Deliverable EC June 2017

Ann Weine

June 14, 2017

1 Introduction

The next part of the paper will be dedicated to F* and our implementation of skiplist in the language.
We will explain why this particular language was chosen for our project and why this data structure was chosen as the most suitable in terms of performance and usability.

2 FStar as a language for formalisation

F* (FStar) is a functional programming language aimed at program verification: FStar main page. The language includes dependent types, refinement types in the type system. It allows to write the efficient and functional correct code.
Nevertheless, there exist several languages that have the same approach: The Coq Proof Assistant, OCaml. The language was chosen due to the several reasons. One of them is that the written code could be easily extracted to C. It means that the code in C will have the same proofs of correctness/side-channel resistance as the source code. The second reason is the possibility to easily prove some specific security properties about the written code.

The language has already several successful applications. It is widely used for protocol verification. One of the most well-known applications is verified reference implementation of the TLS protocol: miTLS.

2.1 Important concepts

In the following section we will introduce several concepts that are useful for further understanding of the provided code.

2.1.1 Refinement types

One distinctive feature of F* is its use of refinement types. For example, the type

```
a:nat{a < 10}
```

is a refinement of the type nat. The subset of the natural numbers is decreased by the predicate

$\{a<10\}$

.

2.1.2 Effects

Some of functions could cause different effects during the computation. The most commonly used effect is Tot. This effect is guaranteed (provided the computer has enough resources) to evaluate to a t-typed result, without entering an infinite loop; reading or writing the program's state; throwing exceptions; performing input or output; or, having any other effect. whatsoever. Other effects could be:

- Dv, the effect of a computation that may diverge;

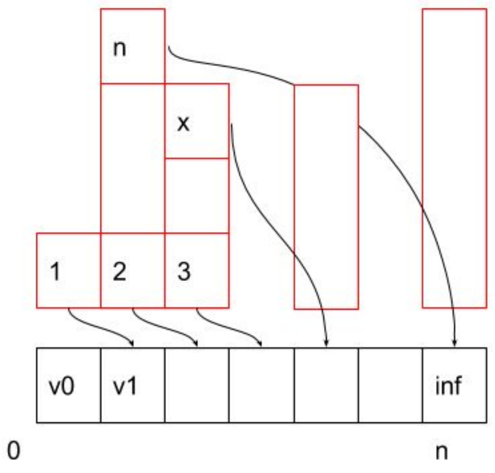
- ST, the effect of a computation that may diverge, read, write or allocate new references in the heap;
- Exn, the effect of a computation that may diverge or raise an exception.

3 SkipList

The next part will be dedicated to an explanation of skiplist as a data structure, the underlying concepts and proofs of efficiency.

3.1 Definition

Skiplist is a data structure that is used to store data. It is built based on usual linked list and combines the ideas of linked list and some specific features to increase the performance of several operations in skiplist.
Our implementation of skiplist consists of two data structures. One of them in used to store the data, while the second one serves as the indexes storage.



```
type skipList
(a:eqtype) (f:(a->a->Tot bool)) =
          | Mk:  values: seq(a){sorted f values}->
          indexes : seq(non_empty_list nat)
                    {Seq.length values = Seq.length indexes} -> skipList a f
```

First, it’s important to take a look on the sequence of values. The sequence is sorted. It means that each element with the index that is more than current one satisfies the predicate f.
Secondly, all the indexes are also stored in to sequence, that has the same length as the value sequence. For each value there exists at least one element it is referencing (and in this case it will be the next element - v1 in case of v0).
The lemma below shows that the data structure can be used as usual linked list (counter global is used to reference the indexes in skiplist):

```
lemma_linked_list : sl:skipList a f {Sl.length sl > 0} ->
counter_global:nat{counter_global < (Sl.length sl -1)} ->
          Lemma(ensures (last_element_indexed sl counter_global =
          counter_global +1))
```

