Assignment 3 - Trie Harder

z5163480 Nabil Shaikh   z5162498 Bailey Ivancic

May 30, 2018

1 Task 1 : Intro

Define a dictionary as a finite set of words, where, for simplicity, a word is a finite sequence of letters over $L = \{\text{'a'}, \ldots, \text{'z'}\}$. Consider a syntactic stack data type **DICT** and its representation variable is a dictionary set W. The data type comprises of an initialization predicate $init^W \triangleq W = \{\}$.The dictionary has the following procedures:

- adding a given word to the set. (addWord(w))
- checking whether a given word is contained in the set. (checkWord(w))
- deleting a given word. (delWord(w))

2 Task2 : Abstract Dictionary

Consider a syntactic set data type **DICT**. Its representation variable is a dictionary set W. The data type comprises of an initialization predicate $init^W \triangleq W = \{\}$. The dictionary has the following procedures:

```
func checkWord(W, value $w$)
    value $b \cdot b : [\textsc{true}, b = (w \in W)]$
    return $b$

proc addWord(W, value $w$)
    $W : [\textsc{true}, W = W_0 \cup w]$

proc delWord(W, value w)
    $W : [\textsc{true}, W = W_0 \setminus \{w\}]$
```

3 Task 3 : Refinement of DICT into DICT^A

Formally, we write $v \leq w$ if word v$\in L^*$ is a prefix of $w \in L^*$, i.e., $\exists v\prime \in L^* \, (vv\prime = w)$. We write B for {0, 1} where 0 represents 'false' and 1 'true'. A trie domain is a prefix- closed finite subset of L^*. A trie is a function from a trie domain to Booleans. Given a trie t we write dom(t) for its trie domain. Let T be the set of all tries.

Example 1 D = {ϵ,'a', 'a''d', 'a''d'd'} is a trie domain and t = {$\epsilon \mapsto$ 0, 'a'$\mapsto$ 0, 'a''d' $\mapsto$ 0, 'a''d''d' $\mapsto$ 1} is a trie with dom t = D. It represents the set {'a', 'a''d', 'a''d''d'} of words. Note ϵ represents a universal collection of all other combinations of L* and this maps to 0, such that t("notInTrie") = 0.

Define a mapping function from a concrete data type to an abstract data type. Such that any Trie maps to a valid abstract set W.

$$f(t) = \begin{cases} \{\} & \text{if } \forall x \in \text{dom } t \, (t(x) = \textsc{false}) \\ \{x \cup f(t : x \mapsto \textsc{false})\} & \text{if } \exists x \in \text{dom } t \, (t(x) = \textsc{true} \land x \notin W_0) \end{cases}$$

let the constructed data type **seq** w := "$word$" be defined as an ordered sequence of characters, such that w[0] = 'w'. let the addition of two sequences, seq1 + seq2, represent the concatenation of those strings and $|seq|$ represent the length/size of the sequence seq.

let the constructed data type t : **seq**$CHAR$ $\mapsto \mathbb{B}$ be a function that maps from a sequence of characters to a boolean value.

let t[w] represent the boolean value from w .ie w $\mapsto$ TRUE. Further define that t[w:= FALSE] either adds the mapping $\{w \mapsto \textsc{false}\}$ or changes the previous value of t[w] = TRUE to t[w] = FALSE. This defines the building blocks of our trie.

Referencing from programming from specification by Carroll Morgan, **seq** CHAR is a sequence of characters and the declaration of a function are both inspired from the chapter constructed types.

```
module DICT$^A$

    proc init$^A$()
        var $t$ : seq CHAR $\mapsto \mathbb{B}$
        t := $\{\epsilon \mapsto \textsc{false}\}$

    func checkWord$^A$(t, value pre · seq, value w · seq, value i · $\mathbb{N}$)
        value $b \cdot b, i, pre : [\textsc{true}, b = t[pre] \land pre = w \land i = |pre|]$
        return $b$

    proc addWord$^A$(t, value pre · seq, value w ·seq, value i · $\mathbb{N}$)
```

```
        $t, i, pre : [\text{TRUE}, \forall v \leq w\,(\forall v \notin D\,(t[v] = \text{FALSE}) \land \forall v \in D\,(t[v] = t[v]_0))$
                $\land t[w] = \text{TRUE} \land pre = w \land i = |pre|]$

    **proc** delWord$^A$(t, **value** pre $\cdot$ **seq**, **value** w $\cdot$ **seq**, **value** i $\cdot \mathbb{N}$)
        $t, pre, i : [\text{TRUE}, t = t_0 : pre \mapsto \text{FALSE} \land pre = w \land i = |pre|]$

**end**
```

