COMP2111

ASSIGNMENT 3

Trie Harder

May 30, 2018

Charlie Bradford z5114682

Julian Yu z5135085

Contents

1 Syntactic Data Type: Dict

We will define a dict to be a set, W, of all the words in the dict.

We take the syntactic data type Dict to be defined by the predicate

$$\mathrm{init}^{\mathrm{D}} \ = \ (W \ = \ \langle\rangle)$$

and the following operations.

1.1 Add Word$^{\mathbf{D}}$

proc *addword^D* (**value** x: word)
$\quad x \ : \ [\mathbf{TRUE},\ W = \langle x,\ W_0\rangle]$

1.2 Check Word$^{\mathbf{D}}$

func *checkword^D* (**value** x: word): bool
$\quad \mathbf{var}\ y \ \bullet\ x,\ y \ : \ [\mathbf{TRUE},\ y \ = \ (x \ \in \ W)];\mathbf{return}\ y$

1.3 Delete Word$^{\mathbf{D}}$

proc *delword^D* (**value** x: word)
$\quad x \ : \ [x \ \in \ W,\ x \ \notin \ W]$

2 Refinement to DictA

Before refinement several things must be stated. Our alphabet is L and consists of the 26 letters of the Roman alphabet. All our words w are in the set $L*$. We say a word v is less than another word w if v is a prefix of w $(\exists v'.vv' = w)$.

We define a Trie t to be a mapping between words $(w \in L^*)$ and boolean values $(t : L^* \mapsto \mathbb{B})$. t has domain $Dom\ t \subseteq L^*$.There are several requirements of t:

Minimum Trie The smallest Trie domain possible is the empty word ϵ

- See section 2.1.1

Prefix Condition If a word is in the domain of the Trie, all prefixs of the word are as well

- $w \in Dom\ t \Rightarrow \forall\ v \in L^*.(\exists v'.(vv' = w) \Rightarrow (v \in Dom\ t))$

We also define a relation f between our abstract and concrete states. $f(t) = \{w \mid w \in Dom\ t.(t(w) = \mathbf{TRUE})\}$.

We define our refinement predicate r to equal $(a = f(t))$

2.1 New Specifications

2.1.1 Initialisation Predicate

$$init^D = (t = \{\epsilon \mapsto \mathbf{FALSE}\})$$

2.1.2 Add Word

proc *addword*^{D} (**value** x: word)
$x,\ t : [\mathbf{TRUE},\ x \in Dom\ t \wedge t(x)]$

2.1.3 Check Word

func *checkword*^{D} (**value** x: word): bool
var y • $x,\ y : [\mathbf{TRUE},\ y = (x \in Dom\ t_0 \wedge t_0(x))]$; **return** y

2.1.4 Delete Word

proc *delword*^{D} (**value** x: word)
$x : [x \in Dom\ t \wedge t(x),\ \neg t(x)]$

2.2 Proof Burdens

2.2.1 Initialisation

$$init^C \Rightarrow \exists a.(init^A \wedge r(a,\ c)$$

$$\begin{aligned}
& init^C \\
\Leftrightarrow\ & \langle \text{ definition of } init^C \rangle \\
& t = \{\epsilon \mapsto \mathbf{FALSE}\} \\
\Rightarrow\ & \langle \text{ logic: create a set with existing information } \rangle \\
& \langle\rangle = \{x \mid x \in Dom\ t.(t(x) = \mathbf{TRUE}\} \\
\Rightarrow\ & \langle \text{ logic } \rangle \\
& \exists W.(W = \langle\rangle \wedge W = \{x \mid x \in Dom\ t.(t(x) = \mathbf{TRUE})\}) \\
\Leftrightarrow\ & \langle \text{ definition of } init^A \text{ and } r \rangle \\
& \exists\ a.(init^A \wedge r(a,\ c))
\end{aligned}$$

