Assignment 3

Ruofei HUANG(z5141448)   Anqi ZHU(z5141541)

May 30, 2018

1 Task 1

1.1 Some useful symbol definitions about manipulating words

A **word**, as mentioned in the assignment specification, is defined as a finite sequence of letters over $L = \{'a', ..,' z'\}$. The specification already defines the relationship of '$\leq$' between a **word** v and a **word** w (this means v is a prefix of w or w itself if $v \leq w$).

So we would like to further this definition and define a relationship of '$<$' when v is a proper prefix of w which cannot be w itself if $v < w$. That means:

$$v < w \Leftrightarrow v \leq w \wedge v \neq w$$

We also would like to define a symbol $|w|$ that represents the length of a *word* w.We formally define this by:

$$|w| = \begin{cases} 0 & \text{if } w = \epsilon \\ 1 + |w'| & \text{else} \end{cases}$$

$$where\ \exists l \in L\ \{w = w'\ l\}$$

1.2 Syntactic(Abstract) Data Type *Dict*

Inspired by the program sketch and the assignment statement, we could describe the syntactic data type *Dict* as below (the encapsulated state would be a dictionary word set W).

$$Dict = (W = \phi,$$
$$\left(\begin{array}{l} \textbf{proc } addword^{Dict}(\textbf{word } w) \cdot b, W : [\text{TRUE}, b = b_0 \wedge W = W_0 \cup \{w\}] \\ \textbf{func } checkword^{Dict}(\textbf{word } w) : \mathbb{B} \cdot \\ \qquad \textbf{var } b \cdot\ b, W : [\text{TRUE}, b = (w \in W) \wedge W = W_0];\ \textbf{return } b \\ \textbf{proc } delword^{Dict}(\textbf{word } w) \cdot b, W : [w \in W, b = b_0 \wedge W = W_0 \backslash \{w\}] \end{array}\right))$$

2 Task 2

2.1 Data Type Refinement

Now we would like to refine *Dict* to a second data type *DictA* where we replace W with a trie t. We would also like to define the domain of a t as $\mathbf{dom}(t)$. We shall use this definition later in our refinement.

2.1.1 Inductive Relation Predicate

The correspondence between the two state space W and t is captured by the inductively defined predicate as follows:

$$r = (W = \{w \in \mathbf{dom}(t) | t(w) = 1\})$$

We can translate this into a relation function that transfers a concrete state space t to an abstract state space W. The function is as follows:

$$f(t) = \{w \in \mathbf{dom}(t) | t(w) = 1\}$$

With that in mind, we can propose the initialisation predicate and corresponding operations of *DictA*.

2.1.2 Initialisation Predicate

We would like to define the initialisation predicate of DictA as follows:

$$init^{DictA} = (t := \{\epsilon \mapsto 0\})$$

2.1.3 Operations

We would like to define the operations of DictA as follows:

```
$\mathbf{proc}\ addword^{DictA}(\mathbf{word}\ w) \cdot b, t :$
    $[\text{TRUE}, b = b_0 \wedge t = t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}]$
$\mathbf{func}\ checkword^{DictA}(\mathbf{word}\ w) : \mathbb{B} \cdot \mathbf{var}\ b \cdot b, t :$
    $[\text{TRUE}, t = t_0 \wedge b = (\forall w' \leq w\, (w' \in \mathbf{dom}(t)) \wedge t(w) = 1)];\ \mathbf{return}\ b$
$\mathbf{proc}\ delword^{DictA}(\mathbf{word}\ w) \cdot b, t :$
    $[\text{TRUE}, b = b_0 \wedge (w \notin \mathbf{dom}(t) \vee t = (t : w \mapsto 0))]$
```

2.2 Proof of Refinement

Now we would like to start proving the refinements of the initialization and each operation from t to W. So that we can prove *DictA* is a data refinement of *Dict*.