3.1 Refinement of checkWord

Lets start refining the concrete data type by refining checkword.

```
**func** checkWord$^A$(**value** w $\cdot$ **seq**, **value** pre $\cdot$ **seq**, **value** i $\cdot \mathbb{N}$)
    **value** $b \cdot \llcorner b, i, pre : [\text{TRUE}, b = t[pre] \land pre = w \land i = |pre|]\lrcorner_{(0)}$
    **return** $b$

$(0) \sqsubseteq$        ⟨if condition⟩
    **if** w $\notin$ D **then**
        $\llcorner b : [w \notin D, b = t[pre] \land pre = w \land i = |pre|]\lrcorner_{(1)}$
    **else**
        $\llcorner b : [w \in D, b = t[pre] \land pre = w \land i = |pre|]\lrcorner_{(2)}$
    **fi**

$(1) \sqsubseteq$        ⟨ass⟩
    $b := \text{FALSE}$

$(2) \sqsubseteq$        ⟨if-condition⟩
    **if** pre=$w$ **then**
        $\llcorner b : [w \in D \land pre = w, b = t[pre] \land pre = w \land i = |pre|]\lrcorner_{(3)}$
    **else**
        $\llcorner b : [w \in D \land pre \neq w, b = t[pre] \land pre = w \land i = |pre|]\lrcorner_{(4)}$
    **fi**

$(4) \sqsubseteq$        ⟨seq⟩
    $\llcorner b : [w \in D \land pre \neq w, i + 1 = |pre| \land pre \leq w]\lrcorner_{(5)}$
    $\llcorner b : [i + 1 = |pre| \land$ pre$\leq$ w, $i = |pre|]\lrcorner_{(6)}$
    $\llcorner b :$ [i=$|pre|$, $b = t[pre] \land pre = w \land i = |pre|]\lrcorner_{(7)}$
```

$(5) \sqsubseteq$ $\langle$ass$\rangle$
$pre := pre + w[i];$
$(6) \sqsubseteq$ $\langle$ass$\rangle$
$i := i + 1;$
$(7) \sqsubseteq$ $\langle$function call and weaken pre$\rangle$
checkWord(w);

$(3) \sqsubseteq$ $\langle$if condition$\rangle$

```
if t[pre] = TRUE then
    $\llcorner b : [t[pre] = \text{TRUE} \wedge pre = w \wedge \text{w} \in \text{D},\ b = t[pre] \wedge pre = w \wedge i = |pre| \lrcorner_{(8)}$
else
    $\llcorner b : [t[pre] \neq \text{TRUE} \wedge pre = w \wedge \text{w} \in \text{D},\ b = t[pre] \wedge pre = w \wedge i = |pre|] \lrcorner_{(9)}$
fi
```

$(8) \sqsubseteq$ $\langle$since pre=w and t[pre] = TRUE$\rangle$
$b :=$ TRUE
$(9) \sqsubseteq$ $\langle$since pre=w and t[pre] = FALSE$\rangle$
$b :=$ FALSE

Now to complete the refinement.

```
func checkWord$^A(w, pre, i)$
    if w$\notin$ D then
        $b :=$ FALSE;
    else
        if pre = w then
            if t[pre] = TRUE then
                $b :=$ TRUE;
            else
                $b :=$ FALSE;
            fi
        else
            pre := pre + $w[i]$;
            $i := i + 1$;
            $b :=$ checkWord($w$, pre, $i$);
        fi
    fi
return b;
```

3.2 Refinement of addWord

Moving on to the refinement of addWord:

```
proc addWord$^A$(value w $\cdot$seq, value pre $\cdot$seq, value i $\cdot\mathbb{N}$)
    value $t \cdot \llcorner t, i, pre : [\text{TRUE}, \forall v \leq w\,(t = t_0 : v \mapsto \text{FALSE}) \wedge t = t_0 : w \mapsto \text{TRUE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(0)}$
```

