Algebraic Data Types

Árvores de pesquisa

Nos exercícios seguintes considere a definição do tipo de árvores de pesquisa apresentada na aulas teórica:

$$data\ Arv\ a = Vazia\ \mid\ No\ a\ (Arv\ a)\ (Arv\ a)$$

4.1 Escreva uma definição recursiva da função $sumArv :: Num\ a \Rightarrow Arv\ a \rightarrow a$ que soma todos os valores duma árvore binária de números.

4.2 Baseado-se na função $listar :: Arv\ a \rightarrow [a]$ apresentada na aula teórica, escreva a definição duma função para listar os elementos duma árvore de pesquisa por *ordem decrescente*.

4.3 Escreva uma definição da função $nivel :: Int \rightarrow Arv\ a \rightarrow [a]$ tal que $nivel\ n\ arv$ é a lista ordenada dos valores da árvore no nível n, isto é, a uma altura n (considerando que a raiz tem altura 0).

4.4 Experimente usar o interpretador de Haskell para calcular a altura de árvores de pesquisa com n valores.

(a) usando o método de partições binárias, e.g. $construir\ [1..n]$;

(b) usando inserções simples, e.g. $foldr\ inserir\ Vazia\ [1..n]$;

Experimente com $n = 10$, 100 e 1000 e compare a altura obtida com o minorante teórico: uma árvore binária com n nós tem altura $\geq \log_2 n$.

4.5 Escreva uma definição da função de ordem superior $mapArv :: (a \rightarrow b) \rightarrow Arv\ a \rightarrow Arv\ b$ tal que '$mapArv\ f\ t$' aplica uma função f a cada valor duma árvore t.

4.6 Neste exercício pretende-se implementar uma variante da remoção de um valor duma árvore de pesquisa simples.

(a) Baseando-se na função $mais\_esq :: Arv\ a \rightarrow a$ apresentada na aula teórica, escreva uma definição da função $mais\_dir :: Arv\ a \rightarrow a$ que obtém o valor mais à direita numa árvore (i.e., o maior valor).

(b) Usando a função da alínea anterior, escreva uma definição alternativa da função $remover :: Ord\ a \Rightarrow a \rightarrow Arv\ a \rightarrow Arv\ a$ que use o valor mais à direita da sub-árvore esquerda no caso de um nó com dois descentes não-vazios.

Expressions

4.7 Consider the following data type for expressions:

```
data Expr = Lit Integer |
            Op Ops Expr Expr

data Ops = Add | Sub | Mul | Div
```

Define functions eval, show and size for this type, and discuss the changes you have to make to your definitions if you add the extra operation Mod for remainder on integer division.

4.8 It is possible to extend the type Expr so that it contains conditional expressions, If b e1 e2, where e1 and e2 are expressions, and b is a Boolean ex- pression, a member of the type BExp,

```
data Expr = Lit Integer |
            Op Ops Expr Expr |
            If BExp Expr Expr
```

The expression If b e1 e2 has the value of e1 if b has the value True and otherwise it has the value of e2.

```
data BExp = BoolLit Bool |
            And BExp BExp |
            Not BExp |
            Equal Expr Expr |
            Greater Expr Expr
```

The five clauses here give Boolean literals, BoolLit True and BoolLit False. The conjunction of two expressions; it is True if both sub-expressions have the value True. The negation of an expression. Not be has value True if be has the value False. Equal e1 e2 is True when the two numerical expressions have equal values. Greater e1 e2 is True when the numerical expression e1 has a larger value then e2.

Define the functions

1. $eval :: Expr \rightarrow Integer$
2. $bEval :: BExp \rightarrow Bool$ by mutual recursion, and extend the function show to show the redefined type of expressions.