2.2.1 Refinement proof for *init*

We start from proving the refinement between $init^{DictA}$ and $init^{Dict}$.

$$
\begin{aligned}
& init^{DictA} \Rightarrow init^{Dict}[^{f(t)}/_{W}] \\
\Leftrightarrow \quad & \langle \text{Definition of } init^{DictA} \text{ and } init^{Dict} \rangle \\
& \forall w \in \mathbf{dom}(t)\ (t(w) = 0) \Rightarrow W = \phi
\end{aligned}
$$

Then we move on to prove the refinement of defined operations of *Dict* and *DictA*. We don't need to prove the validity of the condition (3_f) for any operations since their preconditions are always TRUE. We only need to check the validity of the condition (4_f) in all three operations.

2.2.2 Refinement proof for *addword*

$$
\begin{aligned}
& pre_{addword^{Dict}}[^{f(t_0)}/_{W}] \wedge post_{addword^{DictA}} \\
\Leftrightarrow \quad & \langle \text{Definition of } addword^{Dict} \text{ and } addword^{DictA} \rangle \\
& \text{TRUE}[^{f(t_0)}/_{W}] \wedge b = b_0 \wedge t = t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\} \\
\Rightarrow \quad & \langle \text{Definition of } f \rangle \\
& f(t) = f(t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}) \\
\Leftrightarrow \quad & \langle \text{Logic, only } w \text{ maps to 1 in } t (\text{only } w \text{ is newly added into } W) \rangle \\
& f(t) = f(t_0) \cup \{w\} \\
\Leftrightarrow \quad & \langle \text{Definition of } addword^{Dict} \text{ and } addword^{DictA} \rangle \\
& post_{addword^{Dict}}[^{f(t_0), f(t)}/_{W_0, W}]
\end{aligned}
$$

2.2.3 Refinement proof for *checkword*

$$
\begin{aligned}
& pre_{checkword^{Dict}}[^{f(t_0)}/_{W}] \wedge post_{checkword^{DictA}} \\
\Leftrightarrow \quad & \langle \text{Definition of } checkword^{DictA} \text{ and } checkword^{Dict} \rangle \\
& \text{TRUE}[^{f(t_0)}/_{W}] \wedge t = t_0 \wedge b = (\forall w' \leq w\ (w' \in \mathbf{dom}(t)) \wedge t(w) = 1) \\
\Rightarrow \quad & \langle \text{Definition of } f \rangle \\
& b = (w \in f(t) \wedge t(w) = 1) \wedge t = t_0 \\
\Leftrightarrow \quad & \langle \text{Definition of } checkword^{DictA} \text{ and } checkword^{Dict} \rangle \\
& post_{checkword^{Dict}}[^{f(t_0), f(t)}/_{W_0, W}]
\end{aligned}
$$

2.2.4 Refinement proof for *delword*

$$
\begin{aligned}
& pre_{delword^{Dict}}[{}^{f(t_0)}/{}_{W}] \wedge post_{delword^{DictA}} \\
\Leftrightarrow \quad & \langle \text{Definition of } delword^{DictA} \text{ and } delword^{Dict} \rangle \\
& w \in f(t_0) \wedge b = b_0 \wedge (w \notin \mathbf{dom}(t) \vee t = (t : w \mapsto 0)) \\
\Rightarrow \quad & \langle \text{Definition of } f \rangle \\
& f(t) = f(t_0) \backslash \{w\} \\
\Leftrightarrow \quad & \langle \text{Definition of } delword^{DictA} \text{ and } delword^{Dict} \rangle \\
& post_{delword^{Dict}}[{}^{f(t_0), f(t)}/{}_{W_0, W}]
\end{aligned}
$$

3 Task 3

3.1 Pre-defined function calls

Before refining the operations in $DictA$ into toy language, we would like to first define some useful function calls that helps building our later refinement more close to the real c program constructions.

