

Assignment 3 - Trie Harder

Rahil Agrawal z5165505

Aditya Karia z5163287

COMP2111 18s1

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) . $b : [True, b = (W = \{w\})]$

proc delword^{Dict}(value w) . $W : [W \neq \{w\}, W = W_0 \setminus w]$

2 Refinement to DictA

We're 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 **domt** 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 :

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

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:

$proc\ addword^{DictA}(\text{value } w) . t : [True, post(addword^{DictA})]$ where

$$post(addword^{DictA}) = \mathbf{dom}t = \mathbf{dom}t_0 \cup \{v \in L^* \mid v \leq w\} \wedge t = t_0 \wedge \\ \forall v \in \mathbf{dom}t (t(v) = 1 \iff (w = y \vee (y \in \mathbf{dom}t_0 \wedge t_0(y) = 1)))$$

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

$proc\ delword^{DictA}(\text{value } w) . t : [t \neq \{\}, t(w) = 0]$

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

$$init^{DictA} \Rightarrow init^{Dict}[f(t)/w] \tag{1}$$

$$pre_j^{Dict}[f(t)/w] \Rightarrow pre_j^{DictA}, \text{ for } j \in J \tag{2}$$

$$pre_j^{Dict}[f(t_0), x_0 / w, x] \wedge post_j^{DictA} \Rightarrow post_j^{Dict}[f(t_0), f(t) / w_0, w], \text{ for } j \in J \tag{3}$$

We begin with (1).

$$init^{DictA}[f(t_0), x_0 / w, x] = f(t) \neq \{\} \\ \Rightarrow \langle \text{def. of } f \rangle \\ f(t) = \{\} \Rightarrow \\ \Rightarrow \langle \text{def. of Dict} \rangle \\ init^{Dict}[f(t)/w]$$

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.

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

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

For addword, we prove

$$\begin{aligned}
& \textcolor{red}{pre}_{\text{addword}}^{\text{Dict}}[f(t)/w] \wedge \textcolor{blue}{post}_{\text{addword}}^{\text{Dict}A} = \text{True} \wedge \mathbf{dom}t = \mathbf{dom}t_0 \cup \{v \in L^* \mid v \leq w\} \wedge \\
& \forall v \in \mathbf{dom}t \ (t(v) = 1 \iff (w = v \vee (v \in \mathbf{dom}t_0 \wedge t_0(v) = 1))) \\
\Rightarrow & \langle \text{Trie is the same as old trie but we added the prefixes of } w \rangle \\
& \langle \text{without changing their old mappings} \rangle \\
& \langle \text{New prefixes are mapped to 0 and } w \text{ is mapped to 1.} \rangle \\
& \langle \text{This means we added a word } w \text{ if it did not already exist, def. of trie and } f \rangle \\
& f(t) = f(t_0) \cup w \\
\Rightarrow & \langle \text{Definition of } \textcolor{green}{addword}^{\text{Dict}} \rangle \\
& \textcolor{red}{post}_{\text{addword}^{\text{Dict}}} [f(t_0), f(t) / w_0, W]
\end{aligned}$$

For checkword, we prove

$$\begin{aligned}
& \textcolor{red}{pre}_{\text{checkword}}^{\text{Dict}}[f(t)/w] \wedge \textcolor{blue}{post}_{\text{checkword}}^{\text{Dict}A} = \text{True} \wedge b = (t(w) = 1) \\
\Rightarrow & \langle b \text{ is 1 if } w \text{ is in } \mathbf{dom}t \text{ and trie maps } w \text{ to 1, 0 otherwise. def. of trie.} \rangle \\
\Rightarrow & \langle b = 1 \text{ means } w \text{ is in our set of words. def of } f \rangle \\
& b = (w \in f(t)) \\
\Rightarrow & \langle \text{def. of } \textcolor{green}{checkword}^{\text{Dict}} \rangle \\
& \textcolor{red}{post}_{\text{checkword}^{\text{Dict}}} [f(t_0), f(t) / w_0, W]
\end{aligned}$$

For delword, we prove

$$\begin{aligned}
& \textcolor{red}{pre}_{\text{delword}}^{\text{Dict}}[f(t)/w] \wedge \textcolor{blue}{post}_{\text{delword}}^{\text{Dict}A} = W \neq \{\} \wedge t(w) = 0 \\
\Rightarrow & \langle \text{def of } f \rangle \\
& w \notin f(t) \\
\Rightarrow & \langle \text{Clearly} \rangle \\
& f(t) = f(t_0) \setminus w \\
\Rightarrow & \langle \text{def. of } \textcolor{green}{delword}^{\text{Dict}} \rangle \\
& \textcolor{red}{post}_{\text{delword}^{\text{Dict}}} [f(t_0), f(t) / w_0, W]
\end{aligned}$$

3 Derivation

Before we refine the code for our functions for **DictA**, we define certain predicates to help us during derivation.