2.2.2 Add Word

$$pre^A \wedge r(a,\ c) \Rightarrow pre^C$$

$$\begin{aligned}
& \langle \text{ definition of } pre^A \rangle \\
& \textbf{TRUE} \wedge r \\
\Rightarrow\ & \langle \text{ definition of } pre^C \rangle \\
& \textbf{TRUE}
\end{aligned}$$

This is trivial for any value of r.

$$pre^A[{}^{a_0,\ x_0}/_{a,\ x}] \wedge r[{}^{a_0,\ x_0,\ c_0}/_{a,\ x,\ c}] \wedge post^C \Rightarrow \exists a.post^A \wedge r(a,c)$$

$$\begin{aligned}
& pre^A[{}^{a_0,\ x_0}/_{a,\ x}] \wedge r[{}^{a_0,\ x_0,\ c_0}/_{a,\ x,\ c}] \wedge post^C \\
\Leftrightarrow\ & \langle \text{ Defintion of } pre^A \text{ and } post^C \rangle \\
& \textbf{TRUE}[{}^{W_0}/_{W}] \wedge r(t,\ W)[{}^{t_0,\ W_0}/_{t,\ W}] \wedge \{x \in Dom\ t \wedge t(x)\} \\
\Leftrightarrow\ & \langle \text{ logic: drop true conjunct and definition of } r \rangle \\
& \{W = \langle\ w \mid t(w)\rangle\}[{}^{W_0,\ t_0}/_{W,\ t}] \wedge x \in Dom\ t \wedge t(x) \\
\Leftrightarrow\ & \langle \text{ perform subsitutions } \rangle \\
& W_0 = \langle\ w \mid t_0(w)\rangle \wedge x \in Dom\ t \wedge t(x) \\
\Rightarrow\ & \langle \text{ logic: extend quatification with } t(x) = \textbf{TRUE} \rangle \\
& W_0 = \langle\ w \mid t_0(w)\rangle \wedge W = \langle W_0, x\rangle \wedge x \in Dom\ t \wedge t(x) \\
\Rightarrow\ & \langle \text{ logic: last two conjucts equivalent to last conjuct in new statement } \rangle \\
& \exists a.(x \in a \wedge a = \langle\ w \mid t(w)\rangle) \\
\Rightarrow\ & \langle \text{ Definition of } r \text{ and } post^A \rangle \\
& \exists a.(post^A \wedge r(a,c))
\end{aligned}$$

2.2.3 Check Word

$$pre^A \wedge r(a,\ c) \Rightarrow pre^C$$

$$\begin{aligned}
& \langle \text{ definition of } pre^A \rangle \\
& \textbf{TRUE} \wedge r \\
\Rightarrow\ & \langle \text{ definition of } pre^C \rangle \\
& \textbf{TRUE}
\end{aligned}$$

This is trivial for any value of r.

$$pre^A[{}^{a_0,\ x_0}/_{a,\ x}] \wedge r[{}^{a_0,\ x_0,\ c_0}/_{a,\ x,\ c}] \wedge post^C \Rightarrow \exists a.(post^A \wedge r(a,\ c))$$