$(0) \sqsubseteq \quad \langle\text{if-condition}\rangle$

```
if pre = w then
    $\llcorner t : [pre = w, \forall v \leq w\,(t = t_0 : v \mapsto \text{FALSE}) \wedge t = t_0 : w \mapsto \text{TRUE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(1)}$
else
    $\llcorner t : [pre \neq w, \forall v \leq w\,(t = t_0 : v \mapsto \text{FALSE}) \wedge t = t_0 : w \mapsto \text{TRUE} \wedge \wedge pre = w \wedge i = |pre|]\lrcorner_{(2)}$
```

$(1) \sqsubseteq \quad \langle\text{ass and with hand waving, assume } \forall v \leq w.t[v] = \text{FALSE} \text{ since it is a recursive function}\rangle$

$t[w := true]$

$(2) \sqsubseteq \quad \langle\text{seq}\rangle$

$\llcorner t : [pre \neq w, pre \leq w \wedge i + 1 = |pre|]\lrcorner_{(3)}$
$\llcorner t : [pre \leq w \wedge i + 1 = |pre|, pre \leq w \wedge i = |pre|]\lrcorner_{(4)}$
$\llcorner t : [pre \leq w \wedge i = |pre|, \forall v \leq pre\,(t = t_0 : v \mapsto \text{FALSE})]\lrcorner_{(5)}$
$\llcorner t : [\forall v \leq pre\,(t = t_0 : v \mapsto \text{FALSE}), \forall v \leq w\,(t = t_0 : v \mapsto \text{FALSE}) \wedge t = t_0 : w \mapsto \text{TRUE}]\lrcorner_{(6)}$

$(3) \sqsubseteq \quad \langle\text{ass}\rangle$

pre $:= pre + w[i]$

$(4) \sqsubseteq \quad \langle\text{ass}\rangle$

$i := i + 1;$

$(5) \sqsubseteq \quad \langle\text{ass}\rangle$

$t[pre := false]$

$(6) \sqsubseteq \quad \langle\text{ass post } [^{i+1}/_i] \text{ and seq}\rangle$

addWord(w, pre, i)

Now to complete the refinement.

```
func addWord$^A$($w$)
    if pre = w
        $t[pre := true]$
    else
        pre = pre + w[i]
        $i = i + 1$
        $t[pre := false]$
        addWord($w$)
    fi
```

3.3 Refinement of delWord

Moving on to the refinement of delWord:

```
proc delWord(value $w \cdot$ seq, value $pre \cdot$ seq, value $i \cdot \mathbb{N}$)
    $\llcorner t, pre, i : [\text{TRUE}, t = t_0 : pre \mapsto \text{FALSE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(0)}$

(0) $\sqsubseteq$     ⟨if condition⟩
    if w $\notin D$ then
        $\llcorner t : [w \notin D, t = t_0 : pre \mapsto \text{FALSE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(1)}$
    else
        $\llcorner t : [w \in D, t = t_0 : pre \mapsto \text{FALSE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(2)}$
    fi

(1) $\sqsubseteq$     ⟨SKIP since element isn't in domain⟩
    skip;

(2) $\sqsubseteq$     ⟨if condition and weaken pre⟩
    if pre = w then
        $\llcorner t : [\text{pre=w}, t = t_0 : pre \mapsto \text{FALSE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(3)}$
    else
        $\llcorner t : [\text{pre}\neq\text{w}, t = t_0 : pre \mapsto \text{FALSE} \wedge pre = w \wedge i = |pre|]\lrcorner_{(4)}$
    fi

(3) $\sqsubseteq$     ⟨ass⟩
    $t[pre := false]$
(4) $\sqsubseteq$     ⟨seq⟩
    $\llcorner t : [\text{pre}\neq\text{w}, i + 1 = |pre| \wedge pre \leq w]\lrcorner_{(5)}$
    $\llcorner t : [pre \leq w \wedge \text{i+1=}|pre|, i = |pre|]\lrcorner_{(6)}$
    $\llcorner t : [\text{ i=}|pre|, t[pre] = false \wedge pre = w \wedge i = |pre|]\lrcorner_{(7)}$

(5) $\sqsubseteq$     ⟨ass⟩
    $pre := pre + w[i]$
(6) $\sqsubseteq$     ⟨ass⟩
```

```
    $i = i + 1$
(7) $\sqsubseteq$      $\langle$weaken pre and function call$\rangle$
    delWord($w$, pre, $i$)
```