In terms of memory correctness, we prove the lemma that shows that all the indexes that are used are smaller than the length of the data structure. It means that all the values indexes are not going out of the data structure (counter global in used for the referencing of the indexes, while counter local is used to reference the concrete index in array of indexes):

```
lemma_indexLessThanSize: sl: skipList a f{length sl> 0}  ->
          Lemma(ensures(forall (counter_global: nat {counter_global < length sl})
          (counter_local : nat {counter_local <List.length
                    (getIndex sl counter_global)}).
                    (fun (x: nat) -> x < (length sl))
          (List.index(getIndex sl counter_global) counter_local)))
```

All the indexes that the concrete value has references to are strictly more than the index of the value. It means that all the elements the value is referencing will be strictly more than the value. The following lemma proves the property(counter global in used for the referencing of the indexes, while counter local is used to reference the concrete index in array of indexes):

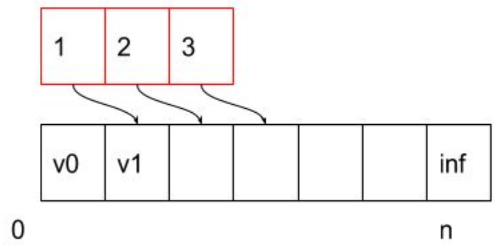
```
lemma_valuesMoreThanIndex: sl: skipList a f {length sl> 1} ->
          Lemma(ensures
                    (forall (counter_global: nat {counter_global < length sl})
                    (counter_local : nat
                              {counter_local <List.length
                                        (getIndex sl counter_global)}).
                    (fun (x: nat) -> x > counter_global)
          (List.index(getIndex sl counter_global) counter_local)))
```

The main draw of the data structure is that there is no need to go through to whole data structure to find an element, as it is done over traditional lists. The search routine will be explained in more details in the next sections.

The complexity of linked list is O(n), while the average complexities of search/insert/delete algorithms for skiplist is $log(n)$ where n is the number of elements stored in skiplist. Now we will show how the skiplist does the trick.

Let's start with a simple linked list. Each element of skiplist has just one link to the next element (or our data structure that uses just the first elements for each values

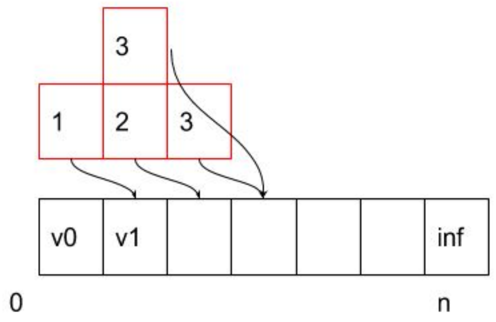
As it was discussed before, a skiplist has several layers of indexes. The first layer is used to store the next-coming value and in this case the skiplist could be respresented as a usual linked list.



```
type skipList1level(a:eqtype) (f:cmp a) =
| Mk:  values: seq(a){sorted f values}->
          indexes : seq nat{forall (y: nat{y < length indexes).
          indexes[y] = indexes[y]+1 } -> skipList a f
```

To find an element: assume that one is looking for a element a. A person goes through the all elements until he finds a.

As the next step, we assume that the index list now has a index not only on the next element, but also on the element two positions ahead:



```
type skipList(a:eqtype) (f:cmp a) =
| Mk:   values: seq(a){sorted f values}->
          indexes : seq (non_empty_list nat)
          {forall (y: nat {y< length indexes}).
          indexes[y][0] = indexes[y+1][0]+1 /\
          indexes[y][1] = indexes[y+1][1]+2 // for each second element
} -> skipList a f
```

It makes the search twice quicker. While going through the element v1, one checks whether the element value[3] is more than the element a. If it's the case, the search will be repeated starting from the element 3, skipping the second element.

