University of Amsterdam
Computer Systems Architecture
Dr Clemens Grelck

Functionele Talen
2009/2010
February 23, 2010

Assignment Series 4

Algebraic Data Types

Assignment 8: Binary trees

Consider the following type definition of a binary tree:

```
type 'a bintree = Empty
                | Leaf of 'a
                | Node of 'a bintree * 'a * 'a bintree
```

Implement the following functions on binary trees:

a) `height t`
that yields the height of the given binary tree `t`, i.e. the length of the longest path from the root of the tree to any leaf;

b) `size t`
that yields the size of the given binary tree `t`, i.e. the number of data items stored;

c) `normalise t`
that normalises the given binary tree `t`, i.e. removes all occurrences of the `Empty` constructor in non-root positions while mimicking the original tree structure as close as possible;

d) `fringe t`
that yields the fringe of the given binary tree `t` as a list of all the data elements stored in leaf positions of the tree.

Assignment 9: Generalised trees

Generalised trees differ from binary trees in that each parent node may have any number of child nodes, including none at all.

a) Define a polymorphic algebraic data type `gentree` to represent generalised trees.

b) Implement a function `bintree2gentree` that converts a given binary tree into an equivalent generalised tree.

c) Implement a function `gentree2bintree` that converts a given generalised tree into an equivalent binary tree. Equivalence here shall mean that each level of the generalised tree is converted into a balanced binary tree, where balanced means that the difference in length between the shortest and the longest path through the tree is at most 1.

d) Prove by structural induction that the functional composition
$$\texttt{gentree2bintree} \circ \texttt{bintree2gentree}$$
yields the identity function on binary trees.

Assignment 10: Expression language

Algebraic data types are well suited to represent computer programs. Define an algebraic data type `expr` that allows you to represent programs of the following simple O'Caml-like expression language defined in Backus-Naur form as an abstract syntax tree. Constants are to be adopted directly from the O'Caml language.

Expr	⇒	`(` *Expr* `)`
	|	*Expr BinOp Expr*
	|	*MonOp Expr*
	|	*Id*
	|	*Const*
BinOp	⇒	*ArithOp* | *RelOp* | *LogicOp*
ArithOp	⇒	`+` | `-` | `*` | `/` | **mod**
RelOp	⇒	`=` | `<>` | `<` | `<=` | `>` | `>=`
LogicOp	⇒	`&&` | `||`
MonOp	⇒	`-` | `not`
Const	⇒	*BoolConst*
	|	*IntConst*

Assignment due date: March 2, 2010, beginning of lecture