Now to complete the refinement.

```
func delWord$^A$($w, pre, i$)
    if w$\notin$ D then
        skip;
    else
        if pre = w
            $t[pre := false]$;
        else
            pre = pre[i] + w[i];
            $i = i + 1$;
            delWord($w$, pre, $i$);
        fi
    fi
```

3.4 Proof obligations

We begin to list the relevant proof obligations. Note that the coupling invariant r is replaced by an equivalent mapping function f, that creates an abstract data type w from a given concrete data type t.

$$\text{init}^A \Rightarrow \text{init}[^{f(t)}/_w] \qquad (2_f)$$

$$\text{pre}_j[^{f(t)}/_w] \Rightarrow \text{pre}_j^A \text{ , for j}\in\text{J} \qquad (3_f)$$

$$\text{pre}_j[^{f(t_0),x_0}/_{w_0,x}] \wedge \text{post}_j^A \Rightarrow \text{post}_j[^{f(t_0),f(t)}/_{w_0,w}] \text{ , for j}\in\text{J} \qquad (4_f)$$

$$\forall t\left(\exists w\left(f(t) = w\right)\right) \text{, if } \exists j, t\left(\neg\text{pre}_j^A\right) \qquad (5_f)$$

Begin with addressing the 1st proof obligation (2_f).

$$\text{init}^A \Rightarrow \text{init}[^{f(c)}/_a]$$

$$\Leftrightarrow \qquad \langle\text{definition of init}^A \text{ and init}\rangle$$

$$t := \epsilon \mapsto \text{FALSE} \Rightarrow f(t) = \{\}$$

Next, consider 5_f. This falls since f is a total function, therefore satisfying the condition.

Next prove 3_f for all operations of DICT^A. Yet, all the preconditions of the operation are true, this falls immediately.

To begin with the proof obligations for 4_f first add all relevant variables to the frame and add zero sub scripted variables to them.

Begin with the operation checkWord

$$
\begin{aligned}
& \text{pre}_j[{}^{f(t_0),x_0}/_{w_0,x}] \wedge \text{post}_j^A \\
\Leftrightarrow \quad & \langle \text{definition of LHS} \rangle \\
& \textsc{true}[{}^{f(t_0),x_0}/_{w_0,x}] \wedge b = t[pre] \wedge pre = w \wedge i = |pre| \wedge w, t = w_0, t_0 \\
\Rightarrow \quad & \langle \text{since pre=}w \rangle \\
& b = t[w] \\
\Leftrightarrow \quad & \langle \text{comparison between function mapping and set belonging} \rangle \\
& b = (w \in f[t]) \\
\Leftrightarrow \quad & \langle \text{The definition of the RHS} \rangle \\
& \text{post}_j[{}^{f(t_0),f(t)}/_{w_0,w}]
\end{aligned}
$$

Now check the condition with delWord

$$
\begin{aligned}
& \text{pre}_j[{}^{f(t_0),x_0}/_{w_0,x}] \wedge \text{post}_j^A \\
\Leftrightarrow \quad & \langle \text{definition of LHS} \rangle \\
& \textsc{true}[{}^{f(t_0),x_0}/_{w_0,x}] \wedge t = t_0 : pre \mapsto \textsc{false} \wedge pre = w \wedge i = |pre| \\
\Rightarrow \quad & \langle \text{since pre=}w \rangle \\
& t = t_0 : w \mapsto \textsc{false} \\
\Leftrightarrow \quad & \langle \text{comparison between function mapping and set minus} \rangle \\
& f(t) = f(t_0) \setminus \{w\} \\
\Leftrightarrow \quad & \langle \text{The definition of the RHS} \rangle \\
& \text{post}_j[{}^{f(t_0),f(t)}/_{w_0,w}]
\end{aligned}
$$