$$\begin{aligned}
& pre^A[{}^{a_0,\ x_0}/_{a,\ x}] \wedge r[{}^{a_0,\ x_0,\ c_0}/_{a,\ x,\ c}] \wedge post^C \\
\Leftrightarrow\ & \langle \text{ definition of } pre^A \text{ and } post^C \rangle \\
& \textbf{TRUE}[{}^{W_0}/_{W}] \wedge r[{}^{W_0,\ t_0}/_{W,\ t}] \wedge \{y = (x \in Dom\ t_0 \wedge t_0(x))\} \\
\Leftrightarrow\ & \langle \text{ logic: drop true conjunct and definition of } r \rangle \\
& \{W = \langle\ x \mid t(x)\rangle\}[{}^{W_0,\ t_0}/_{W,\ t}] \wedge \{y = (x \in Dom\ t_0 \wedge t_0(x))\} \\
\Leftrightarrow\ & \langle \text{ perform subsitutions } \rangle \\
& W_0 = \langle\ x \mid t_0(x)\rangle \wedge y = (x \in Dom\ t_0 \wedge t_0(x)) \\
\Rightarrow\ & \langle \text{ logic: extend quatification with } t(x) = \textbf{TRUE} \rangle \\
& W_0 = \langle\ w \mid t_0(w)\rangle \wedge W = \langle W_0, x\rangle \wedge y = (x \in Dom\ t_0 \wedge t_0(x)) \\
\Rightarrow\ & \langle \text{ logic } \rangle \\
& \exists\ a.(y = (x \in a) \wedge a = \langle\ w \mid t(w)\rangle) \\
\Leftrightarrow\ & \langle \text{ definition of } r \text{ and } post^A \rangle \\
& \exists\ a.(post^A \wedge r(a,\ c))
\end{aligned}$$

2.2.4 Delete Word

$$pre^A \wedge r(a,\ c) \Rightarrow pre^C$$

$$\begin{aligned}
& pre^A \wedge r(a,\ c) \\
\Leftrightarrow\ & \langle \text{ definition of } pre^A \text{ and } r(a,\ c) \ \rangle \\
& x \in W \wedge W = \langle\, w \mid t(w) \,\rangle \\
\Rightarrow\ & \langle \text{ logic } \rangle \\
& t(x) = \textbf{TRUE} \\
\Rightarrow\ & \langle \text{ logic } \rangle \\
& x \in Dom\ t \wedge t(x) \\
\Rightarrow\ & \langle \text{ definition of } pre^C \ \rangle \\
& pre^C
\end{aligned}$$

$$pre^A[a_0,\ x_0,\ W_0/a,\ x, W] \wedge r[a_0,\ x_0,\ W_0/a,\ x] \wedge post^C \Rightarrow \exists a. post^A \wedge r$$

$$\begin{aligned}
& pre^A[a_0,\ x_0,\ W_0/a,\ x, W] \wedge r[a_0,\ x_0,\ W_0/a,\ x] \wedge post^C \\
\Leftrightarrow\ & \langle \text{ definition of } pre^A,\ r, \text{ and } post^C \ \rangle \\
& \{x \in W\}[W_0,\ x_0/W,\ x] \wedge r[t_0,\ W_0/t,\ W] \wedge \{\neg t(x)\} \\
\Leftrightarrow\ & \langle \text{ definition of r and perform substitutions } \rangle \\
& x_0 \in W_0 \wedge W_0 = \{w \mid t_0(w)\} \wedge \neg t(x) \\
\Rightarrow\ & \langle \text{ logic } \rangle \\
& x_0 \in w_0 \wedge w_0 = \{w \mid t_0(w)\} \wedge x \notin \{w \mid t(w)\} \\
\Rightarrow\ & \langle \text{ logic: for every concrete state there is an abstract state (see 2.2.5) } \rangle \\
& x_0 \in w_0 \wedge w_0 = \{w \mid t_0(w)\} \wedge \exists W.(x \notin W \wedge W = \{w \mid t(w)\}) \\
\Rightarrow\ & \langle \text{ logic :}(\phi \Rightarrow \psi) \Leftrightarrow (\phi \wedge \chi \Rightarrow \psi) \ \rangle \\
& \exists W.(x \notin W \wedge W = \{w \mid t(w)\}) \\
\Leftrightarrow\ & \langle \text{ definition of } post^A \text{ and } r \ \rangle
\end{aligned}$$

2.2.5 Falsifiable Precondition

$$\forall\ c.(\exists a.\ r)$$

$$\begin{aligned}
& \langle \text{ definition of } c,\ a,\ \text{and, } r \ \rangle \\
& \forall\ t.(\exists\ W.\ W = \{w \mid t(w)\}
\end{aligned}$$

This is trivial as the values in the domain of t that map to true is just some subset of $L*$, and we can pick any W that is the same subset.