3.1.1 *POPWORD*

The semantic function $POPWORD$ returns a substring w' of the word w starting from the first letter to the $index$'th letter of w. The definition of the function and corresponding function call is as below:

Definition of $POPWORD$

$$POPWORD(w, i) = w' \qquad where\ w' \leq w \wedge |w'| = i$$

Definition of the function call $popWord$

```
func $popWord$(value $w$, value $i$) : word ·
    var $w'$ · $\llcorner w' : [i \leq |w|, w' \leq w \wedge |w'| = i]; \lrcorner_{(P1)}$ return $w'$
```

Specification of the procedure in $popWord$

$$
\begin{aligned}
(P1) \sqsubseteq \quad & \langle |w[0..i-1]| = i \wedge w[0..i-1] \leq w \rangle \\
& w' := w[0..i-1]
\end{aligned}
$$

This function helps us to build our refinement as close to the trie construction in c programs as possible. (In c programs, words are splitted into prefixes in increasing length and checked in the trie recursively.)

3.1.2 *doAddword*

Using the previous definition of $POPWORD$, we would like to develop a corresponding function call named $doAddword(\mathbf{var}\ w, \mathbf{var}\ index)$ that allows using a variable i to locate which prefix of w(or say which sub-trie of t) the procedure currently is checking at during the entire word-adding operation. This function will be called by the function *addword* of *DictA* to do most of the adding-word job.

All the prefixes of w should exist in t eventually, so if the current prefix $POPWORD(w, i)$ does not exist, *doAddword* will add the new prefix into t and continue searching for the next prefix $POPWORD(w, i+1)$. Considering this and the fact that t always initializes with ϵ included, it is guaranteed that every *doAddword* operation at i level will already have all prefixes of w with length no longer than i exist in t, which satisfies the precondition of the *doAddword* function itself. The definition of the function and the refinement of its linking procedure is as follows.

Definition of *doAddword*

$$\mathbf{proc}\ doAddword^{DictA}(\mathbf{value}\ w, \mathbf{value}\ index)$$

$$\llcorner t : \left[\begin{array}{l} 0 \leq index \leq |w| \wedge \forall w' \leq w \wedge |w'| \leq index\,(w' \in \mathbf{dom}(t))\,, \\ t = t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}] \end{array} \right] \lrcorner^{(A2)}$$

Specification of the procedure in *doAddword*

```
$(A2) \sqsubseteq$        $\langle \text{if} \rangle$
    **if** $index < |w|$
    **then** $\llcorner t : [index < |w| \wedge pre(A2), post(A2)] \lrcorner_{(A3-1)}$
    **else** $\llcorner t : [index \geq |w| \wedge pre(A2), post(A2)] \lrcorner_{(A3-2)}$
    **fi**
$(A3-1) \sqsubseteq$        $\langle \text{seq} \rangle$
    $\llcorner t : \begin{bmatrix} index < |w| \wedge pre(A2), \\ index \geq |w| \wedge pre(A2) \wedge POPWORD(w, index+1) \in \mathbf{dom}(t) \end{bmatrix} \lrcorner_{(A3-3)}$
    $\llcorner t : \begin{bmatrix} index \geq |w| \wedge pre(A2) \wedge POPWORD(w, index+1) \in \mathbf{dom}(t), \\ post(A2) \end{bmatrix} \lrcorner_{(A3-4)}$
$(A3-3) \sqsubseteq$        $\langle \text{if} \rangle$
    **if** $(POPWORD(w, index+1) \in \mathbf{dom}(t))$
    **then** $\llcorner t : \begin{bmatrix} pre(A3-3) \wedge POPWORD(w, index+1) \in \mathbf{dom}(t), \\ post(A3-3) \end{bmatrix} \lrcorner_{(A3-3-1)}$
    **else** $\llcorner t : \begin{bmatrix} pre(A3-3) \wedge POPWORD(w, index+1) \notin \mathbf{dom}(t), \\ post(A3-3) \end{bmatrix} \lrcorner_{(A3-3-2)}$
    **fi**
$(A3-3-1) \sqsubseteq$        $\langle POPWORD(w, index+1) \in \mathbf{dom}(t) \Rightarrow post(A3-3) = \text{TRUE} \rangle$
    $skip;$
$(A3-3-2) \sqsubseteq$        $\langle \text{Ass} \rangle$
    $t := t \cup \{popWord(w, index) \mapsto 0\};$
$(A3-4) \sqsubseteq$        $\langle \text{proc} \rangle$
    $doAddword(w, index+1)$
$(A3-2) \sqsubseteq$        $\langle index \geq |w| \wedge index \leq |w| \Rightarrow index = |w| \text{ and definition of } post(A2) \rangle$
    $t := t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\}$
```

