 24 de Julho**, serão depois agendadas as apresentações dos projectos.

Descrição do problema

Certas linguagens de programação e máquinas abstractas avaliam expressões aritméticas usando uma *stack*.

Por exemplo, a expressão $\sim((5*3)+(3*(4-2)))$ seria escrita `5 3 * 3 4 2 - * + ~` (formato *post-order*) e avaliada, com o auxílio de uma *stack*, assim:

STACK		Expressão
[]		5 3 * 3 4 2 - * + ~
[5]		3 * 3 4 2 - * + ~
[3, 5]		* 3 4 2 - * + ~
[15]		3 4 2 - * + ~
[3, 15]		4 2 - * + -
[4, 3, 15]		2 - * + -
[2, 4, 3, 15]		- * + ~
[2, 3, 15]		* +
[6, 15]		+
[21]		~
[-21]		

O objectivo deste exercício é escrever o backend de um pequeno compilador que traduza expressões aritméticas em instruções de uma máquina de *stack*. Note que o nosso ponto de partida é a árvore de sintaxe de uma expressão e não a sua representação textual.

Apresentamos duas versões do problema:

1. Uma versão, mais simples, que lida só com expressões sem variáveis. Uma expressão poderá ter apenas números inteiros, a operação unária simétrico e as operações binárias: adição, subtração e multiplicação.
2. Uma versão, mais desafiante, que aceita expressões com variáveis. Neste caso, uma expressão poderá ser: um número inteiro, uma variável, o simétrico de uma expressão ou uma adição, subtração ou multiplicação de expressões. Neste caso é preciso lidar com a noção de estado (ou valoração), que indica o valor associado a cada variável.

Expressões sem variáveis

1. Comece por definir um tipo indutivo, `aexp`, para representar a sintaxe abstracta das expressões.
2. Defina uma função de avaliação das expressões, `aeval`, que recebe uma expressão e produz um número.
3. Faça alguns exemplos de avaliação de expressões concretas.
4. Defina agora a avaliação de uma expressão como uma relação, `aevalR`, entre expressões aritméticas e números.
5. Prove agora que a definição relacional e funcional de avaliação concordam, isto é,

$$\forall a n, (\text{aevalR } a n) \Leftrightarrow (\text{aeval } a) = n$$

Expressões com variáveis

Nesta versão vamos enriquecer as expressões aritméticas com variáveis. Para avaliar estas expressões precisamos de lidar com a noção de *estado* (ou *valoração*) que representa os valores actuais das variáveis.

Para simplificar, assumimos que o estado é definido para todas as variáveis, mesmo que a expressão apenas mencione algumas. Dado que cada variável vai ter associado um número, podemos representar o estado como um mapeamento de `string` para `int` e usar 0 como valor por omissão.

Será também útil ter uma representação do estado inicial *init* (onde todas as variáveis mapeiam em 0) e uma função de *update* do estado que dado um estado s , uma variável y e um número n , devolve o estado $s[y \mapsto n]$ onde y está associado a n e as restantes variáveis ao valor que já tinham em s . Isto é,

$$(s[y \mapsto n]) x = \begin{cases} n & \text{se } x = y \\ s x & \text{se } x \neq y \end{cases}$$

1. Comece por adicionar variáveis às expressões aritméticas que tínhamos antes, adicionando ao tipo indutivo `aexp` mais um construtor (para o caso da expressão ser uma variável).
2. A função de avaliação das expressões, `aeval`, é agora estendida para manipular variáveis (da maneira óbvia), considerando o estado como um argumento extra.
3. Faça alguns exemplos de avaliação de expressões em estados concretos.
4. A definição da avaliação de uma expressão como uma relação, `aevalR`, é agora estendida para manipular variáveis (da maneira óbvia), e passa a relacionar expressões aritméticas, estados e números.
5. Prove agora que a definição relacional e funcional de avaliação concordam, isto é,

$$\forall a s n, (\text{aevalR } a s n) \Leftrightarrow (\text{aeval } a s) = n$$

A máquina de stack

A máquina de stack terá o seguinte conjunto de instruções:

- S**Push *n*: coloca *n* no topo da stack.
- S**Load *x*: carrega o valor associado no estado à variável *x* para o topo da stack. (Caso não lide com variáveis, pode ignorar esta instrução.)
- S**Sim: retira o número que está no topo da stack e coloca o seu simétrico no topo da stack.
- S**Plus: retira os dois números que estão no topo da stack, adiciona-os e coloca o resultado no topo da stack.
- S**Minus: similar, mas subtrai.
- S**Mult: similar, mas multiplica.

1. Escreva uma função, **execute**, para avaliar programas na linguagem da máquina de stack. A função deve ter como entrada um estado, uma stack representada como uma lista de números (o topo da stack é a cabeça da lista) e um programa representado como uma lista de instruções, e deve retornar a stack após a execução do programa.
2. Teste a sua função com alguns programas concretos.
3. Observe que a especificação não indica o que fazer ao encontrar uma instrução **S**Plus, **S**Minus ou **S**Mult se a pilha contiver menos de dois elementos. Em certo sentido, é irrelevante o que fazemos, já que o nosso compilador nunca produzirá um programa mal formado.

O compilador

Vamos agora escrever o compilador e provar a sua correcção face à semântica de avaliação.

1. Escreva uma função, **compile**, que faz a compilação de uma expressão aritmética num programa da máquina de stack. O efeito de executar o programa deve ser o mesmo que colocar o valor da expressão no topo da stack.
2. Teste a sua função com um exemplo concreto.
3. Finalmente, tente demonstrar a correcção do compilador que implementou. Isto é, prove o seguinte teorema¹

$$\forall s a, \text{execute } s [] (\text{compile } a) = [(a\text{eval } s a)]$$

É natural que precise de declarar um lema mais geral para obter uma hipótese de indução utilizável. O teorema virá depois como um simples corolário desse lemma.

¹Caso não lide com variáveis, o teorema de correcção a provar será simplesmente

$$\forall a, \text{execute } [] (\text{compile } a) = [(a\text{eval } a)]$$

Extensões ao problema inicial

Tendo completado a etapa anterior, poderá agora estender a sua implementação dos seguintes modos:

1. Expandir o compilador para lidar também com expressões booleanas, com as habituais conectivas lógicas negação, conjunção e disjunção, e os operadores relacionais sobre inteiros $=$ e $<$.

A avaliação da conjunção e da disjunção deve ser feita em curto-circuito, isto é, na avaliação da expressão $b_1 \wedge b_2$ (resp. $b_1 \vee b_2$) primeiro avalia-se b_1 e se o resultado for falso (resp. verdade), então toda a expressão avalia imediatamente para falso (resp. verdade). Caso contrário b_2 é avaliado para determinar o resultado da expressão.

2. Expandir o compilador para lidar também com expressões com efeitos laterais, nomeadamente, acrescentado às expressões aritméticas os operadores $++$ e $--$ que podem ser aplicados às variáveis, com o efeito adicional de alterar o estado (semelhante ao efeito de avaliar $x++$, $++x$, $x--$ e $--x$ na linguagem C. Neste caso a avaliação das expressões deverá ser feita sempre da esquerda para direita.