We associate a natural number i with each element x of a set X and define $y_X^{(i)}$ to be the i^{th} element.

The total number of elements in a set X is given by $size(X)$.

```

proc delword(value  $w$ ) ·  $t$  : [  $t \neq \{\}$ ,  $t(w) = 0$  ]
⊆    ⟨i-loc, seq⟩
     $t, i$  : [  $t \neq \{\}$ ,  $i = 0$  ] ;
⊆    ⟨ass⟩
    var  $i := 0$ ;
     $t, i$  : [  $i = 0$ ,  $t(w) = 0$  ]
⊆    ⟨proc,  $i = 0 \Rightarrow i \leq size(t)$ ⟩
    delR( $w, i$ );

proc checkword(value  $b$ , value  $w$ ) ·  $b, t$  : [  $TRUE$ ,  $b = (t(w) = 1)$  ]
⊆    ⟨i-loc, seq⟩
     $b, t, i$  : [  $TRUE$ ,  $i = 0$  ] ;
⊆    ⟨ass⟩
    var  $i := 0$ ;
     $t, i$  : [  $i = 0$ ,  $b = (t(w) = 1)$  ]
⊆    ⟨proc,  $i = 0 \Rightarrow i \leq size(t)$ ⟩
    checkR( $w, b, i$ );

proc addword(value  $w$ ) ·  $b, t$  : [  $TRUE$ ,  $post_{addword}^{DictA}$  ]
⊆    ⟨i-loc, seq⟩
     $b, t, i$  : [  $TRUE$ ,  $i = 0$  ] ;
⊆    ⟨ass⟩
    var  $i := 0$ ;
     $t, i$  : [  $i = 0$ ,  $post_{addword}^{DictA}$  ]
⊆    ⟨proc,  $i = 0 \Rightarrow i \leq size(t)$ ⟩
    addR( $w, b, i$ );

```

```

    proc delR(value w, value i) · t : [ i ≤ size(t), t(w) = 0 ]
⊆    ⟨if⟩
    if i ≠ size(t)
    then  $\sqcup t, i : [i < \text{size}(t), t(w) = 0] \sqcup_{(1)}$ 
    else  $t, i : [i = \text{size}(t), t(w) = 0]$ 
⊆    ⟨skip - Proof(1)⟩
    skip;
fi;
(1) ⊆    ⟨if⟩
    if  $y_t^i = w \mapsto t(w)$ 
    then  $t, i : [i < \text{size}(t) \wedge y_t^i = w \mapsto t(w), t(w) = 0]$ 
⊆    ⟨ass,  $0 = 0$ ⟩
     $t(w) := 0$ ;
    else  $\sqcup t, i : [i < \text{size}(t) \wedge y_t^i \neq w \mapsto t(w), t(w) = 0] \sqcup_{(2)}$ 
    fi;
(2) ⊆    ⟨seq, con c⟩
     $t, i : [i < \text{size}(t) \wedge y_t^i \neq w \mapsto t(w) \wedge i = c, i = c + 1];$ 
⊆    ⟨ass,  $c = i_0 \wedge i = c + 1 \Rightarrow i = i_0 + 1$ ⟩
     $i := i + 1$ ;
     $t, i : [i = c + 1, t(w) = 0]$ 
⊆    ⟨proc,  $c = i < \text{size}(t) \Rightarrow i + 1 \leq \text{size}(t)$ ⟩
    delR(w, i);

proc checkR(value w, result b, value i) · t : [ i ≤ size(t), b = (t(w) = 1) ]
⊆    ⟨if⟩
    if i ≠ size(t)
    then  $\sqcup t, i : [i < \text{size}(t), b = (t(w) = 1)] \sqcup_{(1)}$ 
    else  $t, i : [i = \text{size}(t), b = (t(w) = 1)]$ 
⊆    ⟨skip - Proof(2)⟩
    skip;
fi;