We gather the code for the body of *doAddword*

```
if $index < |w|$
then
    if $POPWORD(w, index + 1) \in \mathbf{dom}(t)$
    then $skip$;
    else
        $t := t \cup \{popWord(w, index) \mapsto 0\}$;
    fi
    $doAddword(w, index + 1)$
else
    $t := t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\}$
fi
```

3.1.3 *doCheckword*

Similarly to *doAddword*, we would also develop a corresponding function call named *doCheckword*(**var** w, **var** $index$) which will search the complete word w in t based on the precondition that all prefixes of w with length no longer than *index* are guaranteed to exist in t. This function would be called by *checkword* of *DictA* to do most of the checking-word job. The definition of the function and the refinement of its linking procedure is as follows.

Definition of doCheckword

$$\mathbf{func}\ doCheckword(\mathbf{value}\ w, \mathbf{value}\ index : \mathbb{N}) : \mathbb{B} \cdot$$

$$\mathbf{var}\ b \cdot \llcorner b : \left[\begin{array}{l} 0 \leq index \leq |w| \wedge \\ \forall w' \leq w \wedge |w'| \leq index\, (w' \in \mathbf{dom}(t))\,, \\ b = (\forall w' \leq w\, (w' \in \mathbf{dom}(t)) \wedge t(w) = 1) \end{array} \right] \lrcorner_{(C2)}; \mathbf{return}\ b$$

Specification of the procedure in doCheckword

$(C2) \sqsubseteq$ $\langle \text{if} \rangle$

```
if $index < |w|$
then$\llcorner b : [index < |w| \wedge pre(C2), post(C2)]\lrcorner_{(C3-1)}$
else$\llcorner b : [index \geq |w| \wedge pre(C2), post(C2)]\lrcorner_{(C3-2)}$
fi
```

$(C3-1) \sqsubseteq$ $\langle \text{if} \rangle$

```
if $(POPWORD(w, index+1) \in \mathbf{dom}(t))$
then $\llcorner b : \begin{bmatrix} pre(C3-1) \wedge POPWORD(w, index+1) \in \mathbf{dom}(t), \\ post(C2) \end{bmatrix}\lrcorner_{(C3-1-1)}$
else $\llcorner b : \begin{bmatrix} pre(C3-1) \wedge POPWORD(w, index+1) \notin \mathbf{dom}(t), \\ post(C2) \end{bmatrix}\lrcorner_{(C3-1-2)}$
fi
```

$(C3-1-1) \sqsubseteq$ $\langle \text{ass, func} \rangle$

$b := doCheckword(w, index+1)$

$(C3-1-2) \sqsubseteq$ $\langle \text{Ass and } popWord(w, index+1) \notin \mathbf{dom}(t) \Rightarrow b = \text{FALSE} \rangle$

$b := \text{FALSE}$

$(C3-2) \sqsubseteq$ $\langle index \geq |w| \wedge index \leq |w| \Rightarrow index = |w| \text{ and definition of } post(C2) \rangle$

