Assignment 3:

$\mathbb{E}$mmet $\mathbb{M}$urray z5059840, $\mathbb{D}$anni $\mathbb{O}$vens z5059491

February 3, 2018

1 BFS Implementation

1.1 Defining an abstract queue

We will begin by defining some abstract queue operations in our toy language:

QUEUE:

$$\mathcal{Q} :: (N : \mathbb{N},\ s : V_t^*)$$

Where N is the max size of the queue, n is the current size, and s is a sequence of queue values.
We introduce the notation of $\llcorner s \lrcorner$, meaning to treat the sequence s as a set such that all the values of s are in the $\llcorner s \lrcorner$. For convenience, we define $|s|$ as equivalent to $|\llcorner s \lrcorner|$.
We can now define our 5 core abstract queue operations:

Initq:
$q : [True,\ q = (N, s) \wedge |s| = 0] \sqsubseteq initq(q)$
Enq:
$q : [|s| < N,\ q = (N, s) \wedge s = xs_0 \wedge |s| \neq 0] \sqsubseteq enq(q, x)$
Deq:
$q : [|sx| > 0 \wedge q = (N, sx), q = (N, s) \wedge s = s_0] \sqsubseteq deq(q)$
WhosNext:
$x : [q = (N, sy), x = y] \sqsubseteq x := whosNext(q)$
isEmpty:
$b : [q = (N, s), b \iff |s| = 0] \sqsubseteq b := isEmpty(q)$

1.2 Refinement

We begin by refining the provided specification of SEARCH:

proc SEARCH(**value** t, **value** N, **value** k, **result** v, **result** f) $\cdot$

$$t, N, k, v, f : \left[\begin{array}{l} \textsc{Tree}(t) \wedge \max_{i \in \mathbb{N}} |\Gamma_t^i(r_t) \cup \Gamma_t^{i+1}(r_t)| \leq N, \\ (f \wedge \exists w \in V_{t_0}(\kappa_{t_0}(w) = k_0 \wedge \lambda_{t_0}(w) = v)) \vee \\ (\neg f \wedge \forall w \in V_{t_0}(\kappa_{t_0}(w) \neq k_0)) \end{array}\right]$$

$\sqsubseteq$ $\langle$ **c-frame** $\rangle$

$$\llcorner v, f : \left[\begin{array}{l} \textsc{Tree}(t) \wedge \max_{i \in \mathbb{N}} |\Gamma_t^i(r_t) \cup \Gamma_t^{i+1}(r_t)| \leq N, \\ (f \wedge \exists w \in V_t(\kappa_t(w) = k \wedge \lambda_t(w) = v)) \vee \\ (\neg f \wedge \forall w \in V_t(\kappa_t(w) \neq k)) \end{array}\right] \lrcorner (1)$$

In order to use a breadth first search along the tree, we need a queue. Thus we must create a queue variable and initialise it to an empty queue:

(1) $\sqsubseteq$ $\langle$ **i-loc** $\rangle$ *q doesn't occur yet*

var q : $\mathcal{Q} \cdot v, f, q\,[\, pre(1),\ post(1)\,]$

$\sqsubseteq$ $\langle$ **seq** $\rangle$ *so we can refine to initq*

$v, f, q : [\, pre(1),\ pre(1) \wedge q = (N, s) \wedge |s| = 0];$

$v, f, q : [pre(1) \wedge q = (N, s) \wedge |s| = 0,\ post(1)]$

$\sqsubseteq$ $\langle$ **initq** $\rangle$

$initq(q);$

$\llcorner v, f, q : [pre(1) \wedge q = (N, s) \wedge |s| = 0,\ post(1)] \lrcorner (2)$

Similarly, we push our initial element r_t onto the queue:

(2) $\sqsubseteq$ $\langle$ **seq, con** $\rangle$

con m $\cdot$

$$v, f, q : \left[\begin{array}{l} pre(1) \wedge q = (N, s) \wedge |s| = 0 \wedge m = s, \\ pre(1) \wedge q = (N, s) \wedge s = xm \wedge |s| \neq 0 \wedge x = r_t \end{array}\right];$$

$$v, f, q : \left[\begin{array}{l} pre(1) \wedge q = (N, s) \wedge s = xm \wedge |s| \neq 0 \wedge x = r_t, \\ post(1) \end{array}\right]$$