If each fourth node has links to four nodes ahead, the search will be done 4 times faster.
If to continue the practice, such that i=0...$log(n)$ $1/2^i$ of the nodes will have a link to next 2^i elements, the search efficiency will be $O(log(n))$.
At the same time, it assumes that the structure should stay balances. It leads to the loss of performance during the insert and delete procedure because it's needed to rebuild the indexes. Nevertheless, it was shown (Skip Lists: A Probabilistic Alternative to Balanced Trees) that it's not important to keep the structure.
Instead, we use a random number generator during insertion to have a randomized structure with the performance close to ideal.
We associate a skiplist with some probability p. The probability of the new node to have 2 levels equals to 1/2, to have 3 levels equals to 1/4 etc.

```
assume val flipcoin : unit -> Tot(r: nat{r = 0 \/ r = 1})

val random : max: nat -> counter : nat{counter <=max} -> result:nat  ->
              Tot(nat)(decreases (max - counter))

let rec random max counter result =
     if counter = max then result else
          let flip = flipcoin () in
               if flip = 0 then result else
               let result = result + 1 in
               random max (counter + 1) result
```

Having several additional levels brings some memory overhead. The overhead for each node will be equal to the number of levels multiplied by the size for an index.
The data structure is very often used in the frameworks when it's important to provide a good level of performance, for example a database management systems: MemSQL.

3.2 API

Our implementation supports the following API:

- Inserting of a new element
- Searching for an existing element
- Search for an element by index
- Removing an existing element (this and the next one is not used in ClaimChain, it was implemented for consistency and re-use of data structure).
- Removing an existing element found by index
- Splitting a skiplist into 2 skiplists

3.3 Insert

The insertion is implemented as a routine to insert a new claim into the Claimchain.
In case of the list already exists, the insertion function is called, otherwise the creation function is called.
The creation function consists on the generation of the list with one element - infinity. This element has the biggest list of indexes and doesnot have a reference to any element. In other words, it's an element that means end of the list. In case of an element doesnot have an element that could be referenced, the reference will be put to the last element.

```
val create :  value_max : a -> elements_number: nat {elements_number > 0}
->Tot(sl: skipList a f {length sl = 1})
```

The insertion procedure could be divided into two subroutines: searching for place for the new element to be inserted and the insert itself. The algorithm for the place finding will be discussed in the next subsection.

The abstract procedure of insertion could be presented as follows:

```
let addition value sl max_level  =
    let place = searchPlace sl value in
    let level = generate_level place max_level in
    let values = change_values value sl place in
    let indexes = change_indexes sl place level in
    Mk values indexes
```

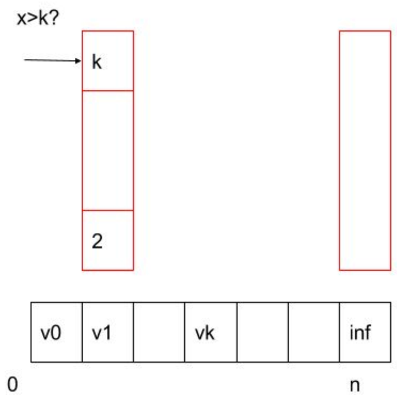
The searchPlace is a procedure that returns the place for the element 'value' to put. It ensures that the place satisfies the conditions: value[place] should be strictly more than 'value', value[place+1] should be strictly less than 'value'.

```
val searchPlace:  sl: skipList a f{Sl.length sl > 0} -> value: a->
Tot(place: nat{(
        (place = 0 /\ f value (Seq.index (getValues sl) 0))
        \/ (place < (Sl.length sl -1) /\
                f value (Seq.index (getValues sl) (place+1)) /\
                f (Seq.index (getValues sl) place) value ))
})
```

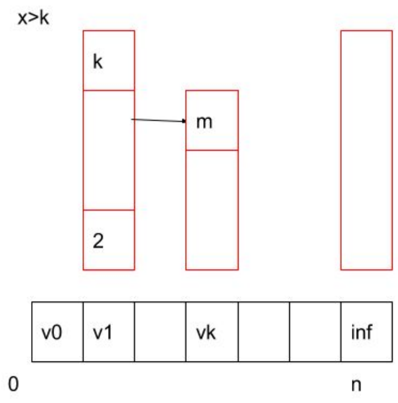
The *generateLevel* is a procedure that aims to return a number of elements that new value will have reference to. The *changeValues* procedure is a procedure that takes the existing sequence of values and return the sequence with the new element put on the place calculated by *searchPlace* routine. The *changeIndexes* procedure is a procedure that takes the existing sequence of indexes for each value and return the sequence with index list for the new element put on the place calculated by *searchPlace* routine.
As a result, we ensures that the element was inserted to the right position and that the new structure satisfies all the properties of skiplist.