$b := (t(w) = 1)$

We gather the code of *doCheckword* below:

```
if  $index < |w|$
then
    if  $POPWORD(w, index+1) \in \mathbf{dom}(t)$
    then skip
    else return $doCheckword(w, index+1)$
    fi
else $b := (t(w) = 1)$
fi
```

3.1.4 *doDelword*

Similarly to *doCheckword*, we would also develop a corresponding function call named *doDelword*(**var** w, **var** *index*) which will set the result of $t(w)$ as 0. This function would be called by *delword* of *DictA* to do most of the deleting-word job. The definition of the function and the refinement of its linking procedure is as follows.

Definition of doDelword

proc $doDelword^{DictA}$(**value** w, **value** $index$)

$$\llcorner t : \begin{bmatrix} 0 \le index \le |w| \wedge \forall w' \le w \wedge |w'| \le index\,(w' \in \mathbf{dom}(t))\,, \\ (w \notin \mathbf{dom}(t) \vee t = t_0 : w \mapsto 0) \end{bmatrix} \lrcorner_{(D2)}$$

Specification of the procedure in doDelword

```
(D2) ⊑        ⟨if⟩
              if $index < |w|$
              then$\llcorner t : [index < |w| \wedge pre(D2), post(D2)]\lrcorner_{(D3-1)}$
              else$\llcorner t : [index \ge |w| \wedge pre(D2), post(D2)]\lrcorner_{(C3-2)}$
              fi
(D3 − 1) ⊑    ⟨if⟩
              if $(POPWORD(w, index + 1) \in \mathbf{dom}(t))$
              then $\llcorner t : \begin{bmatrix} pre(D3-1) \wedge POPWORD(w, index+1) \in \mathbf{dom}(t), \\ post(D2) \end{bmatrix} \lrcorner_{(D3-1-1)}$
              else $\llcorner t : \begin{bmatrix} pre(D3-1) \wedge POPWORD(w, index+1) \notin \mathbf{dom}(t), \\ post(D2) \end{bmatrix} \lrcorner_{(D3-1-2)}$
              fi
(D3 − 1 − 1) ⊑    ⟨ass, func⟩
              $doDelword(w, index + 1)$
(D3 − 1 − 2) ⊑    ⟨Ass and $POPWORD(w, index + 1) \notin \mathbf{dom}(t) \Rightarrow post(D2) = \text{TRUE}$ ⟩
              $skip$
(D3 − 2) ⊑    ⟨$index \ge |w| \wedge index \le |w| \Rightarrow index = |w|$ and definition of $post(D2)$⟩
              $t := (t : w \mapsto 0)$
```

We gather the code of *doDelword* below:

```
if  $index < |w|$
then
    if  $POPWORD(w, index + 1) \in \mathbf{dom}(t)$
    then $doDelword(w, index + 1)$
    else $skip$
    fi
else $t := (t : w \mapsto 0)$
fi
```

Now we could specify the initialisation and all operations of *DictA* into toy language, in which above functions will be called to do the corresponding jobs.

3.2 Specification of *init*

From the spec we have:

$$\begin{array}{ll} & \mathbf{dom}(t) = \{\epsilon\} \wedge f(t) = \phi \\ \sqsubseteq & \quad \langle \text{ass} \rangle \\ & t := \{\epsilon \mapsto 0\} \end{array}$$

3.3 Specification of *addword*

From the spec[1] we have:

$$\begin{array}{l} \mathbf{proc}\ addword^{DictA}(\mathbf{value}\ w)\cdot \\ \quad \llcorner b, t : [\text{TRUE}, b = b_0 \wedge t = t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}]_{\lrcorner(A1)} \end{array}$$

