INDIAN INSTITUTE OF TECHNOLOGY, MADRAS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNDERGRADUATE RESEARCH IN COMPUTER SCIENCE - I

Project Report

Verified Implementation of Interval Tree in F*

Mantra Trambadia - CS20B083
under Dr. KC Sivaramakrishnan

CHENNAI, APRIL 2023

Contents

1 Acknowledgements

I would like to thank Dr. KC Sivaramakrishnan for his guidance and support throughout the project. Additionally, I would like to express my appreciation to my co-guide Vimala S for her invaluable assistance. I would also like to thank Dr. Narayanswamy for his coordination of the UGRC course.

2 Abstract

This project involved the implementation of an interval tree data structure in F*, with a focus on verifying its correctness. Interval trees provide an efficient means of storing and querying intervals. F* is a functional programming language with effects that are designed for program verification. As a final step, we extracted the interval tree implementation from F* to OCaml, allowing for greater flexibility in its use. Overall, this project has contributed to a deeper understanding of the benefits of using F* for program verification, specifically data structures.

3 Introduction

3.1 F* (FStar)

F* is a programming language that blends functional programming with program verification, aimed at producing reliable software.[1] It is a dependently typed language, meaning that the types of expressions can depend on values, enabling F* to express complex invariants that can be checked by its type-checker. This, in turn, allows F* to perform powerful static analysis of programs, detecting and preventing a wide range of errors before run-time.

F* also supports the use of effects, enabling it to express computational effects such as I/O, exceptions, and non-determinism, in a principled and composable way. Additionally, F* supports both interactive and automated theorem proving, allowing developers to reason about the correctness of their programs, even in the presence of complex data structures and algorithms.[4] With its combination of the expressive type system, effect system, and theorem-proving capabilities, F* provides a powerful tool for building reliable software that is both efficient and correct.

F*'s type system includes dependent types, monadic effects, refinement types, and a weakest precondition calculus. After verification, F* programs can be extracted to efficient OCaml, F#, C, WASM, or ASM code. The main ongoing use case of F* is building a verified, drop-in replacement for the whole HTTPS stack.

```
(* Dependent types *)

type modified = (x:a{Prop}) (* a is original type *)
type even_nat = (n:nat{n \% 2 = 0}) (* even_nat is also called refinement type of nat *)

(* We can also specify specifications on the output of functions *)
let f (x:even_nat) : (y:nat{y \% 2 = 1}) =
  x + 1 (* x is even_nat, so x + 1 is odd *)
```

```
(* Function specification tells F* what conditions should output satisfy, it is heavily used in proofs *)

(* Lemmas *)

let even_sum_even (x y:even_nat) : Lemma (ensures (x+y)%2 = 0) =
  ()
```

Lemmas are used to tell F* some properties about the variables involved in the proof. You need to prove lemmas as well, the above lemma is simple enough that it does not require any proof from our side, F* can prove that on its own. We will see many lemmas in this report which require tricky proofs.

3.2 Interval Trees

Interval tree is a data structure that allows us to store intervals and perform various queries efficiently[5]. It is often used for systems where windowing queries are required such as computational geometry[3], scheduling, and database systems.

There are many implementations of interval trees in the literature, we will be looking at a specific implementation of interval tree as described in the book "Computational Geometry: Algorithms and Applications"[2].

Interval tree at each node stores a set of intervals which contain the node's key, left and right child nodes. The key is chosen to be the median of the endpoints of intervals in the sub-tree of that node to make the queries faster. The left child node stores all the intervals which have their endpoint less than the node's key. The right child node stores all the intervals which have their start point greater than the node's key. The set of intervals are stored in 2 different ways, one is sorted by the start point and the other is sorted by the endpoint.

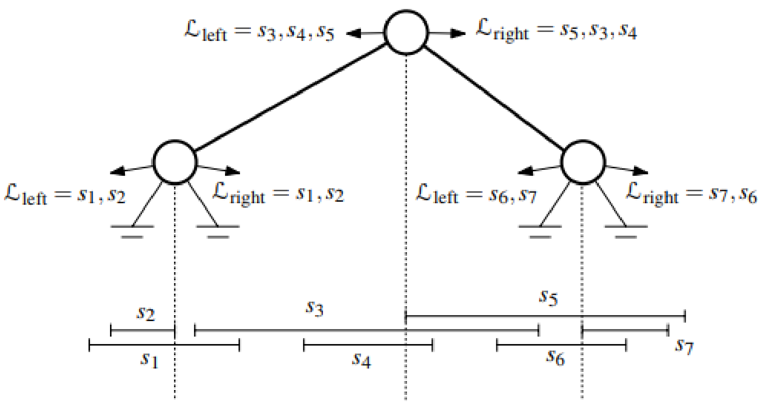

Figure 1: Interval Tree

If $I = \emptyset$, then the interval tree is a leaf node. Otherwise, let x_{mid} be the median of the bounds(both start and end) of the intervals. Let

- $I_{mid} = \{[x_j : x'_j] \in I : x_j \leq x_{mid} \leq x'_j\}$

- $I_{left} = \{[x_j : x'_j] \in I : x'_j < x_{mid}\}$
- $I_{right} = \{[x_j : x'_j] \in I : x_j > x_{mid}\}$
- $L_{left} = sorted\ by\ start\ point\ of\ I_{mid}\ in\ increasing\ order$
- $L_{right} = sorted\ by\ end\ point\ of\ I_{mid}\ in\ decreasing\ order$

Then, the node stores x_{mid}, L_{left}, L_{right}, and left and right child are made from I_{left} and I_{right}.

We will implement `create` and `query` function similar to what is implemented in the current version of OCaml code[6] and prove their correctness.

4 Verification Problem

Formal verification of data structures is a relatively new field but has become more accessible and easier with the introduction of tools such as z3[8], Dafny[7], and F*. F* has been used to verify a wide range of data structures, including some of the complex data structures, such as red-black trees[4].

