Assignment 3 - Trie Harder

Rahil Agrawal z5165505
Aditya Karia z5163287

COMP2111 18s1

1 Task 1 -Syntactic Data Type Dict

We define a syntactic data type **Dict** that encapsulates a dictionary set W as follows :

$$\mathbf{Dict} = (init^{Dict}, (W, x : [pre_j^{Dict}, post_j^{Dict}]_{j \in J}))$$

which consists of an initialisation predicate $init^{Dict} = (W = \{\})$ and the following operations:
proc addword^{Dict}(value w) . $W : [True, W = W_0 \cup w]$
proc checkword^{Dict}(value W, value w, result b) . $b : [True, b = (W = \{\})]$
proc delword^{Dict}(value w) . $W : [W \neq \{\}, W = W_0 \setminus w]$

2 Task 2 - Refinement to DictA

Were refining this to a data type DictA where we replace W with a Trie t. From Ass3 2018 S1 Specification, "A *trie domain* is a prefixclosed 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."
The correspondance between the two data types is captured by the function $f : T \mapsto P(L^*)$ given by :

$$f(t) = \{w \in L^* \mid w \in \mathbf{dom}t \wedge t(w) = 1\}$$

Also, from Ass3 2018 S1 Specification, "Formally, we write $v \leq w$ if word $v \in L^*$ is a prefix of $w \in L^*$, i.e., $\exists v' \in L^* \, (vv' = w)$. We write B for 0, 1 where 0 represents false and 1 true."
We define our With the aforementioned facts in mind, we define a concrete syntactic data type **DictA** that encapsulates a trie t as follows :

$$\mathbf{DictA} = (init^{DictA}, (t, x : [pre_j^{DictA}, post_j^{DictA}]_{j \in J}))$$

which consists of an initialisation predicate $init^{DictA} = (t = \{\})$ and the following operations:
$proc\ addword^{DictA}$(value w) . $t : [True, post(addword^{DictA})]$ where

$$post(addword^{DictA}) = \mathbf{dom} t = \mathbf{dom} t_0 \cup \{v \in L^* \mid v \leq w\} \wedge t = t_0 \cup \{v \mapsto t(v) \mid v \in \mathbf{dom} t \ \wedge v < w \wedge t_0(v) = 1 \Rightarrow t(v) = 1$$
$$v < w \wedge t_0(v) = 0 \Rightarrow t(v) = 0$$
$$v < w \wedge t_0(v) \neq 0, 1 \Rightarrow t(v) = 0$$
$$v = w \Rightarrow t(v) = 1\}$$

$proc\ checkword^{DictA}$(value t, value w, result b) . $b : [True, b = (t(w) = 1)]$
$proc\ delword^{DictA}$(value w) . $t : [t \neq \{\}, t(w) = 0]$

That this indeed is a refinement requires checking the relevant proof obligations:

$$init^{DictA} \Rightarrow init^{Dict}[^{f(t)}/_W] \tag{1}$$
$$pre_j^{Dict}[^{f(t)}/_W] \Rightarrow pre_j^{DictA}, for\ j \in J \tag{2}$$
$$pre_j^{Dict}[^{f(t_0),x_0}/_{W,x}] \wedge post_j^{DictA} \Rightarrow post_j^{Dict}[^{f(t_0),f(t)}/_{W_0,W}], for\ j \in J \tag{3}$$

We begin with (1).

$$\begin{aligned}
& init^{DictA}[^{f(t_0),x_0}/_{W,x}] = f(t) \neq \{\} \\
\Rightarrow\ & \langle \text{def. of f} \rangle \\
& f(t) = \{\} \Rightarrow \\
\Rightarrow\ & \langle \text{def. of Dict} \rangle \\
& init^{Dict}[^{f(t)}/_W]
\end{aligned}$$

Condition (2) is only required to be proven when the concrete pre-condition is non-trivial (not True). This is only the case in delword.

$$\begin{aligned}
& pre_{delword}^{Dict}[^{f(t_0),x_0}/_{W,x}] = f(t) \neq \{\} \\
\Rightarrow\ & \langle \text{def. of f} \rangle \\
& w \in \mathbf{dom} t \wedge t(w) = 1 \\
\Rightarrow\ & \langle \text{def. of trie} \rangle \\
& t \neq \{\} \\
\Rightarrow\ & \langle \text{def. of } pre_{delword}^{DictA} \rangle \\
& pre_{delword}^{DictA}
\end{aligned}$$

Finally, condition (4) needs to be checked for all operations.