$$\begin{array}{rll} (A1) \sqsubseteq & & \langle \text{c-frame} \rangle \\ & & t : [\text{TRUE}, t = t_0 \backslash \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}] \\ \sqsubseteq & & \langle \text{if} \rangle \\ & & \mathbf{if}\ w = \epsilon \\ & & \mathbf{then}\ \llcorner b : \begin{bmatrix} pre(A1) \wedge w = \epsilon, \\ post(A1) \end{bmatrix}_{\lrcorner(A1-s)} \\ & & \mathbf{else}\ \llcorner b : \begin{bmatrix} pre(A1) \wedge w \neq \epsilon, \\ post(A1) \end{bmatrix}_{\lrcorner(A1-c)} \\ & & \mathbf{fi} \\ (A1 - s) \sqsubseteq & & \langle \epsilon \text{ already exists in } \mathbf{dom}(t) \rangle \\ & & t := (t : w \mapsto 1) \\ (A1 - c) \sqsubseteq & & \langle \text{proc, } 0 \leq |w| \text{ and } \epsilon \in \mathbf{dom}(t) \text{ since } init^{DictA} \rangle \\ & & doAddword(w, 0) \end{array}$$

We gather the code for the body of *addword*

```
if  $w = \epsilon$
then $t := (t : w \mapsto 1)$
else
    $doAddword(w, 0)$
fi
```

[1]Definition of this is in the Assignment 3 requirements of cs2111.

3.4 Specification of *checkword*

From the spec we have:

```
func $checkword^{DictA}$(value $w$) : $\mathbb{B}\cdot$
    var $b \cdot \llcorner b, t : [\text{TRUE}, b = (\forall w' \leq w\,(w' \in \mathbf{dom}(t)) \wedge t(w) = 1)];\lrcorner_{(C1)}$ return $b$
```

$(C1) \sqsubseteq$ ⟨c-frame⟩

$b : [\text{TRUE}, b = (\forall w' \leq w\,(w' \in \mathbf{dom}(t)) \wedge t(w) = 1)];$

$\sqsubseteq$ ⟨if⟩

```
if $w = \epsilon$
then $\llcorner b : \begin{bmatrix} pre(C1) \wedge w = \epsilon, \\ post(C1) \end{bmatrix} \lrcorner_{(C1-s)}$
else $\llcorner b : \begin{bmatrix} pre(C1) \wedge w \neq \epsilon, \\ post(C1) \end{bmatrix} \lrcorner_{(C1-c)}$
fi
```

$(C1-s) \sqsubseteq$ ⟨ϵ already exists in $\mathbf{dom}(t)$⟩

$b := (t(w) = 1)$

$(C1-c) \sqsubseteq$ ⟨proc, $0 \leq |w|$ and $\epsilon \in \mathbf{dom}(t)$ since $init^{DictA}$⟩

$b := doCheckword(w, 0);$

We gather the code for the body of *checkword*

```
if  $w = \epsilon$
then
    $b := (t(w) = 1);$
else
    $b := doCheckword(w, 0);$
fi
return $b$
```

3.5 Specification of *delword*

From the spec we have:

```
proc $delword^{DictA}$(value $w$)
    $\llcorner \cdot b, t : [\text{TRUE}, b = b_0 \wedge (w \notin \mathbf{dom}(t) \vee t := t : w \mapsto 0)]\lrcorner_{(D1)}$
```

$$
\begin{aligned}
(D1) \sqsubseteq \quad & \langle \text{c-frame} \rangle \\
& t : [\text{TRUE}, (w \notin \mathbf{dom}(t) \vee t = t_0 : w \mapsto 0)] \\
\sqsubseteq \quad & \langle \text{if} \rangle \\
& \mathbf{if}\ w = \epsilon \\
& \mathbf{then}\ \llcorner b : \left[\begin{array}{l} pre(D1) \wedge w = \epsilon, \\ post(D1) \end{array} \right] \lrcorner^{(D1-s)} \\
& \mathbf{else}\ \llcorner b : \left[\begin{array}{l} pre(D1) \wedge w \neq \epsilon, \\ post(D1) \end{array} \right] \lrcorner^{(D1-c)} \\
& \mathbf{fi} \\
(D1 - s) \sqsubseteq \quad & \langle \epsilon \text{ already exists in } \mathbf{dom}(t) \rangle \\
& t := (t : w \mapsto 0) \\
(D1 - c) \sqsubseteq \quad & \langle \text{proc, } 0 \leq |w| \text{ and } \epsilon \in \mathbf{dom}(t) \text{ since } init^{DictA} \rangle \\
& doDelword(w, 0)
\end{aligned}
$$

