Case Studies Of Data Structures In Leon: Implementation Of Catenable Lists And Binomial Heaps

Synthesis, Analysis And Verification 2015
Final Report

Mathieu Demarne Maëlle Colussi
EPFL
{firstname.lastname}@epfl.ch

1. Introduction

In this report we present two studies of data structures implemented in Leon. These implementations were done for the course *Synthesis, Analysis and Verification* 2015 at EPFL. We coded and verified some properties of Catenable Lists and Binomial Heaps implemented as described in *"Purely Functional Data Structures"* from Chris Okasaki [Okasaki 1998].

The code for our project as well as the slides of our presentation can be found on GitHub [1].

2. Catenable Lists

The implementation of Catenable Lists we used can be found on pages 93-98 of the reference book ([Okasaki 1998]), and the implementation for Queues, which are used internally by the lists, is described on pages 15-18.

2.1 Data Structure Implementation Overview

The aim of the Catenable List data structure is to have a List implementation with efficient concatenation, hence its name. As said above, Catenable Lists are based on Queues and for this reason we had to implement them as well.

2.1.1 Queues Overview

The implementation represents a Queue as two lists (left and right), and the order of elements in the Queue is the order of the left list, followed by the reverse order of the right list. The operations implementations are easy to deduce from that: we add elements on the right list (with `snoc`) and remove them from the left list (with `head`). At some point, the right list must obviously be reversed and put on the left. In our case, in order to have the `head` operation in constant time, as is wanted in the implementation, the invariant must be enforced that a non-empty list has a non-empty left list. It comes from that, that the `tail` operation must reverse the right list when the left one becomes empty, and `snoc` must add an element on the left list of an empty Queue.

All operations are in constant time, or amortized constant time for `tail`. Let us now see an overview of the Catenable List structure.

2.1.2 Catenable Lists Overview

We can represent a Catenable List as a tree in which the order of elements can be recovered by traversing the tree in preorder. It is implemented as an element (the root of the tree), and a Queue of Catenable Lists (its children).

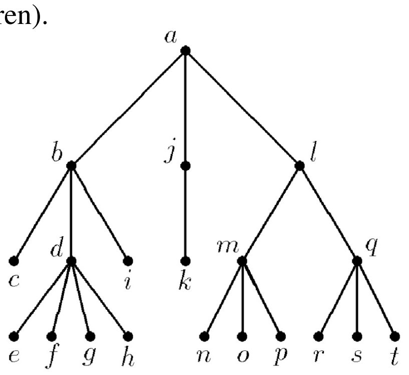

Illustration 1: representation of a list from a to t

The concatenation is then simply done by adding the List we want to concatenate at the end of the Queue of the current List (operation called `link`). When it comes to the other operations, `head` is trivial (it simply returns the root) and operations adding an element at the end

[1] See github.com/mdemarne/leon-functional-datastructures

or beginning of the List, `snoc` and `cons`, simply use the `concatenation` operation. The `tail` operation requires to create a Catenable List from a Queue of Catenable Lists. For that, it just links all Lists of the Queue by putting the successor of a List l in the Queue at the end of l's Queue.

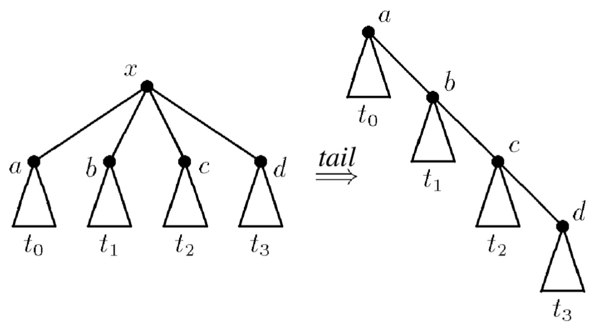

Illustration 2: overview of the `tail` operation

All operations are done in amortized constant time. Let us now see its verification with Leon.

2.2 Verification with Leon

In order to verify the correctness of the implementation of both data structures in Leon, we used some functions like `size`, `content` and `toList`. The Queue implementation had also an invariant to be checked (explained above), so the Catenable List implementation had to ensure that the Queues it uses satisfy the invariant as well.

We also wrote some tests to be evaluated that helped us to find some errors, where we could not be precise enough in the specification of functions, or where Leon gave `Unknowns`.

The Queue implementation could be verified by Leon, but the Catenable List implementation had some `Unknowns`, probably due to its recursive structure. Overall, we were able to verify 67 checks over 78.

In the process of verification we learned some limitations of Leon, such as that the use of high-order functions makes verification hard if not impossible. We had also some difficulties with mutually recursive checks, which generated stack-overflows at run-time.

In the process, we also managed to compile Leon code directly using `scalac`, which allowed us to test our implementation both using the `--eval` option of Leon and as a standalone program [2].

[2] See stackoverflow.com/questions/29990106/how-do-i-build-with-scalac-using-leons-library

3. Binomial Heaps

The implementation of Binomial Heaps we used, along with Trees, can be found in pages 68-72 of the reference book ([Okasaki 1998]).