3.4 Search

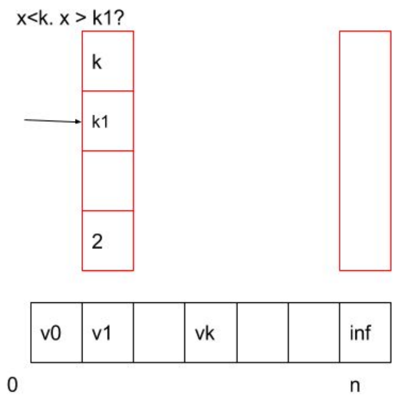
The search procedure is implemented as a routine in order to find an element in existing skiplist. It's a procedure that is used to find a particular claim in claimchain. Alternatively, it is used as a part of split, insert and remove procedures.
One aims to find a position of element v. The search procedure starts with the 0 position element. One compares first element with v. If they are not equal, one checks the indexes list:



One takes the biggest index element and compares whether it is bigger than v. If it's the case than the search routine starts again with an element referenced by the index:



Otherwise, the next biggest index is checked:



The algorithm is done either until the end of the skiplist, or until the element is found. First case means the absence of the element v in the list.
As the result, we get the position of the element v and ensures that the values[place] effectively has the element.

```
val search: sl: skipList a f {Sl.length sl > 1}-> element : a ->
Tot(option(place: nat { place < Sl.length sl /\ element = getValue sl place}))
```

Additionally, the search routine provides the following interfaces:

```
val nextElement: sl: skipList a f {Sl.length sl > 1}-> element: a -> Tot(option (a))
val nextElement: sl: skipList a f {Sl.length sl > 1}-> element: a -> Tot(option (a))
val exist: sl: skipList a f{Sl.length sl >1} -> element : a -> Tot(bool)
```

Search routine used in insert and remove routines is implemented the same way.

3.5 Split

The split procedure is implemented as a routine in order to split a skiplist into two different skiplists. Having the part of the skiplist could be useful several case. The most important cases includes the situation when one is needed to reason only about the part of skiplist. The possible application is to have a search routine until some existing element. In this case, the procedure could be implemented as a combination of splitting and searching.
The second possible application is to take a piece of information in case of key compromise. One should know the point until which the chain was trusted and to discard another part.
The split procedure is implemented as follows:

```
let split sl place max =
        let fst_v, fst_i = add_infinity fst_v fst_i max in
        let fst_i = reg_indexes fst_i place in
        let snd_i = right_part_reg_indexes place sl [place + 1] 0 in
        (Sl.Mk fst_v fst_i; Sl.Mk snd_v snd_i)
```

The *addInfinity* is a procedure that takes a splitted sequences of values and indexes and returns the sequence with a tail added. The *regIndexes* and *rightPartRegIndexes* procedures take the index lists of both sequences and regenerate them to be able to index the newly changed value sequences.
As a result, we ensure the new structures satisfy all the properties of skiplist.

The usual remove procedure uses the place of the element as a divisor:

```
val split: sl : skipList a f {Sl.length sl > 0} ->
        place: nat {place > 0 /\ place < Sl.length sl -1} ->
        Tot(skipList a f * skipList a f)
```

It could be preceded by a search routine. This function will provide an interface to split using the particular value of skiplist:

```
val split: sl : skipList a f {Sl.length sl > 0} ->
        element: a ->
        Tot(skipList a f * skipList a f)
```

