Ordered-sorts and Error/Exceptions.

The general approach for treating errors using the order-sorted concept is that for every sort σ, we have two subsorts — σ -Ok < σ and σ -Error < σ (for greater generality we can admit multiple error subsorts to accommodate different kinds of errors). This treatment then requires that σ -Ok ∩ σ -Error = ∅ and σ -Ok ∪ σ -Error = σ for all sorts. If these conditions are met by the initial algebra, the specification is called **clean** (M. Gogoila, "On parametric algebraic specifications with clean error handling", *Proc. Conf. on Theory and Practice of Software Development (TAPSOFT '87)*, Springer-Verlag). Note that, as with the OK-functions, in order to treat errors in this way, it is necessary that all pre-defined and parameter ADTs follow the same approach.

This subsort arrangement effectively accomplishes the effect of the OK-functions suggested earlier, but without adding the functions, their equations, and without complicating all the normal-case equations with tests for troublesome arguments. It's all accomplished by the signature alone! Transferring the description of this behavior from the semantic domain of equations to the syntactic domain of signatures is a great simplification, and that is a very worthwhile contribution. It's not completely without cost, as we have some semantic subtleties that were not present with the OK-function strategy. None-the-less, the net gain in both simplicity and brevity is a rare achievement.

Example: Tree[Entry]
This example presents the familiar binary tree using the error sort paradigm outlined above. For specificity, it is assumed that each sort named X has a corresponding error subsort named X-Error, and a normal subsort named X-Ok. Each X-Error subsort has a value known as NonX (i.e., NonX: X-Error). The binary tree objects described here have entries only in the leaf nodes — the tree provides a hierarchical organization for these leaf entries, but non-leaf nodes contain no data.

SORTS
Tree-Error < Tree
Tree-Ok < Tree

SIGNATURES
NonTree: → Tree-Error
Leaf: Entry → Tree
Leaf: Entry-Ok → Tree-Ok
Node: Tree × Tree → Tree
Node: Tree-Ok × Tree-Ok → Tree-Ok
GetEntry: Tree → Entry
GetLeft, GetRight: Tree → Tree

EQUATIONS

For each e_1:Entry-Ok, e_2:Entry-Error, t:Tree, t_1,t_2: Tree-Ok

Leaf(e_2) = NonTree

Node(NonTree, t) = Node(t, NonTree) = NonTree

GetEntry(Leaf(e_1)) = e_1

GetEntry(Node(t_1,t_2)) = NonEntry

GetEntry(NonTree) = NonEntry

GetLeft(Leaf(e_1)) = GetRight(Leaf(e_1)) = NonTree

GetLeft(Node(t_1,t_2)) = GetRight(Node(t_2,t_1)) = t_1

GetLeft(NonTree)) = GetRight(NonTree)) = NonTree

EQUIVALENCE CLASSES

Constructors for this ADT are Leaf and Node — every binary tree can be formed with these operations.

[NonTree] = {NonTree, Leaf(NonEntry), Node(NonTree, NonTree),
GetLeft(Leaf(e_1)), … } sort is Tree-Error

[Leaf(e_1)] = [Leaf(e_1), GetLeft(Node(Leaf(e_1), Leaf(e_1))), … } sort is Tree-Ok

[Node(Leaf(e_1), Leaf(e_1))] =
{Node(Leaf(e_1), Leaf(e_1)), GetLeft(Node(Leaf(e_1), Leaf(e_1)), Leaf(e_1)), … }
sort is Tree-Ok

…

So there is one class in sort Tree-Error, and all others are in sort Tree-Ok. This meets the requirement for a clean specification.