Thinking Programs: Exercises

Chapter 6: Abstract Data Types

1. A *queue* is a "First In/First Out" data structure to which elements can be added at the tail and from which elements can be removed at the head. Develop a formal specification of a data type `Queue` that extends the following skeleton:

```
spec QUEUE[sort Elem] import NAT :=
free type Queue = empty | enqueue(Queue,Elem)
then {
  fun head: Queue → Elem
  fun dequeue: Queue → Queue
  fun size: Queue → Nat
  pred has ⊆ Queue × Elem
  ...
}
```

   The operations shall have the following interpretations:

   - `empty` denotes the queue without any elements;
   - `enqueue` adds an element to the tail of the queue;
   - `head` denotes the element at the head of the queue;
   - `dequeue` removes an element from the head of the queue;
   - `size` denotes the number of elements in the queue;
   - `has` tells whether the queue holds a specific element.

   Here `Queue` is freely generated by constructors `empty` and `enqueue`, i.e., every queue can be uniquely represented by a term of form

```
enqueue(enqueue(...enqueue(empty,e_1)...,e_(n-1)),e_n)
```

   You may assume that `NAT` provides the usual operations on natural numbers (which may be written in the usual infix style).

   Is the abstract datatype denoted by your specification monomorphic or not? Justify your answer.

2. A *file system* is hierarchical collection of *files* and in *directories* which both are denoted by structured names called *paths*. Develop a formal specification of these concepts based on the following skeleton:

```
spec FILESYSTEM import STRING :=
  free type Path = root | path(Path, String)
then free
{
  type FileSystem = empty | mkdir(FileSystem, Path)
                          | write(FileSystem, Path, String)
  ...
```

```
}
then ...
{
  ...
}
```

The specification shall provide the operations:

```
fun parent: Path → Path
fun base:   Path → String

exists ⊆ FileSystem × Path
isdir  ⊆ FileSystem × Path
read:    FileSystem × Path → String
remove:  FileSystem × Path → FileSystem
```

The operations shall have the following interpretations:

- A (file/directory) *path* is a structured name that is either empty (denoting the *root*) or a composition of another path (its *parent*) with a *base* name (a string identifying the file/directory relative to the parent). Among other operations, it is possible to extract the parent and the base name of a path.
- A *file system* is either *empty* or the result of
  - creating by an operation *mkdir* a directory at a denoted path,
  - creating/updating by an operation *write* a file at a certain path with a certain content.

  In both cases, the parent of the path (of the directory/file) must already exist. In a parent directory, there cannot exist two entities (directories and/or files) with the same base name.

  Among other operations, *exists* determines whether the file system has an entity with a given path, *isdir* determines whether this entity is a directory (otherwise it is a file), *read* determines the content of a file and *remove* removes an entity (directory/file) from the file system (if an entity with that path does not exist, the file system remains unchanged).

You may assume that `STRING` provides an (otherwise unspecified) type `String` for base names and file contents.

Please consider carefully in which part of the specification you declare operations and where you add axioms. Axioms in the `free` part of the specification must be formulated in the language of conditional equational logic to ensure the existence of a free (initial) interpretation. However, do not try to further constrain the entities specified in the free part in subsequent parts of the specification by additional axioms; this will make the specification inconsistent.

Are according to your specification two file systems identical, if they provide the same directories and files and file contents? Justify your answer.

3. The set of *integers* consists of *zero* (0), the *positive natural numbers* (1, 2, 3, . . .), also called *whole numbers* or *counting numbers* and their additive inverses (the *negative integers*, i.e., -1, -2, -3, . . .). Develop a formal specification of the domain of integers with the usual operations (constants, functions, predicates) by etending the following skeleton:

```
spec INTEGER import NAT and BOOL :=
free {
  type Int = ...
  ...
}
then ...
{
  const 0I: Int
  const 1I: Int
  fun +: Int × Int → Int
  fun -: Int × Int → Int
  fun -: Int → Int
  fun *: Int × Int → Int
  pred ≤ ⊆ Int × Int
  pred < ⊆ Int × Int
  ...
}
```

Actually, develop *two* specifications, each with a different "representation" of integers:

- In the first version, use the representation of an integer as a pair of a natural number and a sign (represented by a boolean value)

```
free {
  type Int = i(Bool,Nat)
  ...
}
```

but make sure that 0 is uniquely represented ($+0 = -0$).

- In the second version, use the representation of an integer as a difference of two natural numbers

```
free {
  type Int = i(Nat,Nat)
  ...
}
```

but make sure that every integer is uniquely represented ($5 - 3 = 4 - 2 = \ldots = 2 - 0$).

On top of such a free core specification of the integer domain, specify all integer operations either loosely or freely (as you prefer, but do not forget the keyword "free" in the later case). Is your specification monomorphic? Justify your answer.

4. 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.

First, develop a formal specification of the domain of finite binary trees with labels of sort *Value* by extending the following specification skeleton:

```
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. Furthermore, the specification shall contain the following entities:

- 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.

Second, develop a formal specification of the domain of infinite binary trees based on the following specification skeleton:

```
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).

5. In set theory, a *directed graph* is a pair $G = (N, E)$ where N is a set of elements called *nodes* and E is a set of pairs of nodes called *edges*. The goal of this exercise is to specify an abstract datatype of directed graphs with associated operations where $N = \mathbb{N}_n = \{0, ..., n-1\}$ for some $n \in \mathbb{N}$ by extending the following sepecification skeleton:

```
spec GRAPH import NAT :=
... {
  type Graph = g(Nat,Edges)
  type Edges = ...
  ...
}
then ... {
  empty: Nat → Graph
  link: Graph × Node × Node → Graph

  nodes: Graph → Nat
  edges: Graph → Nat

  adjacent ⊆ Graph × Node × Node
  connected ⊆ Graph × Node × Node
  distance: Graph × Node × Node → Nat

  complete ⊆ Graph
  connected ⊆ Graph
  cyclic ⊆ Graph

  ...
}
```

Make sure that two graphs are equal if they have the same nodes and the same edges. The specification shall contain the following entities:

- The function $graph(n)$ denotes the graph with n nodes and no edges while $link(g, n_1, n_2)$

denotes the graph that adds to graph g an edge from n_1 to n_2 (if n_1 or n_2 is not a node of g or g already contains that edge, the graph remains unchanged). Furthermore, $nodes(g)$ denotes the number of nodes in g while $edges(g)$ denotes the number of its edges.

- The predicate $adjacent(g, n_1, n_2)$ holds if g has an edge from n_1 to n_2 (thus it does trivially not hold, if n_1 or n_2 is not a node of g). The predicate $connected(g, n_1, n_2)$ holds if there exists for some $k \geq 1$ a path of nodes $n_1 = m_1, m_2, \ldots, m_k = n_2$ where every pair m_i and m_{i+1} with $i < k$ is adjacent (thus every node n is trivially connected to itself by the path $n = m_1 = n$). For two connected nodes n_1 and n_2 in g, $distance(g, n_1, n_2)$ denotes the number of edges in the shortest path from n_1 to n_2 (thus every node has distance 0 to itself).
- The predicate $complete(g)$ holds if every pair of nodes in g is adjacent. The predicate $connected(g)$ holds if every pair of nodes in g is connected. The predicate $cyclic(g)$ holds, if there is some node in g that is connected to itself by a path with at least one edge.

Does your specification give rise to a monomorphic datatype (justify your answer)?