```

$$\begin{aligned}
(1) &\sqsubseteq \langle \text{if} \rangle \\
&\quad \text{if } y_t^i = w \mapsto 1 \\
&\quad \text{then } t, i : [i < \text{size}(t) \wedge y_t^i = w \mapsto 1, b = (t(w) = 1)] \\
&\sqsubseteq \langle \text{ass}, w \mapsto 1 \Rightarrow t(w) = 1 \Rightarrow b = \text{TRUE} \rangle \\
&\quad \mathbf{b} := \text{TRUE}; \\
&\quad \text{else } \perp t, i : [i = \text{size}(t) \wedge y_t^i \neq w \mapsto 1, b = (t(w) = 1)] \lrcorner (2) \\
&\quad \mathbf{fi}; \\
(2) &\sqsubseteq \langle \text{seq, con } \mathbf{c} \rangle \\
&\quad t, i : [i < \text{size}(t) \wedge y_t^i \neq w \mapsto 1 \wedge i = c, i = c + 1]; \\
&\sqsubseteq \langle \text{ass}, c = i_0 \wedge i = c + 1 \Rightarrow i = i_0 + 1 \rangle \\
&\quad \mathbf{i} := \mathbf{i} + 1; \\
&\quad t, i : [i = c + 1, b = (t(w) = 1)] \\
&\sqsubseteq \langle \text{proc}, c = i < \text{size}(t) \Rightarrow i + 1 \leq \text{size}(t) \rangle \\
&\quad \text{checkR}(\mathbf{w}, \mathbf{b}, \mathbf{i});
\end{aligned}$$

Before we derive code for `addR`, we define S to be the set of all prefixes of a word w .

$$\begin{aligned}
&\text{proc } \text{addR}(\text{value } w, \text{value } i) \cdot t : [i \leq \text{size}(t), \text{post}_{\text{addword}}^{\text{Dict}A}] \\
&\sqsubseteq \langle \text{if} \rangle \\
&\quad \text{if } i \neq \text{size}(S) \\
&\quad \text{then } \perp t, i : [i < \text{size}(S), \text{post}_{\text{addword}}^{\text{Dict}A}] \lrcorner (1) \\
&\quad \text{else } t, i : [i < \text{size}(S), \text{post}_{\text{addword}}^{\text{Dict}A}] \\
&\sqsubseteq \langle \text{skip - Proof(3)} \rangle \\
&\quad \text{skip}; \\
&\quad \mathbf{fi}; \\
(1) &\sqsubseteq \langle \text{seq, con } \mathbf{G} \rangle \\
&\quad t, i : [i < \text{size}(S) \wedge G = \text{dom}t, i < \text{size}(S) \wedge \text{dom}t = G \cup y_S^i]; \\
&\sqsubseteq \langle \text{ass}, G = \text{dom}t_0 \wedge \text{dom}t = G \cup y_S^i \rangle \\
&\quad \text{dom}t := \text{dom}t_0 \cup y_S^i; \\
&\quad \perp t, i : [i < \text{size}(S) \wedge \text{dom}t = G \cup y_S^i, \text{post}_{\text{addword}}^{\text{Dict}A}] \lrcorner (2)
\end{aligned}$$