The task on hand is to implement `interval_tree` data structure and verify its correctness. Verification in this context means that the data structure behaves the way it should in all scenarios. This also includes properties of functions on the data structure such as their correctness and time complexity.

We will be looking at 3 major verification problems:

1. Verify that an interval tree satisfies the conditions mentioned in the above section.

   The main challenge here will be to write `(median_bounds : list interval -> nat` function which returns the median of bounds(endpoints) of all elements in the list of intervals.

2. Verify that function `(create : list interval -> interval_tree)` creates `interval_tree` from a list of intervals.

   This task will require us to partition the list of intervals into 3 parts and then prove that those satisfy the verification conditions. And prove that `create` function terminates in all cases and provides a valid `interval_tree` no matter what list of intervals is passed as argument.

3. Verify that function `(query : interval_tree -> nat -> list interval)` returns all the intervals which contain queried value.

   For this task, we can specify that `interval_tree` is equivalent to a list of intervals when queried for any value. The returned list from `interval_tree` is the same as you would have got if you iterated through the whole list to check if a queried value is in the interval, tho `interval_tree` will be much faster in terms of time complexity.

We will use proof by induction, proof by contraposition, and other proof techniques and show how easy it is to use in F*.

5 Implementation

5.1 interval

```
module Interval

type interval_aux = {
    lbound: nat;
    rbound: nat
}

let ordered (i:interval_aux): Tot bool =
    i.rbound >= i.lbound

type interval = (i:interval_aux{ordered i})
```

Here, we have used `i:interval_aux{ordered i}`, which is a refinement type on `interval_aux`. It says that `interval` must have `rbound` greater than or equal to `lbound`.

We have defined some functions on `interval` and given their specifications according to the definitions.

```
(* Create must take lbound less than or equal to rbound *)
let create (lbound:nat) (rbound:nat{rbound>=lbound}): interval =
    { lbound ; rbound }

let is_before (x_mid:nat) (itv:interval) : Tot (b:bool{b <==> itv.rbound < x_mid}) =
    itv.rbound < x_mid

let is_after (x_mid:nat) (itv:interval) : Tot (b:bool{b <==> itv.lbound > x_mid}) =
    itv.lbound > x_mid

let contains (x_mid:nat) (itv:interval) : Tot (b:bool{b <==> (itv.rbound >= x_mid /\ itv.lbound <=
↪  x_mid)}) =
    (itv.lbound <= x_mid) && (x_mid <= itv.rbound)
```

Now we can build `interval_tree` using `interval`.

5.2 interval_tree_aux

We will first define `interval_tree_aux` which encaptures the structure of the interval tree and then define `interval_tree` using refinements on `interval_tree_aux`.

```
type interval_tree_aux =
  | Empty : interval_tree_aux
  | Node : (mid:nat) -> (l:list interval) -> (r:list interval{is_permutation interval l r}) ->
  ↪  (left:interval_tree_aux) -> (right:interval_tree_aux) -> interval_tree_aux
```

Here, we have used `r:list interval{is_permutation interval l r}`, which is a refinement type on `list interval`. This says that `r` is a permutation of `l`. This is the basic structure of an interval tree. We still have to refine it further to ensure the conditions mentioned in Interval Trees subsection.

We have to define some basic functions on `interval_tree_aux` which can be used to further specify conditions on `interval_tree_aux`.

Note that, $i \in I_{mid} \Leftrightarrow i \in I_{left} \Leftrightarrow i \in I_{right}$. So, we will only check $I_{left} \equiv l$ from now on.

```
(* Returns true if interval i is in the tree it *)
let rec in_itv_tree (i:interval) (it:interval_tree_aux) : Tot bool =
  match it with
  | Empty -> false
  | Node m l r left right -> (mem i l) || (in_itv_tree i left) || (in_itv_tree i right)

(* Returns number of times interval i is in the tree it *)
let rec count_itv_tree (i:interval) (it:interval_tree_aux) : Tot nat =
  match it with
  | Empty -> 0
  | Node m l r left right -> (count i l) + (count_itv_tree i left) + (count_itv_tree i right)
```

An interesting thing to note here is that it is not possible to give any specification for the output of the above functions, as they are quite fundamental to the structure of the interval tree.

We can also define a function to convert `interval_tree_aux` to `list interval`, output list will contain intervals in I_{mid}, I_{left} and I_{right}.

```
let rec to_itvs (it:interval_tree_aux) : Tot (l:list interval{forall i. ((count i l) = (count_itv_tree i
↪   it)) /\ ((mem i l) <==> (in_itv_tree i it))})=
    match it with
    | Empty -> []
    | Node _ l r it1 it2 ->
           let ll = to_itvs it1 in
           let rr = to_itvs it2 in
           l@(ll@rr)
```

We have defined the specification of `to_itvs` using `count_itv_tree` and `in_itv_tree`. This is because `to_itvs` is a function that converts `interval_tree_aux` to `list interval`. So, we can say that `count_itv_tree` and `in_itv_tree` give the same result as if done `count` and `mem` on `to_itvs` output.

5.3 `interval_tree`

5.3.1 Refinements

We need to define refinements for `interval_tree_aux` which will ensure that it satisfies conditions mentioned in Interval Trees subsection. These can be represented mathematically as,

1) $\forall i \in$ `(to_itvs left). (is_before mid i)`

2) $\forall i \in$ `(to_itvs right). (is_after mid i)`

3) $\forall i \in l.$ `(contains mid i)`

4) `l is sorted in increasing order of lbound`

5) `r is sorted in decreasing order of rbound`

6) $mid = median\_bounds($`(to_itvs left)@l@(to_itvs right)`$)$

7) `left is a valid interval tree`

8) `right is a valid interval tree`

5.3.2 Membership Refinements

We will define a function `check_list` which will check if a given list satisfies a given property. We will use this function to check conditions 1, 2, and 3.

```
let rec check_list (#a:eqtype) (l:list a) (f:a->bool) : Tot (b:bool{b <==> (forall e. ((mem e l) ==> f
↪  e))}) =
  match l with
    | [] -> true
    | hd::tl -> (f hd) && (check_list tl f)
```

Note that F* can prove on its own that `check_list` satisfies the specification. In some cases, we will have to provide more information to F* to prove the specification. We will see this in later sections.