For addWord we prove

$$
\begin{aligned}
& \text{pre}_j[{}^{f(t_0),x_0}/_{w_0,x}] \wedge \text{post}_j^A \\
\Leftrightarrow \quad & \langle \text{definition of LHS} \rangle \\
& \textsc{true}[{}^{f(t_0),x_0}/_{w_0,x}] \wedge \\
& \forall v \leq w \, (\forall v \notin D \, (t[v] = \textsc{false}) \wedge \forall v \in D \, (t[v] = t[v]_0)) \\
& \wedge t[w] = \textsc{true} \wedge w = w_0 \\
\Rightarrow \quad & \langle \text{since pre=}w \text{ and we aren't changing other function mappings except the ones we add} \rangle \\
& t = t_0 : w \mapsto = -\textsc{true}
\end{aligned}
$$

$\Leftrightarrow$ $\langle$comparison between function mappings and set union$\rangle$

$f(t) = f(t_0) \cup w$

$\Leftrightarrow$ $\langle$The definition of the RHS$\rangle$

$\text{post}_j[^{f(t_0),f(t)}/_{w_0,w}]$

4 Translation to C code

4.1 Explanation of translation

While we have tried to keep the C code as similar to the toy language code as possible, there were a few changes that had to be made in the conversion.

The most obvious change that has been made is the use of struct in the C code instead of a trie function and domain. In the C code, the Dict struct has a field eow, which is a Boolean identifier for the end of a word. This essentially mimics the trie function within our toy language, which would provide a mapping to a boolean (true, false) for each element of the trie domain. Since each Node in the C implementation represents the end of a prefix, the eow boolean takes the place of the trie function, which will also map each word to a given boolean.

Another change is the use of an array of Dict structs in place of the trie domain. In our toy language, each prefix of a word w is contained within the domain as it's own separate entity. This was done explicitly to avoid the use of nodes, which the toy language does not support. In the C code, each Node (Dict) down the tree would represent an element within the trie domain of the toy language. As a result, the recursive function calls in the toy language functions have been retained in the c code, however instead of iterating through the sequence *pre*, we are traversing through the tree Nodes.

Our C functions also differ from the toy language functions in the way they look through strings. In the toy language, we have a variable i that keeps track of our position within the sequence of chars we are looking at. In the C code, instead of using a variable i to track our position within the string, we pass the string into the next recursive call starting at our current position. Since we are able to achieve this without holding onto this extra variable i in the C code, then we get the same result from using this method. As a consequence the variable pre and i in the parameters of the toy language operations, have become obsolete.

Similarly to this, checking if $pre = pre + w[i]$ in the toy language code is equivalent to checking $strlen(w)$ in our C code. Both are checking whether we have reached the end of the word we are checking/adding/deleting, with the C code using a library function to achieve this same task. This is due to string comparison in C being much less simple than in the toy language.

Finally, we moved the check for whether w is in D, in the c code to after the first if condition in its relative positions in delWord and checkWord. This was mostly an aesthetic change.

4.2 Derived C source code

```
#include "dict.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int getIndex (char c);

Dict newdict()
{
        Dict node = malloc(sizeof(struct __tnode__));

        node->eow = FALSE;
        int i = 0;
        while (i < VECSIZE)
        {
                node->cvec[i] = NULL;
                i++;
        }

        return node;
}

void addword (const Dict r, const word w)
{
        if (strlen(w) == 1) {
                r->eow = TRUE;
                return;
        }

        Dict curr = r;
        char* buf = w;

        int i = getIndex(buf[0]);
        if (curr->cvec[i] == NULL) {
                curr->cvec[i] = newdict();
                buf++;
                addword(curr->cvec[i], buf);
        } else {
                buf++;
                addword(curr->cvec[i], buf);
        }
}

bool checkword (const Dict r, const word w)
```

```
{
        if ( strlen(w) == 1 ) {
                if ( r->eow == TRUE )
                        return TRUE;
                else
                        return FALSE;
        } else {
                char* buf = w;
                buf++;
                int i = getIndex(w[0]);
                if (r->cvec[i] == NULL) return FALSE;
                return checkword(r->cvec[i], buf);
        }
}

void delword (const Dict r, const word w)
{
        if ( strlen(w) == 1 && r->eow == TRUE) {
                r->eow = FALSE;
        } else {
                char* buf = w;
                buf++;
                int i = getIndex(w[0]);
                if (r->cvec[i] == NULL) return;
                delword(r->cvec[i], buf);
        }
}

int getIndex (char c)
{
        return c - 'a';
}

void barf(char *s)
{
        fprintf(stderr, "%s", s);
}
```