HW 13 **Prof. Caldwell**
Due: 20 October 2011 **COSC 3015**

1 Abstract Data Types in Haskell

In class we talked about how to use the Haskell module system to implement an abstract data type.

An *abstract data type* (ADT) provides users with the type signatures of the operations supported on the type together with an abstract specification of the expected behavior of the type. This specification takes the form of a list of axioms that relate the behaviors of the interactions of the operators with one another. By only allowing users to access the interface it is possible to change an underlying implementation without having to change code that uses the ADT.

Haskell's module system allows programmers to implement ADT's by providing a mechanism for hiding underlying representations and implementations and simply exporting the names and signatures of the interface.

Exercise 1.1. Read about Modules in Bird (pp.263-264).
Read about modules in LYAHFGG `http://learnyouahaskell.com/modules`.

1.1 An Abstract Data Type of Trees

For example, as in class we defined a data type of trees.

1.1.1 Type Signature

The type signatures of the tree operations was given as follows:

Tree a	– *the type name*
leaf	$:: a \to Tree\ a$
branch	$:: Tree\ a \to Tree\ a \to Tree\ a$
cell	$:: Tree\ a \to a$
left, right	$:: Tree\ a \to Tree\ a$
isLeaf	$:: Tree\ a \to Bool$

1.1.2 The Tree Axioms

If $x :: a$ and $t_1, t_2 :: (Tree\ a)$ then the following axioms specify the behaviors of the operators.

$$
\begin{array}{ll}
T1.) & cell(leaf\ x) = x \\
T2.) & cell(branch\ t_1\ t_2) = \perp \\
T3.) & left(leaf\ x) = \perp \\
T4.) & left(branch\ t_1\ t_2) = t_1 \\
T5.) & right(leaf\ x) = \perp \\
T6.) & right(branch\ t_1\ t_2) = t_2 \\
T7.) & isLeaf(leaf\ x) = True \\
T8.) & isLeaf(branch\ t_1\ t_2) = False
\end{array}
$$

1.1.3 Haskell Implementation

A Haskell implementation is given by the following module.

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where
data Tree a            = Leaf a | Branch (Tree a) (Tree a)  deriving (Eq,Show)
leaf                   = Leaf
branch                 = Branch
cell  (Leaf a)         = a
left  (Branch l r)     = l
right (Branch l r)     = r
isLeaf   (Leaf _)      = True
isLeaf   _             = False
```

Note that the constructors *Leaf* and *Branch* are not in export list following the modules name. In the absence of an export list, all declarations in the module are exported. Since there is an export list in the *TreeADT* module, definitions that are not mentioned in the list are private. Programs that import this module do not have access to *Leaf* and *Branch*. Note that if you load this module into GHCI, you *will* have access to the *Leaf* and *Branch* constructors – but those constructors are not in the name-space of modules that import *TreeADT*. Also, the deriving-clause which include the type *Tree a* in the *Eq* and *Show* type classes make it possible to test elements of the data type for equality and to display them. Displaying them reveals the names of the underlying constructors, but because they were not exported, users of the module can not use them.

1.1.4 Unit Tests

Axioms can be used to design unit tests for an implementation. This is especially easy for axioms that are stated in the form given above – where we assume the varialbes x and T_1 and t_2 are universally quantified in each axiom and the axiom takes the form of an equation. If the axioms contain existential quantifiers the problem is more difficult. In the *TreeADT* axioms *T1* through *T8*, the variables x, t_1 and t_2 that occur in each axiom become arguments to the test function for the behavior specified by that axiom. If the behavior is specified to be undefined *i.e.* it is equal to $\perp$, then the test can be run to see if it results in an error. Note that we have used $\perp$ to denote both run-time errors and looping forever. Obviously, testing if a program loops forever can not be done unless you are extremely patient and have unlimited time on your hands.

Here is a Haskell module implementing unit tests for the *TreeADT* module.

```
module TestTreeADT where

import TreeADT

test_T1 x = cell (leaf x) == x
– the following test should raise an error on all inputs t1 and t2
test_T2 t1 t2 = cell (branch t1 t2)
– the following test should raise an error on all inputs x
test_T3 x = left(leaf x)
test_T4 t1 t2 = left (branch t1 t2) == t1
– the following test should raise an error on all inputs x
test_T5 x = right(leaf x)
test_T6 t1 t2 = right (branch t1 t2) == t2
```

test_T7 x = isLeaf (leaf x) == True
test_T8 t1 t2 = isLeaf (branch t1 t2) == False

1.2 An Abstract Data Type for Sets

Here is a specification of a Set ADT. Note that our sets are *monomorphic* in the sense that they can contain elements of a single type.

1.2.1 Type Signature

data Set a	*– the type name*
empty	*:: Set a*
ismem	*:: a → Set a → Bool*
size	*:: Set a → Int*
insert	*:: a → Set a → Set a*
delete	*:: a → Set a → Set a*
union	*:: Set a → Set a → Set a*
intersection	*:: Set a → Set a → Set a*

1.2.2 The Set Axioms

If *s,t :: (Set a)* and $x, y :: a$ the following axioms specify the behavior for the Set data type.

$$\begin{array}{lrcl}
S1.) & ismem\ x\ empty & = & False \\
S2.) & ismem\ x\ (insert\ y\ s) & = & (x = y) \vee ismem\ x\ s \\
S3.) & size\ empty & = & 0 \\
S4.) & size(insert\ x\ s) & = & \begin{cases} size\ s & \text{if } ismem\ x\ s \\ (size\ s) + 1 & \text{otherwise} \end{cases} \\
S5.) & (insert\ x\ s = empty) & = & False \\
S6.) & insert\ x\ (insert\ x\ s) & = & insert\ x\ s \\
S7.) & insert\ x\ (insert\ y\ s) & = & insert\ y\ (insert\ x\ s) \\
S8.) & ismem\ x\ (delete\ y\ s) & = & \begin{cases} False & \text{if } x = y \\ ismem\ x\ s & \text{otherwise} \end{cases} \\
S9.) & ismem\ x\ (union\ s\ t) & = & (ismem\ x\ s\ \vee\ ismem\ x\ t) \\
S10.) & ismem\ x\ (intersection\ s\ t) & = & (ismem\ x\ s\ \wedge\ ismem\ x\ t)
\end{array}$$

Exercise 1.2. Your assignment is build on the base code provided on the course web-page to finish an implementation of the *Set* data type using lists. You also need to make a module *TestSet* which implements unit tests for each set axiom.

Note that you may need to use the list function *nub :: (Eq a) ⇒ [a] → [a]* which eliminates duplicate elements of a list. It requires the elements of the list to be members of the Eq type class.