5.3.3 Sorted Refinements

FStar.List already has a function `sorted f l` which checks if list `l` is sorted according to the given comparison function `f`. We can use this function to check conditions 4 and 5. We have defined the following comparison functions for `l` and `r`.

```
(* func_left sorts in increasing order based on left bound first and then rbound *)
let func_left : total_order interval =
  let f (i1 i2:interval) = if i1.lbound < i2.lbound then true else if i1.lbound > i2.lbound then false
            else i1.rbound <= i2.rbound in
  f

(* func_right sorts in decreasing order based on right bound first and then lbound *)
let func_right : total_order interval =
  let f (i1 i2:interval) = if i1.rbound > i2.rbound then true else if i1.rbound < i2.rbound then false
            else i1.lbound >= i2.lbound in
  f
```

We have defined type of `func_left` and `func_right` as `total_order interval`, this is a requirement for `quicksort` function. `total_order` is a type class that is defined as,

```
val quicksort: #a:eqtype -> f:total_order a -> l:list a ->
  Tot (m:list a{sorted f m /\ is_permutation a l m})

type total_order (a:eqtype) =
  f:(a -> a -> Tot bool) {
      (forall a. f a a) (* reflexive *)
    /\ (forall a1 a2. f a1 a2 /\ f a2 a1 ==> a1 = a2) (* anti-symmetric *)
    /\ (forall a1 a2 a3. f a1 a2 /\ f a2 a3 ==> f a1 a3) (* transitive *)
    /\ (forall a1 a2. f a1 a2 \/ f a2 a1) (* total *)
  }
```

5.3.4 Median Refinements

Now, we need to define a function `median_bounds` which will return the median of bounds of a given list of intervals. We will use this function to check condition 6. We also need to specify that there will be at least one element in the list that contains the median of bounds. We will use this property to prove the termination of `create` function.

We would ideally want a function similar to the following,

```
val median_bounds (itvs:list interval{length itvs > 0}) : (mid:nat{exists i. mem i itvs /\ (i.lbound <=
↪  mid /\ i.rbound >= mid)})
let median_bounds itvs =
  let bounds = compute_bounds itvs in
  let sorted_bounds = quicksort (fun (x y:nat) -> x <= y) bounds in
  let mid = median sorted_bounds in
  mid
```

But proving that there is an interval in `itv` which contains `mid` is not trivial. As we need to give F* a lot of information about the functions used in calculating the median.

`compute_bounds` function returns a list of bounds of all intervals in the given list. We will use this function to calculate the median of bounds of intervals in the given list. F* can prove that `compute_bounds` satisfies the specification without any additional information.

```
let rec compute_bounds (itvs:list interval) : (l:list nat{((length l) % 2 = 0) /\ (forall b. mem b l <==>
↪  (exists i. mem i itvs /\ (b = i.lbound \/ b = i.rbound))) /\ (forall i. (mem i itvs) ==> ((mem
↪  i.lbound l) /\ (mem i.rbound l)))}) =
  match itvs with
  | [] -> []
  | hd::tl -> (hd.lbound::(hd.rbound::(compute_bounds tl)))
```

`quicksort` function is predefined and will return a list that is sorted according to the given comparison function and is a permutation of the given list.

It is not clear how we will prove that there is an interval in `itv` which contains `mid` which is returned by `median`. We can rather create `median_helper` function which returns the index of the median of the given list. This is a much easier task and we can directly write the definition of the median.

```
let median_helper (l:list nat{sorted (fun (x y:nat) -> x <= y) l /\ length l > 0}) : (i:nat{((length l % 2
↪   = 0) ==> (i = (length l / 2))) /\ ((length l % 2 = 1) ==> (i = ((length l -1 ) / 2)))}) =
  let n = length l in
  if n % 2 = 1 then
    (n-1) / 2
  else
    n / 2
```

The predefined `index` function does not specify that returned element is in the list. We can choose to create a lemma that will specify that the returned element is in the list. But we will rather define our own `my_index` function which will return the element at the given index in the given list and will also specify that the returned element is in the given list.

```
let rec my_index (#a:eqtype) (l:list a) (i:nat{i < length l}) : Tot (e:a{mem e l /\ count e l > 0}) =
    match l with
    | h::t -> if i = 0 then h else my_ind t (i-1)
```

Thus our modified code to compute the median becomes,

```
val median_bounds (itvs:list interval{length itvs > 0}) : (mid:nat{exists i. mem i itvs /\ contains mid
↪   i})
let median_bounds itvs =
  let bounds = compute_bounds itvs in
  let sorted_bounds = quicksort (fun (x y:nat) -> x <= y) bounds in
  let mid_ind = median_helper sorted_bounds in
  let mid = my_index sorted_bounds mid_ind in
  mid
```

But the above code is not accepted by F* and it complains that the length of `sorted_bounds` should be greater than 0. This is because the `quicksort` function only specifies that returned list is a permutation of the given list which is not enough to prove that it will have the same length. Below is the definition of `is_permutation` function.

```
type is_permutation (a:eqtype) (l:list a) (m:list a) =
  forall x. count x l = count x m
```

We can define a lemma that will specify that if 2 lists are permutations of each other then they will have the same length.

Lemma 5.3.1 `same_length`: *Given two lists l_1, l_2,*

$$l_2 \ \texttt{is permutation of} \ l_1 \rightarrow length(l_1) = length(l_2)$$

The proof will be done by induction on the length of one of the lists. We can erase the first element of `list1` from `list2` using `(remove : list interval -> interval -> list interval)` function and then recursively call the lemma.