(2) \sqsubseteq $\langle \text{if} \rangle$
 if $t_0(y_S^i) \neq 0, 1$
 then $\sqsubseteq t, i : [i < \text{size}(S) \wedge \text{dom}t = G \cup y_S^i \wedge t(y_S^i) \neq 0, 1, \text{post}_{\text{addword}}^{\text{Dict}A}] \sqsubseteq (3)$
 else $\sqsubseteq t, i : [i < \text{size}(S) \wedge \text{dom}t = G \cup y_S^i \wedge t(y_S^i) = 0, 1, \text{post}_{\text{addword}}^{\text{Dict}A}] \sqsubseteq (4)$
 fi;
 (3) \sqsubseteq $\langle \text{if} \rangle$
 if $y_S^i = w$
 then $\sqsubseteq t, i : [i < \text{size}(S) \wedge \text{dom}t = G \cup y_S^i \wedge t(y_S^i) \neq 0, 1 \wedge y_S^i = w, \text{post}_{\text{addword}}^{\text{Dict}A}]$
 $\sqsubseteq \langle \text{ass}, t_0(y_S^i) \neq 0, 1 \wedge y_S^i = w \Rightarrow \text{Add to trie and map to 1 } (\because w \text{ is added}) \rangle$
 $\sqsubseteq t = t_0 : y_S^i \mapsto 1$
 else $\sqsubseteq t, i : [i < \text{size}(S) \wedge \text{dom}t = G \cup y_S^i \wedge t(y_S^i) \neq 0, 1 \wedge y_S^i \neq w, \text{post}_{\text{addword}}^{\text{Dict}A}]$
 $\sqsubseteq \langle \text{ass}, t_0(y_S^i) \neq 0, 1 \wedge y_S^i \neq w \Rightarrow \text{Add to trie and map to 0 } (\because \text{prefix of } w \text{ is added}) \rangle$
 $\sqsubseteq t = t_0 : y_S^i \mapsto 0$
 fi;
 (4) \sqsubseteq $\langle \text{seq, con } c \rangle$
 $\sqsubseteq t, i : [i < \text{size}(S) \wedge \text{dom}t = \text{Given} \cup y_S^i \wedge t(y_S^i) = 0, 1 \wedge i = c, i = c + 1];$
 $\sqsubseteq \langle \text{ass}, c = i_0 \wedge i = c + 1 \Rightarrow i = i_0 + 1 \rangle$
 $\sqsubseteq i := i + 1;$
 $\sqsubseteq t, i : [i = c + 1, \text{post}_{\text{addword}}^{\text{Dict}A}]$
 $\sqsubseteq \langle \text{proc}, c = i < \text{size}(t) \Rightarrow i + 1 \leq \text{size}(t) \rangle$
 $\sqsubseteq \text{addR}(w, i);$

4 C Code

```

1 #include "dict.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 void newdict(Dict *dp) {
6     *dp = malloc(sizeof(TNode));
7     if (*dp == NULL) {
8         return;
9     }
10    for (int i = 0; i < VECSIZE; i++) {
11        ((*dp)->cvec)[i] = NULL;
12    }
13    (*dp)->eow = FALSE;
14 }
```

```

15
16 void addR (Dict r, const word w, int i) {
17     if (w[i] == '\0') {
18         r->eow = TRUE;
19         return;
20     } else {
21         if ((r->cvec)[w[i] - 'a'] == NULL) newdict(&((r->cvec)[w[i] - 'a']));
22         r = (r->cvec)[w[i] - 'a'];
23         i = i + 1;
24         addR(r, w, i);
25     }
26 }
27
28 bool checkR(Dict r, const word w, int i) {
29     if (r == NULL) return FALSE;
30     if (w[i] == '\0') {
31         if (r->eow == TRUE) {
32             return TRUE;
33         }
34         return FALSE;
35     } else {
36         r = (r->cvec)[w[i] - 'a'];
37         i = i + 1;
38         return checkR(r, w, i);
39     }
40 }
41
42 void delR(Dict r, const word w, int i) {
43     if (r == NULL) return;
44     if (w[i] == '\0') {
45         r->eow = FALSE;
46         return;
47     } else {
48         r = (r->cvec)[w[i] - 'a'];
49         i = i + 1;
50         delR(r, w, i);
51     }
52 }
53
54 void printDictR(const Dict r, char str[], int level)
55 {
56     if (r->eow == TRUE)
57     {
58         str[level] = '\0';

```



```

59     printf("%s\n", str);
60 }
61
62 int i;
63 for (i = 0; i < VECSIZE; i++)
64 {
65     if (r->cvec[i] != NULL)
66     {
67         str[level] = i + 'a';
68         printDictR(r->cvec[i], str, level + 1);
69     }
70 }
71 }
72
73 void barf(char *s) {
74     fprintf(stderr, "%s\n", s);
75 }
76
77 void addword(const Dict r, const word w) {
78     int i = 0;
79     addR(r, w, i);
80 }
81
82 bool checkword (const Dict r, const word w) {
83     int i = 0;
84     return checkR(r, w, i);
85 }
86
87 void delword (const Dict r, const word w) {
88     int i = 0;
89     delR(r, w, i);
90 }
91
92 void printDict(const Dict r) {
93     char str[100];
94     int level = 0;
95     printDictR(r, str, level);
96 }

```