American University of Beirut

Project Report: Formalization of Trie Structure Functions

Ameen Jaber

June 10, 2012

1 Introduction

A Trie structure, abbreviated for "Retrieval", is kind of a digital search tree and an efficient indexing method. Starting from head to tail, each node is a state and the input character determines the next state to go to. There are different implementations proposed and used for the Trie structure, starting with the two-dimensional array which is least efficient in terms of memory use.

In the course project, I formalized the implementation of the double-array Trie structure proposed in (Aoe, 1989) . The implementation of this structure is present at the link "An Implementation of Double-Array Trie" I referred to this implementation in my work, and wrote English specifications for a number of the functions present in the code then represented them with a set of Boolean formulas. After that and based on the method introduced in (Attie, Zaraket, Fawaz, & Noureddine), I expressed the previous specifications accurately and formally.

2 Accomplished Work

In this section, i present each of the chosen functions to formalize through writing formal specifications to it. For each one, i will explain the purpose of the function, then present the required vocabulary followed by the pre and post conditions.

2.1 da_get_base

```
TrieIndex
da_get_base (const DArray *d, TrieIndex s)
{
    return (s < d->num_cells) ? d->cells[s].base : TRIE_INDEX_ERROR;
}
```

Explanation For clarification, base and check represent the double array in the trie structure. Those slots are used to make the transitions from one state to another getting the values from the base while the check is used for transition validation. The following function returns the base cell value for a given input state if the index was within the cell number bound, else returns an index error.

Vocabulary The vocabulary used for the precondition and postcondition specifications are present below:
VOC_i= { d $\neq$ 0, s $\geq$ 0 }
VOC_o= { s < d.num_cells, rv=d$\rightarrow$cells[s].base, rv=TRIE_INDEX_ERROR }

Pre/Post Conditions The precondition specification of the function is as follows:
P = d $\neq$ 0 $\wedge$ s $\geq$ 0
And the postcondition specification of it is as follows:
Q=(s< d.num_cells $\wedge$ rv=d$\rightarrow$cells[s].base) $\vee$ (!(s< d.num_cells) $\wedge$ rv= TRIE_INDEX_ERROR)

2.2 da_get_check

```
TrieIndex
da_get_check (const DArray *d, TrieIndex s)
{
    return (s < d->num_cells) ? d->cells[s].check : TRIE_INDEX_ERROR;
}
```

Explanation The following function returns the check cell value for a given input state if the index was within the cell number bound, else returns an index error. The specifications of this function use almost the same vocabulary of the previous one because the same functionality is being implemented for a different field of the double array.

Vocabulary The vocabulary used for the precondition and postcondition specifications are present below:
VOC_i= { d $\neq$ 0, s $\geq$ 0 }
VOC_o= { s < d.num_cells, rv=d$\rightarrow$cells[s].check, rv=TRIE_INDEX_ERROR }

Pre/Post Conditions The precondition specification of the function is as follows:
P = d $\neq$ 0 $\wedge$ s $\geq$ 0
And the postcondition specification of it is as follows:
Q=(s< d.num_cells $\wedge$ rv=d$\rightarrow$cells[s].check) $\vee$ (!(s< d.num_cells) $\wedge$ rv= TRIE_INDEX_ERROR)

2.3 da_set_base

```
TrieIndex
void
da_set_base (DArray *d, TrieIndex s, TrieIndex val)
{
    if (s < d->num_cells) {
        d->cells[s].base = val;
    }
}
```

Explanation The following function sets the **base** slot at index **s** to the value **val** if the index is within the base bound, else nothing happens.

Vocabulary The vocabulary used for the precondition and postcondition specifications are present below:
VOC_i= { d $\neq$ 0, s $\geq$ 0 }
VOC_o= { s < d.num_cells, d$\rightarrow$cells[s].base=val}

Pre/Post Conditions The precondition specification of the function is as follows:
P = d $\neq$ 0 $\wedge$ s $\geq$ 0
And the postcondition specification of it is as follows:
Q=(s< d.num_cells $\wedge$ d$\rightarrow$cells[s].base=val) $\vee$!(s< d.num_cells)

2.4 da_set_check

```
void
da_set_check (DArray *d, TrieIndex s, TrieIndex val)
{
    if (s < d->num_cells) {
        d->cells[s].check = val;
    }
}
```

Explanation The following function sets the **check** slot at index **s** to the value **val** if the index is within the base bound, else nothing happens. Again, this function implements the exact same function expalined in the previous part for the check array, hence the specifications are almost the same.

Vocabulary The vocabulary used for the precondition and postcondition specifications are present below:
VOC_i= { d $\neq$ 0, s $\geq$ 0 }
VOC_o= { s < d.num_cells, d→cells[s].check=val}

Pre/Post Conditions The precondition specification of the function is as follows:
P = d $\neq$ 0 $\wedge$ s $\geq$ 0
And the postcondition specification of it is as follows:
Q=(s< d.num_cells $\wedge$ d→cells[s].check=val) $\vee$!(s< d.num_cells)

2.5 da_walk

```
Bool
da_walk (const DArray *d, TrieIndex *s, TrieChar c)
{
    TrieIndex   next;

    next = da_get_base (d, *s) + c;
    if (da_get_check (d, next) == *s) {
        *s = next;
        return _TRUE;
    }
    return _FALSE;
}
```

Explanation The following function attempts to walk the double array **d** from state **s** using the input character **c**. If such a transition is present, a true value is returned and **s** is set to transition state, else a false value is returned. The specifications of the following function are dependent on on specifications from previous functions formalized, thus we can introduce their specifications as part of the following function's specifications.

Vocabulary The vocabulary used for the precondition and postcondition specifications are present below:
VOC_i= { d $\neq$ 0, s $\geq$ 0, c $\geq$ 0 }
In the post condition specifications, we need to check for the following:

```
t := base[s] + c;

 if check[t] = s then
     next state := t
 else
     fail
 endif
```

Thus, we need to retieve **base[s]** and **check[base[s]+c]** VOC_o= { s<d.num_cells, rv=d→cells[s].base, rv=TRIE_INDEX_ERROR, s=d→cells[s'].check, rv=(rv'+c)}

Pre/Post Conditions The precondition specification of the function is as follows:
$P = d \neq 0 \land s \geq 0 \land c \geq 0$

And the postcondition specification of this function can be derived from the previous base and check get functions, thus it can be formulated as follows:
Q= [(s < d.num_cells ∧ rv'=d→cells[s].base) ∨ (!(s<d.num_cells) ∧ rv'=TRIE_INDEX_ERROR)] ∧ [((rv'+c) < d.num_cells ∧ d→cells[rv'+c].check=s ∧ rv=rv'+c) ∨ (!((rv'+c)<d.num_cells) ∧ rv=TRIE_INDEX_ERROR)]

The first part of the postcondition represents the requirements of the solution from the base read part, whereas the next part represents the requirement of the check array. The final result of the next state is returned for valid result.

3 Conclusion

In the following project, I formalized a couple of the functions implemented in the Trie structure based on the specification construction method discussed in the paper.