3.6 Remove

The remove procedure is not used in claimchain due to the append-only policy, nevertheless, it was implemented for the consistency of the data structure.
The remove procedure is very closely implemented as a split procedure, with a small difference that the first element of the one of sequences will be removed and the skiplists will be connected again.

```
let remove sl place =
    let values_new = rebuildValues values place in
    let indexes_new = rebuildIndexes sl place in
    Sl.Mk values_new indexes_new
```

The *rebuildSequence* is a procedure that takes a sequence of values and returns the sequence with removed element indexed by place. The *rebuildIndexes* procedure is a procedure that takes the existing sequence of indexes for each value and returns the sequence with index list for particular place removed .
As a result, we ensure the new structure satisfies all the properties of skiplist and it decreased the length. The usual remove procedure uses the place of the element to be deleted:

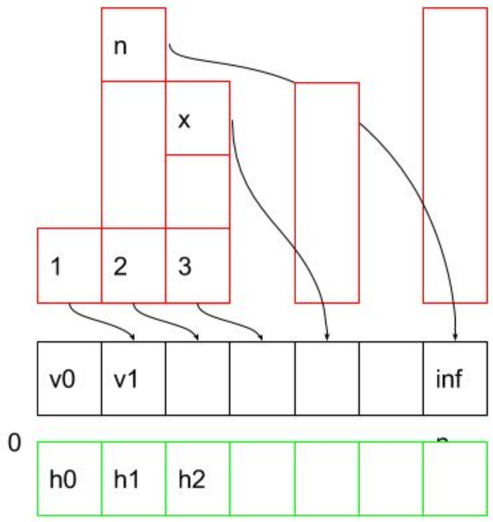
```
val remove:
        sl: skipList a f {Sl.length sl > 1} ->
        place : nat {place < Sl.length sl - 1} ->
        Tot(r: skipList a f {Sl.length sl = Sl.length r + 1})
```

It could be preceded by a search routine. This function will provide an interface to delete the particular value of skiplist:

```
val remove:
        sl: skipList a f {Sl.length sl > 1} ->
        element : a  ->
        ML(r: skipList a f {Sl.length sl = Sl.length r + 1})
```

3.7 Future development

As a future part of the work it will be implemented a skiplist with a hashes for all the elements.



```
type skipList
(a:eqtype) (f:(a->a->Tot bool)) (hash:(a -> hash))  =
| Mk:   values: seq(a){sorted f values}->
        indexes : seq(non_empty_list nat)
                {Seq.length values = Seq.length indexes}->
        hashes: seq (hash) {Seq.length hashed = Seq.length values /\
        {forall (y:nat{length values < y).
        Seq.index y hash = hash (Seq.index y values) + hash (Seq.index y
        -> skipList a f
```

A cryptographic hash function is a hash function which takes an input (or 'message') and returns a fixed-size alphanumeric string. The set of hash functions has the following property: it's computationally difficult to find two messages the hashes of which will be identical.
In terms of skiplist, the hash will consist of the hash of the data and the hash of the indexes.
This mechanism will give us some security guarantees of non-repudiation of the data. An auditor will be able to any moment of time to check that there was no data injection in between.
It will lead to a small change of memory consumption for the data structure. The additional memory will be equal to the size of hash of element and need to be muplіply by the number of all the elements. It will slightly change the addition procedure and the procedures that change either the value of the skiplist or the indexes: for each change it will be needed to regenerate a hash.

4 Merkle Tree

The next part will be dedication to an explanation of Merkle Tree as data structure, the underlying concepts and proofs.

4.1 Definition

Merkle tree is a data structure that is used for efficient data storage and the structure that makes it easier to prove the existence of the element.

