CS 599 D1: Assignment 6

Due: Wednesday, May 1, 2024

Total: 100 pts

Ankush Das

- This is your last assignment of the course! Congratulations on making it this far!!
- This assignment is due on the above date and it must be submitted electronically on Gradescope. Please create an account on Gradescope, if you haven't already done so.
- This assignment only contains coding questions.
- You should hand in one zip file containing the solutions to the coding assignment in a file named asgn6.rast.
- For all of the problems, feel free to define additional helper processes and types.
- To typecheck and execute the file, use the command `./rast -v asgn6.rast`. This gives more information about the type checking and executions.

Programming with Session Types

In this assignment, you will be using the Rast (short for Resource-Aware Session Types) programming language to implement session-typed processes. Refer to Assignment 4 regarding set-up instructions if you haven't set up Rast already.

For this assignment, since we are using refinement types, we need the following text at the start of the file:

```
#test approx success
#options --syntax=explicit
```

Insert the text above at the start of the asgn6.rast file and start solving the following problems.

1 Lists Lists Lists [40 pts]

This section contains a series of problems involving lists. Let's start by recalling the refinement type for natural numbers.

```
type nat{m} = +{succ : ?{m > 0}. nat{m-1},
                zero : ?{m = 0}. 1}
```

1.1 Bounded Lists

Also, recall the simple type for list of natural numbers.

```
  type listN = +{cons : nat * listN,
                 nil : 1}
```

Problem 1 (5 pts) *Define a type `list{n}{k}` for list of natural numbers with the following property: the size of the list is n and all elements of the list are less than or equal to k.*

Problem 2 (5 pts) *As a first exercise, define a process called `list_double` with the following signature:*

```
decl list_double{n}{k} : (L1 : list{n}{k}) |- (L2 : list{n}{2*k})
```

This process doubles each element in a list, proving that if the elements of input list $L1$ are bounded by value k, then after doubling, the elements of output list $L2$ are bounded by $2k$.

Problem 3 (5 pts) *Next define a process called* `list_sum` *that sums up the elements of a list with the following signature:*

```
decl list_sum{n}{k} : (L : list{n}{k}) |- (S : ?s. ?{s <= k*n}. nat{s})
```

This process proves that if a list is of size n and each element is bounded by k, then the total sum of the list is bounded by $k \times n$.

1.2 Sorted Lists

Problem 4 (5 pts) *Define a type* `sorted_list{k}` *for list of natural numbers with the following property: the elements of the list are sorted in non-decreasing order (i.e., each element is greater than or equal to the previous element) and the first (aka minimum) element in the list has value k.*

Problem 5 (5 pts) *Next define a type* `consecutive_list{k}` *for list of natural numbers with the following property: the elements of the list are consecutive and the first element in the list has value k (e.g., [k; k+1; k+2; ...]).*

Problem 6 (5 pts) *Prove that a list of consecutive natural numbers is sorted by defining the following* `is_sorted` *process:*

```
decl is_sorted{m} : (X : consecutive_list{m}) |- (Y : sorted_list{m})
```

Problem 7 (5 pts) *Finally, test your implementation (even though it's proven correct by the type checker). Define an example list process* `ex_list` *that produces the following list: [1; 2; 3]*

```
decl ex_list : . |- (L : consecutive_list{1})
```

Now, show that this list is sorted by defining the following process:

```
decl ex_sorted_list : . |- (L : sorted_list{1})
```

This process first creates the example list by calling `ex_list` *and then calls the* `is_sorted` *process on the example list. Also, execute this last process by writing*

```
exec ex_sorted_list
```

and write the output as a comment in your code file.

2 Binary Trees [60 pts]

In this section, we will implement several variants of binary trees. First, the basic type of trees without refinements is defined as (for now, the tree does not contain an element at each node)

```
type tree = +{leaf : 1,
              node : tree * tree}
```

Problem 8 (5 pts) *Define a type called* `tree1{n}` *where n tracks the number of nodes in the tree. We only count internal nodes; leaves are not counted. Thus, the type is something like:*

```
type tree1{n} = +{leaf : ?{n = 0}. 1,
                  node : ...}
```

Problem 9 (5 pts) *Prove that your type definition is correct by defining the following* `size` *process that calculates the number of internal nodes in a tree.*

```
decl size{n} : (T : tree1{n}) |- (N : nat{n})
```

Now, we will introduce trees containing natural numbers at each of the nodes. Hence, the type definition (informally) is

```
type tree = +{leaf : 1,
              node : nat{m} * tree * tree}
```

Problem 10 (5 pts) *Define a type called `tree2{n}` where n tracks the sum of all the elements in the tree. Again, leaves do not store any values and are not counted towards the sum.*

Problem 11 (5 pts) *Again, prove that your type definition is correct by defining the following `sum` process that calculates the sum of all the elements of the tree.*

```
decl sum{n} : (T : tree2{n}) |- (N : nat{n})
```

Problem 12 (5 pts) *Just like binary trees, define a list type for natural numbers called `slist{n}` where n tracks the sum of all elements of the list.*

Next, we will be defining inorder and preorder traversals on the trees and proving some arithmetic properties about them.

Problem 13 (10 pts) *Define a process called `in_order` that performs an inorder traversal of the tree and produces a list of natural numbers. The process should have the following signature*

```
decl in_order{s} : (T : tree2{s}) |- (L : slist{s})
```

Problem 14 (10 pts) *Similarly, define a process called `pre_order` that performs a preorder traversal of the tree and produces a list of natural numbers. The process should have the same signature*

```
decl pre_order{s} : (T : tree2{s}) |- (L : slist{s})
```

The two traversal processes above together show that the sum of elements in a tree is equal to the sum of elements in its inorder and preorder traversals.

Problem 15 (5 pts) *Let's test the implementation by defining an example tree with the following structure:*

Define a process `ex_tree` with the following signature:

```
decl ex_tree : . |- (T : tree2{6})
```

Now, define two processes below:

```
decl in_order_ex_tree : . |- (L : slist{6})
decl pre_order_ex_tree : . |- (L : slist{6})
```

These processes both create the example tree by calling `ex_tree` and then calling `in_order` and `pre_order` respectively on the tree.

Execute both these processes using:

```
exec in_order_ex_tree
exec pre_order_ex_tree
```

and write the output as a comment in your code file.

Problem 16 (5 pts) *Define a type called `tree3{h}` where h tracks the height of the tree. Since we do not have a max operator in the refinements, we need to distinguish between nodes where the left subtree is taller than the right subtree and vice-versa. Concretely, the type can be defined as*

```
type tree3{h} = +{nodeL: ... // left tree height >= right tree height
                  nodeR: ... // right tree height > left tree height
                  leaf : ... // leaf nodes have 0 height}
```

Problem 17 (10 pts) *Lastly, prove that your type definition is correct by defining a* `height` *process that calculates the height of a tree.*

```
decl height{h} : (T : tree3{h}) |- (N : nat{h})
```

For this, it might help to define (possibly two) process(es) that return the maximum of two natural numbers.