$\sqsubseteq$ $\langle$ **enq** $\rangle$

$enq(q, r_t);$

$$\llcorner v, f, q : \left[\begin{array}{l} pre(1) \wedge q = (N, s) \wedge s = xm \wedge |s| \neq 0 \wedge x = r_t, \\ post(1) \end{array}\right] \lrcorner (3)$$

Now we set our result flag f to false so we can begin our traversal:

(3) $\sqsubseteq$ $\langle$ **seq, ass** $\rangle$ *note* $\neg f$ *is equivalent to* $f = False$

$f := False;$

$\llcorner v, f, q : \left[\begin{array}{l} pre(1) \land q = (N, s) \land s = xm \land |s| \neq 0 \land x = r_t \land \neg f, \\ post(1) \end{array}\right] \lrcorner^{(4)}$

Now we conquer the more difficult task of refining our loop. We begin with a sequential composition:

(4) $\sqsubseteq$ ⟨ **w-pre** ⟩ *Remove auxiliary statements from our precondition with a w-pre*

$\llcorner v, f, q : \left[\begin{array}{l} pre(1) \land q = (N, s) \land |s| \neq 0 \land \neg f, \\ post(1) \end{array}\right] \lrcorner^{(5)}$

And now we refine (5) into our main loop:

(5) $\sqsubseteq$ ⟨ **while, isEmpty** ⟩

while $\neg(f \lor isEmpty(q))$ **do**

$\quad v, f, q : \llcorner \left[\begin{array}{l} pre(1) \land q = (N, s) \land \neg f \land |s| \neq 0, \\ pre(1) \land q = (N, s) \end{array}\right] \lrcorner^{(6)}$

end

We derived our invariant based on the following properties of the queue:

- There exists a number i, such that the queue will be comprised of elements from the i'th and $(i+1)$'th layer of the tree (in other words, $\Gamma_t^i(r_t)$ and $\Gamma_t^{i+1}(r_t)$).
- For any element in i'th layer of the tree, if any of their successors are in the queue, then that element is not in the queue, and it is not the element we are searching for.
- Or, we have found an element w such that it's key matches the element we are searching for.

Where our loop invariant is:

$$Inv : \left(\begin{array}{l} \exists i \in \mathbb{N}. \Big(\forall p \in \llcorner s \lrcorner . (p \in \Gamma_t^i(r_t) \cup \Gamma_t^{i+1}(r_t)) \\ \land\ \forall z \in \Gamma_t^i(r_t). p \in \Gamma(z) \implies (z \notin \llcorner s \lrcorner \land \kappa_t(z) \notin k) \Big) \\ \lor\ (\exists w \in V_t.\ w \in \Gamma_t^*(r_t) \land \kappa_t(w) = k \land \lambda_t(w) = v) \end{array} \right)$$

There are a few cases of the state of the queue that were unnecessary to consider due to the definition of a tree. We are able to represent the queue as a set directly as the elements in the queue at any given time will be unique, as no node can be added multiple times. This is due to the acyclic nature of the tree structure.

Clearly, the only element in the queue is the root node. Thus, there is an i, 0, where all of the elements of the set $\llcorner s \lrcorner$ are contained in $\Gamma_t^0(r_t) \cup \Gamma_t^1(r_t)$. As per the definition of the successor function and the identity properties of the zero exponent, it is trivial that $\Gamma_t^0(r_t) = r_t$.
This satisfies the first conjunct of our invariant.
Secondly, since no successors of the root node are in the queue, the second conjunct

holds as $False \implies True$.

(6) $\sqsubseteq$ $\langle$ **i-loc, seq x 2, con, c-frame** $\rangle$ *We sequentially composed our statement so that we can perform the operations deq() and whosNext(). Note here we've rewritten our queue's sequence as ys, where $y = s_1 \wedge s = s_2 \ldots$. For this implementation we have defined s_1 as the first element in the sequence s, and so forth. This can be achieved as $|s| \neq 0$. We also remove q from the frame so that we can refine to whosNext()*

$\mathbf{con}\ y \cdot \mathbf{var}\ e : V_t \cdot$
$v, f, e : [pre(1) \wedge q = (N, sy), pre(1) \wedge q = (N, sy) \wedge e = y]$
$v, f, q, e : [pre(1) \wedge q = (N, sy) \wedge e = y, pre(1) \wedge q = (N, s) \wedge e = y]$
$v, f, q, e : [pre(6) \wedge e = y, post(6)]$

Now we make the first step of getting a node to search over via DEQ:

(6) $\sqsubseteq$ $\langle$ **deq, whosNext** $\rangle$ *As we have $n \neq 0$ in the precondition of (6) we can get the first element with whosNext() and then remove it from the queue with deq().*

$\mathbf{var}\ e : V_t \cdot e := whosNext(q);$
$deq(q);$
$\llcorner v, f, q, e : [pre(6) \wedge e = y, post(6)] \lrcorner_{(7)}$

We now need a conditional statement to check if the current node t is our goal.