The specification of `remove` function can be given in terms of `count` of members as follows,

```
let rec remove (#a:eqtype) (x:a) (l:list a{mem x l})
  : Tot (l1:list a{(forall e. (x <> e) ==> (count e l = count e l1)) /\ (count x l = count x l1 + 1) /\
  ↪  (length l1 = length l - 1)}) =
  match l with
  |[] -> []
  |hd::tl -> if x = hd then tl else hd::remove x tl
```

If the length of `list1` is 0 then we will have to prove that `list2` is also empty. This can be done by using `no_mem_impl_empty` lemma.

Lemma 5.3.2 `no_mem_impl_empty`*: Given a list l,*

$$(\forall e.\ \neg mem(e, l)) \rightarrow length(l) = 0$$

This is not easy to prove on its own in F*, we can rather prove the counter-positive of this lemma, and use it to prove `no_mem_impl_empty` lemma.

Lemma 5.3.3 `non_empty_impl_mem`*: Given a list l,*

$$length(l) > 0 \rightarrow (\exists e.\ mem(e, l))$$

```
let non_empty_impl_mem (#a:eqtype) (l:list a) : Lemma (ensures ((length l > 0) ==> (exists e. (mem e l))))
↪  =
  match l with
  | hd::tl -> ()
  | [] -> ()

let no_mem_impl_empty (#a:eqtype) (l:list a) : Lemma (ensures (forall e. not (mem e l)) ==> length l = 0)
↪  =
  non_empty_impl_mem l
```

Now, we can prove `same_length` lemma as follows,

```
let rec same_length (#a:eqtype) (l:list a) (m:list a) : Lemma (requires is_permutation a l m) (ensures
↪  length l = length m) =
  match l with
  |[] -> no_mem_impl_empty m;
        ()
  |hd::tl -> same_length tl (remove hd m)
```

Finally, we have enough information to convince F* that `median_bounds` function is correct.

```
let median_bounds (itvs:list interval{length itvs > 0}) : (m:nat{exists i. mem i itvs /\ contains m i}) =
  let bounds = compute_bounds itvs in
  let sorted_bounds = quicksort (fun (x y:nat) -> x <= y) bounds in
  same_length bounds sorted_bounds;
  let mid_ind = median_helper sorted_bounds in
  let mid = my_ind sorted_bounds mid_ind in
  assert(exists i. (mem i itvs) /\ ((mid = i.lbound) \/ (mid = i.rbound)));
  mid
```

5.3.5 Complete refinement

We can use above-defined functions to encode the complete refinement of `interval_tree` type. We will define a function `ordered` which will check if the given `interval_tree` is ordered or not. We will use this function to define the refinement of `interval_tree` type.

```
let rec ordered (it:interval_tree_aux) =
  match it with
  | Empty -> true
  | Node mid l r left right -> (length l > 0) &&
                               (check_list (to_itvs left) (is_before mid)) &&
                               (check_list (to_itvs right) (is_after mid)) &&
                               (check_list l (contains mid)) &&
                               (sorted func_left l) &&
                               (sorted func_right r) &&
                               (mid = (median_bounds (to_itvs it))) &&
                               (ordered left) &&
                               (ordered right)

type interval_tree = (it:interval_tree_aux{ordered it})
```

5.4 Creation of interval tree from a list of intervals

Below is the pseudo-code for the function that will create an interval tree from a list of intervals.

```
function create(I):
  if I is empty:
    return Empty
  else:
    Let node be a new node
    Let mid be the median of the intervals in I
    Let IL be the set of intervals in I that are after mid
    Let IR be the set of intervals in I that are before mid
    Let IM be the set of intervals in I that contain mid
    node.mid = mid
    node.l = IM sorted in increasing order by left endpoint
    node.r = IM sorted in decreasing order by right endpoint
```

```
node.left = create(IL)
node.right = create(IR)
return node
```

We can get the median of a list of intervals by calling the `median_bounds` function. We can define a partition function that will partition the given list of intervals into 3 lists. The first list will contain intervals that are before `mid`, the second list will contain intervals that contain `mid`, and the third list will contain intervals that are after `mid`.

We have used `partition` function which is defined in `FStar.List.Tot` module. This function takes a list and a predicate and returns a pair of lists. The first list will contain elements of the given list for which the predicate is true and the second list will contain elements of the given list for which the predicate is false.

```
let partition_upd (itvs:list interval{length itvs > 0}) (f1:interval->bool) (f2:interval->bool) :
↪  (left:list interval{forall l. (mem l left) ==> (f1 l)}) * (mid:list interval{(forall m. (mem m mid ==>
↪  ((f1 m) = false) /\ (f2 m) = false))}) * (right:list interval{forall r. (mem r right) ==> (((f1 r) =
↪  false) /\ ((f2 r) = true))}) =
  let left_itvs, maybe_right_itvs =
    partition f1 itvs in
  let right_itvs, mid_itvs =
    partition f2 maybe_right_itvs in
  left_itvs, mid_itvs, right_itvs
```

F* is able to prove that the returned lists are indeed partitioned according to the given specification.

Finally, we can use previously defined functions `func_left` and `func_right` to sort the intervals in the desired order. We can use `quicksort` function to sort the intervals.

```
val create (itvs:list interval) : Tot (it:interval_tree{is_permutation interval itvs (to_itvs it)})
↪  (decreases %[length itvs])
let rec create itvs =
  match itvs with
  | [] -> Empty
  | itvs ->
    let mid = (median_bounds itvs) in
    let left_list, mid_list, right_list = partition_upd itvs (is_before mid) (is_after mid) in
    let l = quicksort func_left mid_list in
    let r = quicksort func_right mid_list in
    let it1 = (create left_list) in
    let it2 = (create right_list) in
    let it = Node mmid l r it1 it2 in
    it
```

But of course, the task is not that easy, we have to convince F* that the returned interval tree is indeed ordered. We will have to prove each of the 9 conditions present in the definition of `ordered` function.

