A Leon Case Study: Trie Trees

Ioannis Lamprou

June 7, 2015

1 Introduction

A *trie tree* is a data structure used for storing pairs of strings and values. The name stems from the word re*trie*val. Each node of the tree contains an optional value. The path from the root to a specific node corresponds to a string from the universe of a given alphabet. The root corresponds to the empty string. One can imagine that each edge is asssociated with a specific character from the alphabet. Trie trees are *prefix trees* in the sense that each node corresponds to a prefix and nodes with a common parent share a common prefix. A key application of trie trees is the maintenance of dictonaries. An example of such a tree is given in the following figure, where black nodes correspond to nodes containing values (i.e. words in the dictionary).

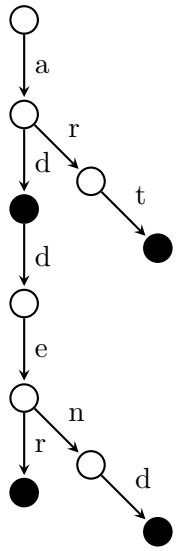

Figure 1: A trie tree containing the words `ad`, `art`, `adder` and `addend`

We choose the following reprentation for a trie tree in Scala. A `Trie` is

- either `Empty`
- or a `Node(v: Option[BigInt], children: List[(Char, Trie)])`

In the latter case, the character-subtree association is dictated by the children list of pairs. Another possible representation (used mostly in procedural programming languages) is to have a fixed alphabet and thus maintain in each node a fixed-size array of subtrees, where there is a bijection from the alphabet to the array indices.

2 Operations

A trie tree supports three main operations, namely

- `findValue`: given a string as a key, return the (optional) value which corresponds to it
- `insert`: given a string and a value, return a new tree that contains the value in a node corresponding to the string
- `delete`: given a string, delete it (and the corresponding value) from the tree

The `findValue` function follows the string character-by-character and traverses the tree accordingly. If, at any step, an empty tree is reached, then it outputs that the string is not contained in the tree. Otherwise, once it has processed the whole string and reached the node in question, then it outputs the optional value contained there.

The `insert` function works in similar fashion. It follows the path indicated by the input string. If the whole string is processed and a non-empty node is reached, then insertion reduces to updating the value of the already present node corresponding to the string. Otherwise, if at some point an empty node is reached, then the function constructs a path starting from that node and corresponding to the suffix of the string not yet processed. The given value is stored at the end-node of the path. The parents' children lists are updated respectively during this process.

The `delete` function is the most complicated. There are a few cases to consider:

- *the string is not in the tree*: the tree need not be modified
- *the string corresponds to an internal node*: the optional value found in the node is updated to `None()`; no further modifications needed.
- *the string is a leaf*: the cases to be considered are the following:
  1. *the string is a suffix of another word*
  2. *the string shares a prefix with another word*
  3. *none of the above two holds*

  The last case is the easiest one, since the word to be deleted holds no correlation with other words. One can remove the whole path from the root to it. On the other hand, the first two cases can both happen simultaneously and concerning many different words. The algorithmic idea is the following: while traversing the tree, keep track of the last node corresponding to either a word (case one) or a branching point (case two) having more than one subtrees as children. Then, after reaching the leaf, do a second traversal from the root till this special node and update its children list by removing the useless subpath.

3 Experiments

Experiments were run on a laptop running Ubuntu 12 on an Intel Core2Duo 2.40GHz processor using an external smt-z3 solver. Many functions gathering different sorts of content are implemented. Leon experiences severe difficulties to verify properties reasoning both for values and strings. We manage to provide some partial verification only for the `findValue` function. Verifying `insert` and `delete` postconditions results in timeouts due to the complexity of their implementation. Nonetheless, all preconditions are met when called recursively.

4 Bit-Trie Trees

A special case of the data structure is tested as well. We consider *bit-trie trees*. That is, trie trees with a fixed alphabet of size two. We now consider bit-strings instead of general strings. All the functionality is similarly implemented for this special case. The motivation is that instead of maintaining a list of children for each node, we now maintain just two explicit children. Thence, it may be easier for Leon to verify certain properties. Indeed, Leon manages to verify `findValue` very fast and `insert` after approximately three seconds; again strings are not considered. For `insert`, we can ensure that the string is contained in the tree via a check on the `findValue` function. Requiring positive values in the `findValue` precondition leads to a timeout in the `insert` postcondition. We also satisfy the *all leafs are words* and size increase properties. Nevertheless, `delete` still timeouts and seems difficult to tackle, given its complicated nature.