3.1 Data Structure Overview

A Binomial Heap is an implementation of mergeable (or meldable) priority queue. It uses Trees that we had to implement as well. Let us first see their implementation.

3.1.1 Trees Overview

A Tree is simply represented as a Node with an element (the priority, of totally ordered type), a rank and a list of Trees as children. The rank, which is ≥ 0, is to be understood as follows: a Node of rank k has k children of ranks k, $k-1$, ..., 0, a rank 0 meaning a Node without children. From that it comes that a Tree with a root Node of rank k contains in total 2^k elements, so a Tree can represent a power of two, like the *k-th* bit in a binary number.

The Trees must satisfy the *Minimum Heap Property*, which means that the element of a parent Node is less or equal than any of its children Node's elements. Also, in the implementation, the children list is maintained in decreasing order of rank. These properties are thus two invariants to check on Trees, along with the domain of the rank.

The `link` operation on Trees takes two Trees of rank k, makes the one with bigger root element the first child of the other so that both invariants are satisfied. As a result, the returned tree is of rank $k+1$. It is thus like the addition of two bits of a binary number. The operation is done in constant time.

Let us now see how the Trees are used in the Binomial Heap implementation.

3.1.2 Binomial Heaps Overview

A Binomial Heap is implemented as a list of Trees which is kept in increasing order of rank. There must also not be more than one Tree of a particular rank. These properties are thus invariants to check.

We can then visualize a well-formed Binomial Heap as the binary number of its size, with the Trees in the list as the 1-valued bits of the number.

With the help of the `link` operation on Trees, we can see a `merge` of two Binomial Heaps as an addition of the two binary numbers that represent them. In this view, `link` is used to add two bits of same position, and the result of the operation is the carry.

The other operations on Binomial Heaps are `insert`, `findMin` and `deleteMin`. The `insert` operation creates a 0-ranked Tree with the element to insert and puts the Tree in the Heap, using an `insertTree` function, also used by `merge` to insert the carry in the recursively merged Heaps, at each carry step. This function inserts the Tree only if its rank is smaller or equal to the smallest rank in the Tree list of the Heap (either it adds it to the list, either it recursively `links` and `insertTrees`), so it had to be checked that the function is not called with a Tree of bigger rank in the implementation.

The `findMin` operation needs only to look for the minimum element in the root Nodes of the Trees in the Tree list. This is because the Trees satisfy the *Minimum Heap Property*.

Finally, the `deleteMin` operation finds the Tree with the minimum root in the list, removes its root and reverses its children so that the children list is of the correct form for a Tree list of a Binomial Heap. After that, it `merges` this new Heap with the Heap composed of the remaining Trees of the original Binomial Heap.

All operation are done in amortized logarithmic time. Let us see now the verification of the implementations for both Trees and Binomial Heaps with Leon.

3.2 Verification with Leon

As for the first data structure verification presented in this report, `size`, `content` and `toList` functions were used for pre- and post-conditions of operations, as well as the invariants for Trees and Binomial Heaps we saw above.

Unfortunately we managed to prove only 46 properties over 53. We have tried to tweak our model in order to help Leon proving some of them, but unfortunately the recursive structure of Trees prevented us to reduce the number of `Unknowns`.

Some tests were written for the Binomial Heaps and it helped to find some mistakes we made in writing the code and which were not found by Leon. For example, an invariant was too strong regarding the ordering of the Trees in the Heap and the error could be seen only with tests, when a precondition for an operation was not satisfied.

We had some difficulties to state the total order of the elements' type (`T <: Ordered[T]` was not understood by Leon), so we decided to use `BigInts` to be able to verify the data structure, as it can be done without loss of generality regarding the logic of the implementation. We also had to adapt a bit our implementation in order to circumvent some of Leon's limitations. For instance, a Heap is represented as a `List[Tree]`. It could therefore have been interested to use an implicit class to operate on such structure, a construct that is not yet available in Leon.

Moreover, our data structures used standard constructs such as lists, but with some specific properties. In order to enforce those properties, we had to verify the invariants as `requires` at the beginning of each `def`. It would have been nice instead to be able to write requirements directly inside `case classes`, as it would have remove some boilerplate code.

Leon could unfortunately not state as valid most of complicated, crucial operations, which are still `Unknowns`.

4. Conclusion

We were able to implement two data structures in our project, namely Catenable Lists and Binomial Heaps, and to check some operations on them. Unfortunately Leon failed to prove or disprove some properties. However our experiments have shown that this is difficult to avoid with recursive definitions. The results we described in this report were obtained with the `--assumepre` option, which allowed to prove a few more conditions.

We were able to isolate some limitations and find ways to circumvent them, being either library ones (like operations on `Sets` which were not supported) for compilation with `scalac` or syntactic ones.

It was nevertheless very interesting to use Leon and it helped us find some errors in translating the book's implementation into Scala ([Okasaki 1998]).

Our implementation of the two data structures we have studied is of course purely functional.

References

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.