(7) $\sqsubseteq$ $\langle$ **if** $\rangle$
if $\kappa_t(e) = k$ **then**
| $\llcorner v, f, q, e : [pre(6) \wedge e = y \wedge \kappa_t(e) = k, post(6)] \lrcorner_{(8)}$
else
| $\llcorner v, f, q, e : [pre(6) \wedge e = y \wedge \kappa_t(e) \neq k, post(6)] \lrcorner_{(9)}$
end

(8) refines into our goal state, where we set the flag to true and assign the payload.

(8) $\sqsubseteq$ $\langle$ **seq** $\rangle$

$$v, f, q, e : \left[\begin{array}{l} pre(6) \wedge e = y \wedge \kappa_t(e) = k, \\ pre(6) \wedge e = y \wedge \kappa_t(e) = k \wedge v = \lambda_t(t) \end{array} \right];$$

$v, f, q, e : [\ pre(6) \wedge e = y \wedge \kappa_t(e) = k \wedge v = \lambda_t(e), post(6)]$

$\sqsubseteq$ $\langle$ **ass** $\rangle$
v := $\lambda_t(e)$;
$v, f, q, e : [pre(6) \wedge e = y \wedge \kappa_t(e) = k \wedge v = \lambda_t(e), post(6)]$
$\sqsubseteq$ $\langle$ **s-post, ass** $\rangle$
v := $\lambda_t(e)$;
f := *True*

Now returning to (9), we need to enqueue all of the successors of the current node. Using the successor function Γ to retrieve a set of all successors:

(9) $\sqsubseteq$ $\langle$ **i-loc, seq, ass** $\rangle$ *Create a variable to store the set of successors.*
var succ · succ := $\Gamma_t(e)$;
$\llcorner v, f, q, e, succ : [pre(6) \wedge e = y \wedge \kappa_t(e) \neq k \wedge succ = \Gamma_t(e), post(6)] \lrcorner_{(10)}$

Now looping through the set, we pick an element from the set each time and enqueue it in our queue.

(10) $\sqsubseteq$ $\langle$ **while, seq x 2, ass x 2, enq** $\rangle$

```
while succ ≠ ∅ do
    var l :∈ succ
    enq(q, l)
    succ := succ \ {l}
end
```

Now, collecting our code we come to our BFS implementation using an abstract queue:

```
var q : Q;
init(q);
enq(q,r_t);
f := False;
while ¬(f ∨ isEmpty(q)) do
    var e : V_t;
    e := whosNext(q);
    deq(q);
    if κ_t(e) = k then
        v := λ_t(e);
        f := True
    else
        var succ · succ := Γ_t(e);
        while succ ≠ ∅ do
            var l :∈ succ;
            enq(q, l);
            succ := succ \{l}
        end
    end
end
```

Note here our abstract queue operations are highlighted in Wild Strawberry.

2 Refining our abstract implementation of a queue

2.1 Step 1

We begin by introducing new variables to describe our concrete implementation. We require:

- An array $A : V_t^{N+1}$ to store our queue elements.
- An enqueue counter $n : \mathbb{N}$ to mark where we push elements to the queue.
- A dequeue counter $m : \mathbb{N}$ to mark where we pop elements from the queue.

Note that both n and m are bounded by the conditions $0 \leq n < N+1$ and $0 \leq m < N+1$ respectively.