3 Derivation to Toy Language

3.1 New Dict

proc *newdict* (**value** t)

$t:$ [**TRUE**, $t = \{\epsilon \mapsto$ **FALSE**$\}$] (1)

(1) $\sqsubseteq$ $\langle$ ass $\rangle$

$t := \{\epsilon \mapsto$ **FALSE**$\}$;

proc *newdict* (**value** t)

$t := \{\epsilon \mapsto$ **FALSE**$\}$;

3.2 Add Word

proc *addword* (**value** x, **value** y, t)

This starts with the non-trivial precondition $x \in Dom\ t$, which could be false for all initial values of x other than ϵ, however, if we require that $x = \epsilon$ whenever the proc is called externally, then then precondition must always be true. Thus, as the operation is only called in the state space where $\epsilon \in Dom\ t$ and is only called with $y = \epsilon$, the precondition is non-falsifiable and equivalent to the true precondition given earlier. For the mathemematically rigourous conditions of the state space see the introduction to section 2.

$x, y, t: [x \in Dom\ t,\ x.y \in Dom\ t \wedge t(x.y)]$ (1)

(1) $\sqsubseteq$ $\langle$ if $\rangle$

if $y =$ '\0' **then**

$x, y, t: [x \in Dom\ t\ x.y \in Dom\ t \wedge t(x.y)]$ (2)

else

$x, y, t: [x \in Dom\ t,\ x.y \in Dom\ t \wedge t(x.y)]$ (3)

fi;

(2) $\sqsubseteq$ $\langle$ func-ass $\rangle$

$t := (t: x \mapsto$ **TRUE**$)$;

(3) $\sqsubseteq$ $\langle$ seq $\rangle$

$x, y, t: [x \in Dom\ t,\ x.y[0] \in Dom\ t]$ (4)

$x, y, t: [x.y[0] \in Dom\ t,\ x.y \in Dom\ t \wedge t(x.y)]$(5)

(4) $\sqsubseteq$ $\langle$ if $\rangle$

if $x.y[0] \notin Dom\ t$ **then**

$x, y, t: [x.y[0] \notin Dom\ t, x.y[0] \in Dom\ t]$ (6)

else

$x, y, t: [x.y[0] \in Dom\ t, x.y[0] \in Dom\ t]$ (7)

fi;

(5) $\sqsubseteq$ $\langle$ procedure call $\rangle$

$addword(x.y[0], y[1..], t)$;

(6) $\sqsubseteq$ $\langle$ ass $\rangle$

$Dom\ t := Dom\ t \cup \{x.y[0]\}$;

(7) $\sqsubseteq$ $\langle$ skip $\rangle$

skip;

proc *addword* (**value** x, **value** y, t)
  if $y = \text{`\textbackslash 0'}$ **then**
    $t := (t : x \mapsto \textbf{TRUE});$
  else
    if $x.y[0] \notin Dom\ t$ **then**
      $Dom\ t := Dom\ t \cup \{x.y[0]\};$
    else
      skip;
    fi;
    $addword(x.y[0],\ y[1...],\ t);$
  fi;

3.3 Check Word

func *checkword* (**value** *prefix*, **value** x, t, **return** y)

$$\textbf{var } y \bullet prefix, x, y, t : [\textbf{TRUE}, y = (x \in Dom\ t \wedge t(x))]; \textbf{return } y \quad (1)$$

(1) $\sqsubseteq$ ⟨ if ⟩

if $x = \text{`\textbackslash 0'}$ **then**

$$prefix, x, y, t : [prefix.x \in Dom\ t,\ y = (prefix.x \in Dom\ t \wedge t(prefix.x))] \quad (2)$$

else

$$prefix, x, y, t : [prefix \in Dom\ t,\ y = (prefix.x \in Dom\ t \wedge t(x))] \quad (3)$$

fi;

