Formal Specification of Abstract Datatypes
Exercise 4 (June 15)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

The result is to be submitted by the deadline stated above via the Moodle interface as a .zip or .tgz file which contains

- a PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - the formal specifications in the style of "Thinking Programs",
  - the CafeOBJ specifications,
  - comprehensive tests of the CafeOBJ specifications (sample reductions),
  - optionally any explanations or comments you would like to make;
- the CafeOBJ (.mod) file(s) of the specifications.

Exercise: Finite and Infinite Trees

In set theory, a *tree* is a collection of finite sequences (called *paths*) such that every prefix of a sequence in the collection also belongs to the collection. The *empty tree* is the tree that has no paths. A *finite tree* is a tree that has only finite paths. A (completely) *infinite tree* is a tree that has for every path p a longer path p' with prefix p. A *binary tree* is a tree which contains for every path p of length n not more than two paths p' of length $n + 1$ whose prefix is p. A *labeled tree* is a tree that assigns to every path a label (also called *value*).

The goal of this exercise is to formalize (finite and infinite) binary labeled trees.

Finite Trees First develop a formal specification of the domain of finite binary trees with labels of sort *Value* by a specification with (at least) the following entities:

```
spec FBTREE[sort Value] import NAT :=
free {
  type Tree = empty | node(Value,Tree,Tree)
  type Values = none | some(Value,Values)
  type Path = null | left(Path) | right(Path)
}
then ... {
  fun value: Tree → Value
  fun left: Tree → Tree
  fun right: Tree → Tree
  fun depth: Tree → Nat
  fun values: Tree × Nat → Values
  pred has ⊆ Tree × Path
  fun get: Tree × Path → Value
  fun put: Tree × Path × Value → Tree
  ...
}
```

Here *empty* denotes the empty tree while $node(v, t_1, t_2)$ denotes that non-empty tree in which every path p is a sequence of elements *left* and *right*: if p is empty, it is assigned the value v (the *root* of the tree); if p starts with *left*, the value assigned to p is the value that t_1 (the *left subtree*) assigns to the remainder of p; if p starts with *right*, the value assigned to p is the value that t_2 (the *right subtree*) assigns to the remainder of p.

The function $value(t)$ denotes the root of the non-empty tree, $left(t)$ denotes its left subtree, and $right(t)$ denotes its right subtree. If t is empty, then $depth(t) = 0$; otherwise $depth(t)$ is 1 plus the maximum of the lengths of all paths in t.

The function $values(t, n)$ constructs the sequence of those values to which all paths of length less than or equal n are assigned by t (thus, if $n = 0$ and t is not empty, the result contains only the root of t); the values are sorted in lexicographic order of the paths (*left* comes before *right*).

The predicate $has(t, p)$ determines whether p is a path of t; if this is the case, then $get(t, p)$ denotes the value to which p is assigned in t while $put(t, p, v)$ denotes a new tree that is identical to t except that p is assigned to v.

You may specify the operations in loose or in free semantics.

Infinite Trees Next develop a formal specification of the domain of infinite binary trees with (at least) the following entities:

```
spec IBTREE[sort Value] import Nat :=
cofree
  cotype Tree = value:Value | left:Tree | right:Tree
then free {
  type Values = none | some(Value,Values) and
  type Path = null | left(Path) | right(Path)
}
then ... {
  fun node: Value × Tree × Tree → Tree
  fun values: Tree × Nat → Values
  fun get: Tree × Path → Value
  fun put: Tree × Path × Value → Tree
  ...
}
```

The operations in this specification are interpreted as in the finite case. However, while in the finite case it was necessary to specify the behavior of the observers *value*, *left*, and *right*, it is now necessary to specify the behavior of the constructor *node* (which combines a value and two infinite trees to another infinite tree).

If the result of an operation is of type *Tree*, it can be uniquely defined by the values of the observers (e.g. $value(node(v, t_1, t_2)) = \ldots$).

You may specify the operations in loose or in free semantics.

CafeOBJ Implement the specification *FBTREE* of finite binary trees developed above in CafeOBJ by a tight CafeOBJ parameterized over the value domain.

```
module* Value { signature { [ Value ] } }
module! FBTREE(V :: Value)
{
  protecting (NAT)
  ...
}
```

Instantiate the module with module `NAT` as the domain of labels.

Test the CafeOBJ module with several reductions. Give the input and output of each test and your interpretation of the results (do they indicate errors in your specification or not?). If your specification contains errors, use the trace facilities of CafeOBJ for debugging.

Does your CafeOBJ module denote the same datatype as the one of the previous specification? Justify your answer.