5.4.1 `length l > 0`

F* knows that `mid_list` has length greater than 0 as we specified in the definition of `median_bounds` function that at least one interval contains the median. F* also knows that `l` is a permutation of `mid_list` (postcondition of `quicksort`). So, we can use `same_length` lemma to prove that `l` has length greater than 0.

5.4.2 `check_list` for `l`, `it1` and `it2`

F* knows that `to_itvs it1`, `to_itvs it2` are permutations of `left_list`, `right_list` respectively (post condition of `create`). So, we can create a lemma `perm_check_list` which will prove that if `check_list` is true for a list then it is true for the permutation of that list. This is trivial enough that we don't have to prove it.

5.4.3 `sorted` for `l` and `r`

Post condition for `quicksort` function is sufficient to prove that `l` and `r` are sorted according to `func_left` and `func_right` respectively.

5.4.4 `mid` = `median_bounds (to_itvs it)`

F* knows that `mid_list` is a permutation of `to_itvs it` (postcondition of `quicksort`). So, we can write a lemma `eq_median` which will prove that if 2 lists are permutations of each other then their median is equal.

Lemma 5.4.1 `eq_median`*: Given two lists $l1$ and $l2$,*

$$is\_permutation(l1, l2) \rightarrow median\_bounds(l1) = median\_bounds(l2)$$

This is significantly tougher than any other conditions as `median_helper` is a complex function. We will use the following steps to prove this lemma.

1. If two lists are permutations of one another then `compute_bounds` of both lists are also permutations of one another.
   (a) Link count of bounds of a list of intervals with the list using `count_bounds` function.
   (b) Prove the correctness of `count_bounds` function to show that it indeed gives `count` of any bound in `compute_bounds`.
   (c) Prove that if two lists are permutations of each other then they give the same `count_bounds` for all bounds.

   This can be described by `perm_bounds_is_perm` lemma,

   Lemma 5.4.2 `perm_bounds_is_perm`*: Given two lists of intervals $l1$ and $l2$,*

$$is\_permutation(l1, l2) \rightarrow is\_permutation(compute\_bounds(l1), compute\_bounds(l2))$$

2. If two lists are permutations of one another then `quicksort` for a common comparison function returns same sorted list for both the lists.

   (a) Prove that all elements in a sorted list are greater than equal to head.

   (b) $h2$ is element of $l1$ and $h1$ is element of $l2$, thus they must be same.

   (c) Induction on tails $t1$ and $t2$ as they are also permutation and sorted.

   This step can be described by `sorted_perm_same` lemma,

   Lemma 5.4.3 `sorted_perm_same`*: Given two sorted lists $l1$ and $l2$,*

$$is\_permutation(l1, l2) \wedge sorted(f, l1) \wedge sorted(f, l2) \rightarrow l1 = l2$$

The definition of `count_bounds` function is as follows, it returns how many times `b` is appearing in `itvs`.

```
let rec count_bounds (itvs:list interval) (b:nat) : nat =
  match itvs with
  | [] -> 0
  | hd::tl -> if (hd.lbound = hd.rbound) then
                if hd.lbound = b then 2 + count_bounds tl b
                else count_bounds tl b
            else if (hd.lbound = b) || (hd.rbound = b) then 1 + count_bounds tl b
                 else count_bounds tl b
```

Now, we need to prove correctness of `count_bounds` function and link it to `count` function with following lemma,

Lemma 5.4.4 `count_bounds_lemma`*: Given a list l and an element e,*

$$\forall b.count\_bounds(itvs, b) = count(b, compute\_bounds(itvs))$$

Since `count_bounds` function is defined in 2 major cases, namely $lbound = rbound$ and $lbound \neq rbound$, we will create 2 lemmas `cons_count_lemma_1` and `cons_count_lemma_2` which will help prove `count_bounds_lemma`.

Lemma 5.4.5 `cons_count_lemma_1`*: Given a list of intervals tl and an interval hd,*

$$hd.lbound = hd.rbound \rightarrow (count\_bounds(hd :: tl, hd.rbound) = count\_bounds(tl, hd.rbound) + 2)$$

Lemma 5.4.6 `cons_count_lemma_2`*: Given a list of intervals tl and an interval hd,*

$$hd.lbound \neq hd.rbound \rightarrow (count\_bounds(hd :: tl, hd.rbound) = count\_bounds(tl, hd.rbound) + 1) \wedge$$

$$(count\_bounds(hd :: tl, hd.lbound) = count\_bounds(tl, hd.lbound) + 1)$$

We need another trivial lemma `cons_lemma` to prove correctness of `count_bounds_lemma`,

Lemma 5.4.7 `cons_lemma`*: Given a list of intervals itv,*

$$\forall b.(b \neq e) \rightarrow (count(b, l) = count(b, e :: l)) \wedge (count(e, l) = count(e, e :: l) - 1)$$

Finally, we can prove `count_bounds_lemma` using induction on tl, `cons_lemma` to create `compute_bounds` of $itvs$ and then `cons_count_lemma_1` in case when $lbound = rbound$ and `cons_count_lemma_2` when $lbound \neq rbound$.

```
let rec count_bounds_lemma (itvs:list interval) : Lemma (ensures (forall b. (count_bounds itvs b) = count
↪  b (compute_bounds itvs))) =
  match itvs with
  | [] -> ()
  | hd::tl -> let t = compute_bounds tl in
            count_bounds_lemma tl;
            cons_lemma t hd.rbound;
            cons_lemma (hd.rbound::t) hd.lbound;
            if hd.rbound = hd.lbound then cons_count_lemma_1 hd tl
            else cons_count_lemma_2 hd tl
```

We can write the following lemma for step 3,

Lemma 5.4.8 `count_bounds_perm_lemma`*: Given two lists of intervals $l1$ and $l2$,*

$$l_2 \ \texttt{is permutation of} \ l_1 \rightarrow \forall b.(count\_bounds(l1, b) = count\_bounds(l2, b))$$

The proof will be done by induction and in cases where $l1_{hd}.rbound = l1_{hd}.lbound$ and $l1_{hd}.rbound \neq l1_{hd}.lbound$. This is done as `count_bounds` function behaves differently in both cases.