2.2 Step 2

Now we define our coupling invariant to relate our concrete and abstract implementations:

$$\mathcal{C} : \mathcal{Q} \times V_t^* \times \mathbb{N} \times \mathbb{N} \to \mathbb{B}$$

$$\begin{aligned} \mathcal{C}((N, s), A, n, m) = & \Big(\forall i \in \Big[0..(n-m)(mod\ N+1)\Big).\ A[(n+i)(mod\ N+1)] = s_i\Big) \\ & \wedge |s| = (n-m)(modN+1) \end{aligned}$$

2.3 Step 3

Now we augment our new assignments in order to re-establish the coupling invariant $\mathcal{C}$:

var q : $\mathcal{Q}$;
init(q);   (A)
var A : V_t^{N+1}; **var** n : $\mathbb{N}$; **var** m : $\mathbb{N}$;
n := 0; m := 0;
enq(q,r_t);   (B)
A[n] := r_t;
n := n + 1(mod N + 1);
f := $False$;
while $\neg(f \vee isEmpty(q))$ **do**
  var e : V_t;
  e := whosNext(q);
  deq(q);   (C)
  m := m + 1 (mod N + 1);
  if $\kappa_t(e) = k$ **then**
    v := λ_t(e);
    f := $True$
  else
    var succ := Γ_t(e);
    while $succ \neq \varnothing$ **do**
      var l :∈ succ;
      enq(q, l);   (D)
      A[n] := l; n := n + 1(mod N + 1);
      succ := succ $\backslash\{l\}$
    end
  end
end

We now must prove that our coupling invariant holds after each operation and after our initialisation. We are obliged to prove (A-D).

(A) We must refine:
$q, A, m, n : [True, q = (N, s) \wedge |s| = 0 \wedge \mathcal{C}(q, A, m, n)]$

$\sqsubseteq$ ⟨ **seq, c-frame** ⟩
$q : [True, q = (N, s) \wedge |s| = 0]$;
$A, m, n : [q = (N, s) \wedge |s| = 0, \mathcal{C}((N, s), A, m, n)]$

$\sqsubseteq$ ⟨ **initq** ⟩
initq(q);
$A, m, n : [q = (N, s) \wedge |s| = 0, \mathcal{C}((N, s), A, m, n)]$

$\sqsubseteq$ ⟨ **s-post, ass x 2** ⟩ *Our coupling invariant allows for any arbtrary* m *such that*

$m = n$ to represent an empty queue. By using an s-post to add $m = 0$ into the post condition, we can initialise m and follow up by setting n to the same value.
initq(q);
m := 0; n := 0;

(B) We must refine the following for our initial enqueue:
$q, A, m, n : [\mathcal{C}(q, A, m, n), \mathcal{C}(q, A, m, n) \wedge q = (N, s) \wedge s = xs_0 \wedge |s| \neq 0]$

$\sqsubseteq$ $\langle$ **i-con, c-frame, seq** $\rangle$

$$\begin{array}{l} \textbf{con}\ \ \mathrm{S} \cdot q : \left[\begin{array}{l} \mathcal{C}((N, S), A, m, n) \wedge s = S, \\ s = xS \wedge |s| \neq 0 \wedge q = (N, s) \wedge \mathcal{C}((N, S), A, m, n) \end{array} \right]; \\ \quad A, m, n : \left[\begin{array}{l} \mathcal{C}((N, S), A, m, n) \wedge s = xS \wedge |s| \neq 0 \wedge q = (N, s), \\ \mathcal{C}(q, A, m, n) \end{array} \right] \end{array}$$

$\sqsubseteq$ $\langle$ **enq** $\rangle$
con S $\cdot$ enq(q,x);

$$A, m, n : \left[\begin{array}{l} \mathcal{C}((N, S), A, m, n) \wedge s = xS \wedge |s| \neq 0 \wedge q = (N, s), \\ \mathcal{C}(q, A, m, n) \end{array} \right]$$

$\sqsubseteq$ $\langle$ **a-ass, ass** $\rangle$ *To re-establish our coupling invariant, we must update out array at position n to match the corresponding element in the sequence (as described in the first conjunct) and then increment n to $n+1(mod\ N+1)$ to ensure that $|s| = (n-m)(modN+1)$ (as described in the second conjunct).*
con S $\cdot$ enq(q,x);
A[n] := x; n := n + 1 (mod N+1)

(C) We must now refine our deq() in the loop:

$q, A, m, n : [\mathcal{C}((N, sy), A, m, n) \wedge |sy| \geq 0 \wedge q = (N, sy), \mathcal{C}((N, s), A, m, n)) \wedge s = s_0];$

$\sqsubseteq$ $\langle$ **seq, c-frame, i-con** $\rangle$ *Removing everything but our dequeue counter from the frame*
con S $\cdot$ $q : [\mathcal{C}((N, sy), A, m, n) \wedge |sy| \geq 0 \wedge q = (N, sy) \wedge S = s, (\mathcal{C}((N, Sy), A, m, n)) \wedge q = (N, S)];$
$m : [\mathcal{C}((N, Sy), A, m, n) \wedge |sy| \geq 0, \mathcal{C}((N, S), A, m, n))];$