For addword, we prove

$$
\begin{aligned}
& pre^{Dict}_{addword}[^{f(t)}/_{W}] \wedge post^{DictA}_{addword} = True \wedge \mathbf{dom}t = \mathbf{dom}t_0 \cup \{v \in L^* \mid v \leq w\} \wedge t = t_0 \cup \\
& \qquad \{v \mapsto t(v) \mid v \in \mathbf{dom}t \ \wedge v < w \wedge t_0(v) = 1 \Rightarrow t(v) = 1 \\
& \qquad\qquad v < w \wedge t_0(v) = 0 \Rightarrow t(v) = 0 \\
& \qquad\qquad v < w \wedge t_0(v) \neq 0, 1 \Rightarrow t(v) = 0 \\
& \qquad\qquad v = w \Rightarrow t(v) = 1\} \\
\Rightarrow\ & \langle \text{Trie is the same as old trie but we added the prefixes of w} \rangle \\
& \langle \text{without changing their old mappings} \rangle \\
& \langle \text{New prefixes are mapped to 0 and w is mapped to 1.} \rangle \\
& \langle \text{This means we added a word w if it did not already exist, def. of trie and f} \rangle \\
& f(t) = f(t_0) \cup w \\
\Rightarrow\ & \langle \text{Definition of } addword^{Dict} \rangle \\
& post_{addword^{Dict}}[^{f(t_0), f(t)}/_{W_0, W}]
\end{aligned}
$$

For checkword, we prove

$$
\begin{aligned}
& pre^{Dict}_{checkword}[^{f(t)}/_{W}] \wedge post^{DictA}_{checkword} = True \wedge b = (t(w) = 1) \\
\Rightarrow\ & \langle \text{b is 1 if w is in } \mathbf{dom}t \text{ and trie maps w to 1, 0 otherwise. def. of trie.} \rangle \\
\Rightarrow\ & \langle \text{b = 1 means w is in our set of words. def of f} \rangle \\
& b = (w \in f(t)) \\
\Rightarrow\ & \langle \text{def. of } checkword^{Dict} \rangle \\
& post_{checkword^{Dict}}[^{f(t_0), f(t)}/_{W_0, W}]
\end{aligned}
$$

For delword, we prove

$$
\begin{aligned}
& pre^{Dict}_{delword}[^{f(t)}/_{W}] \wedge post^{DictA}_{delword} = W \neq \{\} \wedge t(w) = 0 \\
\Rightarrow\ & \langle \text{def of f} \rangle \\
& w \notin f(t) \\
\Rightarrow\ & \langle \text{Clearly} \rangle \\
& f(t) = f(t_0) \setminus w \\
\Rightarrow\ & \langle \text{def. of } delword^{Dict} \rangle \\
& post_{delword^{Dict}}[^{f(t_0), f(t)}/_{W_0, W}]
\end{aligned}
$$

3 Task 3 - Derivation

$\textbf{proc}\ FunctionName(\textbf{value}\ n, \textbf{result}\ r) \cdot \llcorner n, r, x : [\ Pre, Post\ ]\lrcorner_{(1)}$

$(1) \sqsubseteq \quad \langle\textbf{Rule}\rangle$

$\llcorner r, x : [\ NewPre, NewPost\ ]\lrcorner_{(2)}$

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

$\llcorner i, x, r : [\ Pre, Blah\ ]\lrcorner_{(3)};$

$\llcorner i, x : [\ Blah, Post\ ]\lrcorner_{(4)}$

$\sqsubseteq \quad \langle\textbf{ass - (1)}\rangle$

$\mathbf{i := 1}$

$\mathbf{x := 13}$

$\sqsubseteq \quad \langle\textbf{seq}\rangle$

$\llcorner s, x : [\ pre(16), s = 0 \wedge x > 0\ ]\lrcorner_{(18)};$

$\llcorner a, i, r, s, x : [\ x > 0 \wedge s = 0, post(16)\ ]\lrcorner_{(19)}$

$(5) \sqsubseteq \quad \langle\textbf{while}\rangle$

$\textbf{while j} \neq \textbf{r do}$

$\quad \llcorner r, j : [\ Inv_2 \wedge j \neq r, Inv_2\ ]\lrcorner_{(8)}$

od;

$(9) \sqsubseteq \quad \langle\textbf{if}\rangle$

if Gaurd

then $\llcorner a : [\text{Gaurd} \wedge pre(9), post(9)]\lrcorner_{(11)}$

else $\llcorner a : [\text{Not Gaurd} \wedge pre(9), post(9)]\lrcorner_{(12)}$

fi;