For the first case, we need to specify conditions when you remove i where $i.lbound = i.rbound$ from a list of intervals $itvs$ which contains i. We have done this with `remove_lemma_1`,

Lemma 5.4.9 `remove_lemma_1`*: Given a list of intervals itv and an element i,*

$$count\_bounds(itvs, i.lbound) = count\_bounds(remove(i, itvs), i.lbound) + 2$$

$$\forall b.b \neq i.lbound \Rightarrow (count\_bounds(itvs, b) = count\_bounds(remove(i, itvs), b))$$

This lemma can be proven simply by induction until you reach an element equal to i where we can use lemma `cons_count_lemma_1` that we have proved earlier.

```
let rec remove_lemma_1 (itvs:list interval) (i:interval) : Lemma (requires (mem i itvs) /\ i.lbound =
↪  i.rbound) (ensures (count_bounds itvs i.lbound = count_bounds (remove i itvs) i.lbound + 2) /\ (forall
↪  b. b <> i.lbound ==> (count_bounds itvs b = count_bounds (remove i itvs) b))) =
  match itvs with
  | [] -> ()
  | hd::tl -> if hd = i then cons_count_lemma_1 hd tl
            else
            remove_lemma_1 tl i
```

And similarly, for the second case, we need to specify conditions when removing i where $i.lbound \neq i.rbound$ from a list of intervals $itvs$ which contains i then, We have done this with `remove_lemma_2`,

Lemma 5.4.10 `remove_lemma_2`*: Given a list of intervals itv and an element i,*

$$count\_bounds(itvs, i.lbound) = count\_bounds(remove(i, itvs), i.lbound) + 1$$

$$count\_bounds(itvs, i.rbound) = count\_bounds(remove(i, itvs), i.rbound) + 1$$

$$\forall b.b \neq i.lbound \wedge b \neq i.rbound \Rightarrow (count\_bounds(itvs, b) = count\_bounds(remove(i, itvs), b))$$

This lemma is a bit tricky to prove, we will again use induction and call lemma `cons_count_lemma_2` similar to the above case when $hd = i$ and we will have to use `cons_count_diff_same` to show that the difference in the count will remain 1 when we insert hd at front of tl.

Lemma 5.4.11 `cons_count_diff_lemma`*: Given two lists of intervals $l1$ and $l2$, an interval i and a bound e,*

$$count\_bounds(l1, e) - count\_bounds(l2, e) = count\_bounds(i :: l1, e) - count\_bounds(i :: l2, e)$$

```
let rec remove_lemma_2 (itvs:list interval) (i:interval) : Lemma (requires (mem i itvs) /\ i.lbound <>
↪  i.rbound) (ensures (count_bounds itvs i.lbound = count_bounds (remove i itvs) i.lbound + 1) /\
↪  (count_bounds itvs i.rbound = count_bounds (remove i itvs) i.rbound + 1) /\ (forall b. (b <> i.lbound
↪  /\ b <> i.rbound) ==> (count_bounds itvs b = count_bounds (remove i itvs) b))) =
  match itvs with
  | [] -> ()
  | hd::tl -> if hd = i then cons_count_lemma_2 itvs
              else
                begin
                  remove_lemma_2 tl i;
                  assert(count_bounds tl i.lbound = count_bounds (remove i tl) i.lbound + 1);
                  cons_count_diff_lemma tl (remove i tl) hd i.lbound;
                  assert(count_bounds (hd::tl) i.lbound = count_bounds (hd::(remove i tl)) i.lbound + 1);
                  assert(count_bounds itvs i.lbound = count_bounds (remove i itvs) i.lbound + 1)
                end
```

Thus, we can prove `count_bounds_perm_lemma` as follwoing,

```
let rec count_bounds_perm_lemma (l1:list interval) (l2:list interval) : Lemma (requires is_permutation
↪  interval l1 l2) (ensures forall b. (count_bounds l1 b) = count_bounds l2 b) =
  same_length l1 l2;
  match l1 with
   | [] -> ()
   | hd::tl -> let rmx = (remove hd l2) in
               count_bounds_perm_lemma tl rmx;
               if hd.lbound = hd.rbound
               then
                 begin
                   remove_lemma_1 l2 hd;
                   cons_count_lemma_1 hd tl
```

```
    end
  else
    begin
      remove_lemma_2 l2 hd;
      cons_count_lemma_2 hd tl
    end
```

The base case of an empty list is trivial, we can use `remove_lemma_1` and `remove_lemma_2` to let F* know about `count_bounds` of $l2$ and `cons_count_lemma_1` and `cons_count_lemma_2` to let F* know about `count_bounds` of $l1$.

Finally, we can prove `perm_bounds_is_perm` lemma using `count_bounds_perm_lemma` and `count_bounds_lemma`.

```
let perm_bounds_is_perm (l1:list interval{length l1 > 0}) (l2:list interval{length l2 > 0}) : Lemma
↪  (requires is_permutation interval l1 l2) (ensures is_permutation nat (compute_bounds l1)
↪  (compute_bounds l2)) =
  assert(forall i. (count i l1) = count i l2);
  count_bounds_perm_lemma l1 l2;
  assert(forall b. (count_bounds l1 b) = count_bounds l2 b);
  count_bounds_lemma l1;
  count_bounds_lemma l2
```

And now for the second step,

We have written lemma `mem_impl_geq` which will be used to prove that both the lists have the same head,

Lemma 5.4.12 `mem_impl_geq`: *Given a sorted list l, a comparison function f and an element e,*

$$sorted(f, l) \wedge mem(e, l) \rightarrow f(get\_hd(l), e)$$

The proof is simply by induction until you reach $hd = e$ where it is trivial.

```
let rec mem_impl_geq (#a:eqtype) (l:list a{length l > 0}) (f:total_order a) (e:a) : Lemma (requires sorted
↪  f l /\ count e l > 0) (ensures f (get_hd l) e) =
  match l with
  | hd::tl -> if hd = e then ()
              else mem_impl_geq tl f e
```

Now we can write `sorted_perm_head_same` which will be used to prove that both the lists are equal.