We gather the code for the body of *delword*

```
if  $w = \epsilon$
then
    $t := (t : w \mapsto 0)$
else
    $doDelword(w, 0)$
fi
```

3.6 Task 4

Now we would translate our data refinement to C functions to match the prototypes given in dict.h.

There are some significant differences between the structure of a trie used in c functions and in our data refinement (because the tries in c functions can be operated as nodes but they are operated as sets in our data refinement). The created new word/prefix attaches to the current trie by being linked to a pointer in its sub-trie, while in our data refinement, t absorbs the new word/prefix w by combining a new function subset containing w in its domain. Also, the trie in c functions uses sub-tries to recursively find the next extend prefix/word, while in our data refinement, we uses the extension of a substring of w ($POPWORD$) to represent the next level of sub-trie.

However, it should be noted that the logic used on both sides is the same (they both use recursive call to derive the final result). So we can say that our data refinement is a well representation of our c functions.

```
#include "dict.h"
#include <stdio.h>
#include <stdlib.h>
void newdict(Dict *dp){
    // malloc the space of root node.
    *dp = malloc(sizeof(struct __tnode__));
}

void doAddword(const Dict r, const word w, int index) {
    if (w[index]!= '\0') {
        // the word is not ended
        if (r->cvec[w[index+1]-'a']!=NULL) {
            /* POPWORD(w,index+1) \in dom(t) */
            // skip
        }
        else{
            // t:= (POPWORD(w,index+1) -> 0)
            newdict(&(r->cvec[w[index+1]-'a']));
        }
        // recursive call
        doAddword(r->cvec[w[index+1]-'a'],w , index +1);
    }
    else{
        // index = |w|
        // t:= t_0 \{w -> 0} U {w -> 1}
        r->eow = TRUE;
    }
}

bool doCheckword(const Dict r, const word w, int index){
    if (w[index]!= '\0') {
        // the word is not ended
        if (r->cvec[w[index+1]-'a']!=NULL) {
            /* POPWORD(w,index+1) \in dom(t) */
            // skip
        }
        else{
            // t:= (POPWORD(w,index+1) -> 0)
            // there is not exist the word in this dict
            return FALSE;
        }
        // recursive call
        return doCheckword(r->cvec[w[index+1]-'a'],w , index +1);
    }
    else{
        // index = |w|
        // return b:= (t(w) = 1)
        return r->eow;
    }
}

void doDelword(const Dict r, const word w, int index) {
    if (w[index]!= '\0') {
        // the word is not ended
        if (r->cvec[w[index+1]-'a']!=NULL) {
            /* POPWORD(w,index+1) \in dom(t) */
            // recursive call
            doDelword(r->cvec[w[index+1]-'a'],w, index+1);
        }
        else{
            // t:= (POPWORD(w,index+1) -> 0)
            // there is not exist the word in this dict
            // nothing to delete
            // skip;
            return;
        }
    }
    else{
        // index = |w|
        // return t:= t: w-> 0
        r->eow = FALSE;
    }
}

void addword (const Dict r, const word w){
    doAddword(r, w,0);
}
bool checkword (const Dict r, const word w){
    return doCheckword(r, w,0);
}
void delword (const Dict r, const word w){
    doDelword(r,w, 0);
}
void barf(char *s){
    fprintf(stderr, "%s\n",s);
    exit(1);
}
```