Merkle tree is based on the tree. Each leaf of the tree has a value, each root contains a concatenation of hashes of two leaves:

```
open HashFunction

type hash = seq nat
type data = seq nat
type merkleTree: level:nat -> h: hash -> Type  =
| MLeaf: element : data -> merkleTree 0 (hashFunc element)
| MNode: #level: nat -> #h1: hash -> #h2 : hash ->
        lnode: merkleTree level h1 ->
        rnode: option(merkleTree level h2) ->
        merkleTree (level+1) (hashConcat h1 h2)
```

4.2 Proof of existence

To be able to prove the existence of the element in the merkle tree, we provide a path as a proof. The path consists on the list of bits (directions, which root, left or right should be chosen.
To get the element according to path:

```
val get_elt: #h:hash -> path:path -> tree:mtree (len path) h -> Tot data
             (decreases path)
let rec get_elt #h path tree =
  match path with
    | [] -> L?.data tree
    | bit::path' ->
      if bit then
        get_elt #(N?.h1 tree) path' (N?.left tree)
      else
        get_elt #(N?.h2 tree) path' (N?.right tree)
```

The path is provided to verifier with a tree. A verifier is able to compute the hash of the tree according to the provided path:

```
(*
 * verifier takes as input a proof stream for certain lookup path
 * and computes the expected root node hash
 *)
val verifier: path:path -> p:proof{lenp p = len path} -> Tot hash
let rec verifier path p =
  match path with
    | [] -> gen_hash (p_data p)

    | bit::path' ->
      match p_stream p with
        | hd::_ ->
          let h' = verifier path' (p_tail p) in
          if bit then
            gen_hash (Concat h' hd)
          else
            gen_hash (Concat hd h')


(*
 * prover function, generates a proof stream for a path
 *)
val prover: #h:hash ->
            path:path ->
            tree:mtree (len path) h ->
            Tot (p:proof{lenp p = len path})
            (decreases path)
let rec prover #h path tree =
  match path with
    | [] -> Mk_proof (L?.data tree) []

    | bit::path' ->
      let N #dc #hl #hr left right  = tree in
      if bit then
        let p = prover path' left in
        Mk_proof (p_data p) (hr::(p_stream p))
      else
        let p = prover path' right in
        Mk_proof (p_data p) (hl::(p_stream p))

(*
 * correctness theorem: honest prover's proof stream is accepted by the verifier
 *)
val correctness: #h:hash ->
                 path:path ->
                 tree:mtree (len path) h ->
                 p:proof{p = prover path tree} ->
                 Lemma (requires True) (ensures (verifier path p = h))
                 (decreases path)
let rec correctness #h path tree p =
  match path with
    | [] -> ()
    | bit::path' ->
      if bit then
        correctness #(N?.h1 tree) path' (N?.left tree) (p_tail p)
      else
        correctness #(N?.h2 tree) path' (N?.right tree) (p_tail p)
```

The last lemma shows that the only way a verifier can be tricked into accepting proof stream for an non existent element is if there is a hash collision.

```
type hash_collision =
     cexists (fun n -> cexists (fun (s1:mstring n) -> cexists (fun (s2:mstring n) ->
               u:unit{gen_hash s1 = gen_hash s2 /\ not (s1 = s2)})))

val security: #h:hash ->
              path:path ->
              tree:mtree (len path) h ->
              p:proof{lenp p = len path /\ verifier path p = h /\
                      not (get_elt path tree = p_data p)} ->
              Tot hash_collision
              (decreases path)
let rec security #h path tree p =
  match path with
    | [] -> ExIntro data_size (ExIntro (p_data p) (ExIntro (L?.data tree) ()))

    | bit::path' ->
      let N #dc #h1 #h2 left right = tree in
      let h' = verifier path' (p_tail p) in
      let hd = Cons?.hd (p_stream p) in
      if bit then
        if h' = h1 then
          security path' left (p_tail p)
        else
          ExIntro (hash_size + hash_size)
      (ExIntro (Concat h1 h2) (ExIntro (Concat h' hd) ()))
```

```
else
  if h' = h2 then
    security path' right (p_tail p)
  else
    ExIntro (hash_size + hash_size)
(ExIntro (Concat h1 h2) (ExIntro (Concat hd h') ()))
```

4.3 Proof of inclusion

As a proof of inclusion, we are planning to provide two different paths for the trees. Each path will correspond to the existing tree. First path describes the way to reach the existing tree in the new one, the second path shows the way to reach the added tree. Presence of both in merkle tree proves the correctness of addition.