Lemma 5.4.13 `sorted_perm_head_same`: *Given two lists $l1$ and $l2$,*

$$is\_permutation(l1, l2) \wedge sorted(l1) \wedge sorted(l2) \rightarrow get\_hd(l1) = get\_hd(l2)$$

Proving `sorted_perm_head_same` is simple as explained in point 2(b).

```
let sorted_perm_head_same (l1:list nat{length l1 > 0}) (l2:list nat{length l2 > 0}) : Lemma (requires
↪  (is_permutation nat l1 l2) /\ (sorted (fun (x y:nat) -> x <= y) l1) /\ (sorted (fun (x y:nat) -> x <=
↪  y) l2)) (ensures get_hd l1 = get_hd l2) =
  let h1 = get_hd l1 in
  let h2 = get_hd l2 in
  mem_impl_geq l1 (fun (x y:nat) -> x <= y) h2;
  mem_impl_geq l2 (fun (x y:nat) -> x <= y) h1
```

Now we can use induction on the length of $l1$ and $l2$ to prove that they are the same. We will be using `sorted_perm_head_same` and `cons_lemma` on $l1$ and $l2$ to show that $t1$ and $t2$ are permutations of one another.

```
let rec sorted_perm_same (l1:list nat) (l2:list nat) : Lemma (requires (is_permutation nat l1 l2) /\
↪  (sorted (fun (x y:nat) -> x <= y) l1) /\ (sorted (fun (x y:nat) -> x <= y) l2)) (ensures l1 = l2) =
  match l1,l2 with
  | [],[] -> ()
  | h1::t1,h2::t2 -> sorted_perm_head_same l1 l2;
                  cons_lemma t1 h1;
                  cons_lemma t2 h2;
                  sorted_perm_same t1 t2
```

Finally, we can prove the main lemma.

```
let eq_median (l1:list interval{length l1 > 0}) (l2:list interval{length l2 > 0}) : Lemma (requires
↪  is_permutation interval l1 l2) (ensures (median_bounds l1) = (median_bounds l2)) =
  let b1 = (compute_bounds l1) in
  let b2 = (compute_bounds l2) in
  perm_bounds_is_perm l1 l2;
  let sb1 = quicksort (fun (x y:nat) -> x <= y) b1 in
  let sb2 = quicksort (fun (x y:nat) -> x <= y) b2 in
  sorted_perm_same sb1 sb2
```

So, now we just need to prove that `itvs` and `to_itvs(it)` are permutations of each other. Then we can use `eq_median` lemma to prove that they will have the same median. This can be proved by following trivial lemmas.

Lemma 5.4.14 `partition_gives_perm`*: Given a list l and two functions $f1$ and $f2$,*

$$is\_permutation(l, (get\_mid(l, f1, f2)@(get\_left(l, f1, f2)@(get\_right(l, f1, f2)))))$$

Lemma 5.4.15 `pairwise_perm_is_perm`*: Given three lists $l1$, $l2$ and $l3$ and their permutations $m1$, $m2$ and $m3$ respectively,*

$$is\_permutation(l1, m1) \wedge is\_permutation(l2, m2) \wedge is\_permutation(l3, m3)$$

$$\downarrow$$

$$is\_permutation(l1@l2@l3, m1@m2@m3)$$

Thus, the following call to these lemmas will prove that `itvs` and `to_itvs(it)` are permutations of each other.

```
partition_gives_perm itvs (is_before mid) (is_after mid);
assert(is_permutation interval mid_list l);
assert(is_permutation interval left_list (to_itvs it1));
assert(is_permutation interval right_list (to_itvs it2));
pairwise_perm_is_perm mid left right l (to_itvs it1) (to_itvs it2);
assert(to_itvs it = l@(to_itvs it1)@(to_itvs it2));
assert(is_permutation interval itvs mid_list@left_list@right_list);
assert(is_permutation interval itvs (to_itvs it));
```

The assertions help us to understand how the calls to these lemmas convince F* that both lists are permutations.

5.4.5 `ordered` for `it1` and `it2`

This is already proven by the postcondition of `create` function.

5.4.6 Complete `create` function

```
val create (intervals:list interval) : Tot (it:interval_tree{(is_permutation interval intervals (to_itvs
↪  it))}) (decreases %[length intervals])
let rec create intervals =
  match intervals with
  | [] -> Empty
  | intervals ->
    let mid = (median_bounds intervals) in
    let left_list, mid_list, right_list = partition_upd intervals (is_before mid) (is_after mid) in
    let l = quicksort func_left mid_list in
    let r = quicksort func_right mid_list in
    assert(length mid_list > 0);
    same_length mid_list l;
    let left = (create left_list) in
    let right = (create right_list) in
    let l_itvs = (to_itvs left) in
    let r_itvs = (to_itvs right) in
    let it = Node mid l r left right in
    let it_itvs = (to_itvs it) in
    perm_check_eq mid_list l (contains mid);
    perm_check_eq left_list l_itvs (is_before mid);
    perm_check_eq right_list r_itvs (is_after mid);
    partition_gives_perm intervals (is_before mid) (is_after mid);
    pairwise_perm_is_perm mid_list left_list right_list l l_itvs r_itvs;
    eq_median intervals it_itvs;
    it
```