(2) $\sqsubseteq$ ⟨ logic and ass ⟩

$y := t(x);$

(3) $\sqsubseteq$ ⟨ if ⟩

if$prefix.x[0] \in Dom\ t$ **then**

$$prefix, x, y, t : [prefix.x[0] \in Dom\ t,\ y = (prefix.x \in Dom\ t \wedge t(prefix.x))] \quad (4)$$

else

$$prefix, x, y, t : [prefix.x[0] \notin Dom\ t,\ y = (prefix.x \in Dom\ t \wedge t(prefix.x))] \quad (5)$$

fi;

(4) $\sqsubseteq$ ⟨ function call ⟩

$y := checkword\ (prefix.x[0],\ x[1..],\ t,\ y)$

(5) $\sqsubseteq$ ⟨ logic (**prefix condition**) ⟩

$y := \textbf{FALSE}$

func *checkword*(**value** *prefix*, **value** x, t, **return** y)
  if $y[0] = \text{`\textbackslash 0'}$ **then**
    $b := t(x);$
  else
    if $x.y[0] \in Dom\ t$**then**
      $b := checkword(x.y[0],\ w[1..],\ t);$
    else
      $b := \textbf{FALSE};$
    fi;
  fi;
  return b;

3.4 Delete Word

proc *delword* (**value** x, **value** y, t)

$x,\ y,\ t:\ [xy\ \in\ Dom\ t\ \wedge\ t(x),\ \neg t(x)]$ (1)

(1) $\sqsubseteq$ ⟨ if ⟩

if $y\ =$ '\0' **then**

$x,\ y,\ t:\ [xy\ \in\ Dom\ t\ \wedge\ t(xy),\ \neg t(x)];$ (2)

else

$x,\ y,\ t:\ [x\ \in\ Dom\ t\ \wedge\ t(xy),\ \neg t(x)];$ (3)

fi;

(2) $\sqsubseteq$ ⟨ ass ⟩

$t\ :=\ (t:\ x \mapsto\ \textbf{FALSE})$

(3) $\sqsubseteq$ ⟨ procedure call ⟩

$delword(x.y[0],\ y[1..],\ t);$

```
proc delword (value x, value y, t)
    if y = '\0' then
        t := (t : x ↦ FALSE);
    else
        delword(x.y[0], y[1..], t);
    fi;
```

4 Translation to C

4.1 Code

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


void addword (const Dict r, const word w)
{
    if (w[0] == 0)
        r->eow = TRUE;
    else
    {
        if (r->cvec[w[0] - 'a'] == NULL)
            newdict(&(r->cvec[w[0] - 'a']);
        else
            \\ SKIP
        addword(r->cvec[w[0] - 'a'], w + sizeof(char));
    }
}


void checkword (const Dict r, const word w)
{
    bool b;
    if (w[0] == 0)
        b = r->eow;
    else
    {
        if (r->cvec[w[0] - 'a'] == NULL)
            b = FALSE;
        else
            b = checkword(r->cvec[w[0] - 'a'], w + sizeof(char));
    }
    return b;
}


void delword (const Dict r, const word w)
{
    if (w[0] == 0)
        r->eow = FALSE;
    else
        delword(r[w[0] - 'a'], w + sizeof(char));
}
```

4.2 Changes

- In the toy language we could access $t(w)$ for any w at any point, however in C we can only check one letter at a time, and the rest of the word is made up of context (nodes already traversed)
- In the toy language we simply pass t to the recursive function, whereas in C we must pass the appropriate trie node
- In the toy language it is enough to set $t := (\epsilon \mapsto \textbf{FALSE})$ however in C we need to allocate memory and set the values of all the sub-nodes to NULL
- When we add a word or prefix to *Dom* t in the toy language we just take the union of t with the mapping, however in c we must add a new node to the Trie and treat it like a fresh Trie.