We gather the code for the procedure body of blah:

$$
\begin{array}{l}
\mathbf{blah}(\mathbf{r}, \mathbf{n}) : \\
\quad var\ i := 1; \\
\quad var\ x := 13; \\
\quad r := 13; \\
\quad \textbf{while } j \neq r \textbf{ do} \\
\quad\quad x := x + 1; \\
\quad\quad var\ a := 1; \\
\quad\quad isPrime(x, a); \\
\quad\quad \textbf{if } a = 1 \textbf{ then} \\
\quad\quad\quad var\ s := 0; \\
\quad\quad\quad reversen(x, s); \\
\quad\quad\quad var\ b := 1; \\
\quad\quad\quad isPrime(s, b); \\
\quad\quad\quad \textbf{if } b = 1 \wedge s \neq x \textbf{ then} \\
\quad\quad\quad\quad i := i + 1; \\
\quad\quad\quad\quad r := x; \\
\quad \textbf{od};
\end{array}
$$

Also, we gather the code for the procedure body of blah:

$$
\begin{array}{l}
\mathbf{isPrime}(\mathbf{r}, \mathbf{j}) : \\
\quad var\ j := 2; \\
\quad \textbf{while } j \neq r \textbf{ do} \\
\quad\quad \textbf{if } (r \bmod j) = 0 \textbf{ then} \\
\quad\quad\quad a := 0; \\
\quad\quad j := j + 1; \\
\quad \textbf{od};
\end{array}
$$

We have derived our code. However we need to prove some refinements.

3.1 Implication 1: $[BLAH, BLAH] \sqsubseteq BLAH$

To prove:$BLAH \Rightarrow BLAH$

Proof:

$LHS = BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

RHS

3.2 Implication 2: $[BLAH, BLAH] \sqsubseteq BLAH$

To prove:$BLAH \Rightarrow BLAH$

Proof:

$LHS = BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

$BLAH$

$\Rightarrow$ ⟨BLAH⟩

RHS

4 Task 4 - C Code

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

void newdict(Dict *dp) {
    *dp = malloc(sizeof(TNode));
    if (*dp == NULL) {
        return;
    }
    for (int i = 0; i < VECSIZE; i++) {
        ((*dp)->cvec)[i] = NULL;
    }
    (*dp)->eow = FALSE;
}

// void addword (const Dict r, const word w) {
// Dict loop = r;
// int i;
// for (i = 0; w[i] != '\0'; i++) {
// if ((loop->cvec)[w[i]-'a'] == NULL)
// newdict(&((loop->cvec)[w[i] - 'a']));
// loop = (loop->cvec)[w[i] - 'a'];
// }
// loop->eow = TRUE;
// }

void addr (Dict r, const word w, int i) {
    if (w[i] == '\0') {
        r->eow = TRUE;
        return;
    } else {
        if ((r->cvec)[w[i]-'a'] == NULL) newdict(&((r->cvec)[w[i] - 'a']));
        r = (r->cvec)[w[i]-'a'];
        i=i+1;
        addr(r, w, i);
    }
}

void addword(const Dict r, const word w) {
    int i = 0;
    addr(r, w, i);
```

```
}


bool checkr(Dict r, const word w, int i) {
    if (r == NULL) return FALSE;
    if (w[i] == '\0') {
        if (r->eow == TRUE){
            return TRUE;
        }
        return FALSE;
    } else {
        r = (r->cvec)[w[i]-'a'];
        i=i+1;
        return checkr(r, w, i);
    }
}

bool checkword (const Dict r, const word w) {
    int i = 0;
    return checkr(r, w, i);
}



void delr(Dict r, const word w, int i) {
    if (r == NULL) return;
    if (w[i] == '\0') {
        r->eow = FALSE;
        return;
    } else {
        r = (r->cvec)[w[i]-'a'];
        i=i+1;
        delr(r, w, i);
    }
}

void delword (const Dict r, const word w) {
    int i = 0;
    delr(r, w, i);
}


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

```
}

void printDictR(const Dict r, char str[], int level)
{
    // If node is leaf node, it indiicates end
    // of string, so a null charcter is added
    // and string is displayed
    if (r->eow == TRUE)
    {
        str[level] = '\0';
        printf("%s\n", str);
    }

    int i;
    for (i = 0; i < VECSIZE; i++)
    {
        if (r->cvec[i] != NULL)
        {
            str[level] = i + 'a';
            printDictR(r->cvec[i], str, level + 1);
        }
    }
}

void printDict(const Dict r) {
    char str[100];
    int level = 0;
    printDictR(r, str, level);
}
```