5.5 Querying interval tree

Below is the pseudo-code for the function that will query an interval tree for all the intervals containing a value.

```
function filter(l, x):
  if l is empty:
    return []
  else if contains x l.head:
    return l.head :: filter(l.tail, x)
  else:
    return []

function query(I, x):
  if I is empty:
    return []
  else if x < I.mid
    return query(I.left,x) @ filter(l,x)
  else if x >= I.mid
    return query(I.right,x) @ filter(r,x)
```

Notice that we use `filter` on `l` or `r` depending on whether `x` is less than or greater than or equal to `I.mid`. This is because `l` and `r` contains all the intervals that contain `I.mid`. So if `x` is less than `I.mid`, then we need to search `l` for all the intervals that contain `x`. But all intervals must have `rbound` greater than `x` and thus we should go in increasing order of `lbound` and stop once we find an interval that has `lbound` greater than `x`. Similarly, if `x` is greater than or equal to `I.mid`, then we need to search `r`.

We can write lemma `mem_impl_geq_forall` to prove that all intervals in the tail of a sorted list are greater than or equal to the head of the list.

Lemma 5.5.1 `mem_impl_geq_forall`*: Given a list l and a comparison function f,*

$$\forall e.(count\ e\ l > 0) \rightarrow f\ (get\_hd\ l)\ e$$

This is easily proved by doing induction on the list.

```
let rec mem_impl_geq_forall (#a:eqtype) (l:list a{length l > 0}) (f:total_order a) : Lemma (requires
↪  sorted f l) (ensures forall e. (count e l > 0) ==> f (get_hd l) e) =
  match l with
  | [] -> ()
  | hd::[] -> ()
  | hd::tl -> mem_impl_geq_forall tl f
```

We can then use this lemma to prove the correctness of `filter_left` and `filter_right` that these functions will return all the intervals in a sorted list that contains queried value.

```
let rec filter_left (xmid:nat) (l:list interval{sorted func_left l /\ check_list l (contains xmid)})
↪  (qx:nat{qx < xmid}) :  Tot (t:list interval{forall i. (mem i t) <==> ((mem i l) /\ (contains qx i))})
↪  =
  match l with
    | [] -> []
    | hd::tl -> if (contains qx hd) then hd::(filter_left xmid tl qx) else
              begin
              (* assert(hd.lbound > qx); *)
              mem_impl_geq_forall l func_left;
              (* assert(forall i. (mem i tl) ==> func_left hd i); *)
              []
              end

let rec filter_right (xmid:nat) (l:list interval{sorted func_right l /\ check_list l (contains xmid)})
↪  (qx:nat{qx >= xmid}) :  Tot (t:list interval{forall i. (mem i t) <==> ((mem i l) /\ (contains qx i))})
↪  =
  match l with
    | [] -> []
    | hd::tl -> if (contains qx hd) then hd::(filter_right xmid tl qx) else
              begin
              (* assert(hd.rbound < qx); *)
              mem_impl_geq_forall l func_right;
              (* assert(forall i. (mem i tl) ==> func_right hd i); *)
              []
              end
```

Finally, we can write the function `query` that will return all the intervals in the interval tree that contain the queried value.

```
val query (qx:nat) (it:interval_tree) : Tot (l:list interval{forall i. (mem i l) <==> ((contains qx i) /\
↪  (in_itv_tree i it))}) (decreases %[length (to_itvs it)])
let rec query qx it =
    match it with
    | Empty -> []
    | Node mid l r left right ->
      if qx < mid
      then (filter_left mid l qx)@(query qx left)
      else (filter_right mid r qx)@(query qx right)
```

The time complexity of this function is `O(log n + k)` where `n` is the number of intervals in the tree and `k` is the number of intervals that contain the queried value.

6 Extraction of OCaml code

F* has inbuilt functionality to extract code to OCaml. The generated code works similarly to the one already implemented[6], but both differ in a few functions.

The generated code is guaranteed to be verified and will behave as expected in all cases.

7 Future work

1. **Efficient interval `insert` and `delete`**: The implementation could be extended to support efficient interval insertion and deletion operations. One possible approach is to use a self-adjusting binary search tree structure, such as a splay tree, which can perform these operations in amortized logarithmic time.

2. **Range queries**: The implementation could be extended to support `range_query`, which returns all intervals in the tree that intersect a given range. One possible approach is to use a recursive traversal of the tree, similar to the interval search operation, but with additional checks to determine whether a subtree can be skipped based on the location of its interval relative to the range.

3. **Higher-dimensional intervals**: The implementation could be extended to support intervals in higher dimensions, such as rectangles in 2D or boxes in 3D. This would require extending the interval comparison function and the interval search operation to handle multiple dimensions.

4. **Additional interval data**: The `interval` type could be extended to store additional data, such as a priority value or a reference to an associated object. This would enable the implementation of priority queue or index structures based on interval comparison, similar to the use of binary heaps in priority queue implementations based on the numerical comparison.

References

[1] F* documentation. URL:`https://www.fstar-lang.org/`.

[2] M. de Berg et al. *Computational Geometry: Algorithms and Applications, pages 223–224*. Springer, third edition edition, 2008, DOI: 10.1007/978-3-540-77974-2. URL: `https://ieeexplore.ieee.org/document/597798`.

[3] P. Cignoni et al. *Speeding up isosurface extraction using interval trees.* 1997, DOI: 10.1109/2945.597798. URL: `https://ieeexplore.ieee.org/document/597798`.

[4] F* implementation of data structures. URL:`https://github.com/FStarLang/FStar/tree/master/examples/data_structures`.

[5] Interval Tree. URL:`https://en.wikipedia.org/wiki/Interval_tree`.

[6] Interval tree implementation in OCaml. URL:`https://github.com/UnixJunkie/interval-tree`.

[7] Dafny: verification-ready programming language. URL:`https://github.com/dafny-lang/dafny`.

[8] z3: SMT solver. URL:`https://github.com/Z3Prover/z3`.