$\sqsubseteq$ $\langle$ **deq** $\rangle$
deq(q);
$m : [\mathcal{C}((N, Sy), A, m, n) \wedge |sy| \geq 0, \mathcal{C}((N, S), A, m, n))];$

$\sqsubseteq$ $\langle$ **ass** $\rangle$ *To re-establish our coupling invariant in the post condition, we must match the new sequence established in the abstract stack. To accomplish this, we increment m to $m(modN + 1)$ such that $n - m = |s|$*
deq(q);

m := m + 1 (mod N + 1);

(D) Finally, the proof of the final enque follows that of (B).

2.4 Step 4

We now replace boolean expressions involving our abstract queue with concrete ones. We only have 1 expression involving our abstract queue, isEmpty(), which we can show is equivalent to $m = n$. To show this, we bring in our coupling invariant with $|s| = 0$:

$$\mathcal{C}((N, s), A, n, m) = \Big(\forall i \in [0..0).\ A[(m + i)(mod\ N + 1)] = s_i\Big) \wedge 0 = 0$$

Clearly, $[0, 0)$ gives us no i to describe the array, i.e. we have no elements on the LHS. As our sequence is empty, there are no $n \in \mathbb{N}$ such that s_n is an element, therefore we have an empty sequence, so our coupling invariant is satisfied.
Now we may replace isEmpty() with $m = n$.

Next, we must show that our whosNext() operation maintains our coupling invariant. Note that as neither our abstract queue nor our concrete queue are in the frame when we refine to whosNext, we cannot manipulate the queue and therefore by a breathtakingly pedestrian conclusion our coupling invariant is maintained.

2.5 Step 5

Our program is now independant of our abstract queue, and we can remove auxillary queue references. We now arrive at our concrete implementation:

var A : V_t^{N+1}; **var** n : $\mathbb{N}$; **var** m : $\mathbb{N}$;
n := 0; m := 0;
A[n] := r_t;
n := n + 1(mod N + 1);
f := $False$;
while $\neg(f \vee m = n)$ **do**
    var e : V_t;
    e := A[m];
    m := m + 1 (mod N + 1);
    if $\kappa_t(e) = k$ **then**
        v := λ_t(e);
        f := $True$
    else
        var succ := Γ_t(e);
        while $succ \neq \varnothing$ **do**
            var l :$\in$ succ;
            A[n] := l; n := n + 1(mod N + 1);
            succ := succ $\setminus\{l\}$
        end
    end
end

3 C code

```c
#include <stdlib.h>
#include <stdio.h>
#include "bbq.h"
#include "bfs.h"

/* Search function, where:
 * t is our tree
 * N is our max size
 * k is our node k value we search for
 * v is a pointer to the found node with key k
 */
void search (Tree root, unsigned int N, Key key, T *val, RetVal *found) {
    // Declare our circular buffer and two counters
    Tree A[N+1]; int n; int m;
    n = 0; m = 0;
    // Set the initial value of the array
    A[n] = root;
    // Increment n and set found
    n = (n + 1) % (N + 1);
    *found = Failure;
    // While nothing is found or our queue is empty
    while (! (*found || m == n)) {
        // take our top element
        Tree e = A[m];
        // remove it from the queue
        m = (m + 1) % (N + 1);
        // Check if we've found our value
        if (cmpKey(key, e->id)) {
            *val = e->val;
            *found = Success;
        // Add successors otherwise
        } else {
            List succ = e->list;
            while (succ != NULL) {
                // Add the successor and increment list
                A[n] = succ->n; n = (n + 1) % (N + 1);
                succ = succ->next;
            }
        }
    }
}
```

bfs.c

We make a few adjustments between the C code and our implementation.

Firsly, our C code uses the provided linked list of successors as opposed to the set we defined. This means that rather than picking and removing an element from the linked list, we merely take the element and iterate over to the next element in the list. Thus, our condition that $succ \neq \varnothing$ is equated to 'succ != NULL'.

Our payload and key functions are equated to accessing the *value* and *id* fields of the tree node.

Any references to $(mod N + 1)$ are replaced with the C equivalent *% (N + 1)*.
Our r, f, k and v variables are the provided *root*, *found*, *key val* respectively.