Fragment Abstraction for Concurrent Shape Analysis

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh

Uppsala University

1 Introduction

Concurrent algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state are of central importance in a large number of software systems. They provide efficient concurrent realizations of common interface abstractions, and are widely used in libraries, such as the Intel Threading Building Blocks or the `java.util.concurrent` package. They are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking when possible. A number of bugs in published algorithms have been reported [9, 27]. Consequently, significant research efforts have been directed towards developing techniques to verify correctness of such algorithms. One important use of such techniques is to verify that concurrent algorithms that implement standard data structure interfaces are *linearizable*, meaning that each method invocation can be considered to occur atomically at some point between its call and return. Many of the developed verification techniques require significant *manual* effort for constructing correctness proofs (e.g., [25, 39]), in some cases with the support of an interactive theorem prover (e.g., [37, 6, 7, 32, 31]). Development of automated verification techniques remains a difficult challenge.

A major challenge for the development of automated verification techniques is that such techniques must be able to reason about fine-grained concurrent algorithms that are infinite-state in many dimensions: they consist of an unbounded number of concurrent threads, which operate on an unbounded domain of data values, and use unbounded dynamically allocated memory. Perhaps the hardest of these challenges is that of handling dynamically allocated memory. Consequently, existing techniques that can automatically prove correctness of such fine-grained concurrent algorithms restrict attention to the simple case where heap structures represent shared data by singly-linked lists [1, 17, 2, 33, 40]. Furthermore, many of these techniques impose additional restrictions on the considered verification problem, such as bounding the number of accessing threads [3, 43, 41]. [Add other restrictions] However, in many concurrent data structure implementations the heap represents more sophisticated structures, such as skiplists [13, 22, 35] and arrays of of singly-linked lists [8]. There are no techniques that have been applied to automatically verify concurrent algorithms that operate on such data structures.

The list of description of restrictions of SSL work must be improved. We actually should be more precise about what previous work has achieved, in terms of which combinations of challenges have been overcome

Contributions In this paper, we present a technique for automatic verification of concurrent data structure implementations that operate on dynamically allocated heap structures which are more complex than just singly-linked lists. Our approach is the first framework that can automatically verify concurrent data structure implementations that singly linked lists, skip lists [13, 22, 35], as well as arrays of singly linked lists [8], at the same time as handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap.

Our technique is based on a novel shape abstraction, called *fragment abstraction*, which in a simple and uniform way is able to represent different kinds of unbounded heap structures. Its main idea is to represent a set of heap states by a set of *fragments*. A fragment consists of a pair of heap cells that are connected by a pointer field. For each of its cells, the fragment represents the values of its non-pointer fields as well as information about how it can be reached from pointer variables. The latter information is both *local*, saying which pointer variables point directly to the cell, and *global*, saying how the cell can reach to and be reached from (by following chains of pointers) other heap cells that are pointed to by global variables. A set of fragments represents the set of heap structures in which each pair of pointer-connected nodes are represented by some fragment in the set. Inuitively, a set of fragments describes the set of heaps that can be formed by "pieced together" fragments in the set. The combination of local and global information in fragments allows the verification to reason about the sequence of cells that can be "seen" by threads as the traverse the heap by following pointer fields in cells and pointer variables, thereby enabling automatic verification of reachability properties. The local information captures which cell patterns may be seen while traversing the heap, whereas the global information also captures whether certain accesses (also by other threads) will be possible in the future.

Fragment abstraction can and should be, in a natural way, be combined with data abstractions for handling unbounded data domains and with tread abstractions for handling an unbounded number of threads. For the latter we adapt the successful thread-modular approach [4], which constructs a representation of the local state of a single, but arbitrary thread, together with part of the global state and heap that is accessible to that thread. Our combination of fragment abstraction, thread abstraction, and data abstraction results in a finite abstract domain, thereby guaranteeing termination of our analysis.

We have implemented our approach and applied it to automatically verify correctness, in the sense of linearizability, of a large number of concurrent data structure algorithms. More specifically, we have automatically verified linearizability of most linearizable concurrent implementations of sets, stacks, and queues, which emply singly-linked lists, skip lists, or arrays of timestamped singly-linked lists, which are known to us in the literature on concurrent data structures.

For this verification, we we specify linearizability using the simple and powerful technique of *observers* [1], which can be seen as monitors that report violations of the linearizability criterion. Observers synchronize with the monitored concurrent programs at designated actions. This can be done in two ways. (1) For concurrent implementations of stacks and queues, linearizability can be precisely specified by observers that synchronize on call and return actions of methods, as shown by [5, 20]; this is done without any user annotation. (2) For sets, the verification requires the user to annotate how linearization points are placed in each method; in most cases this is a small burden for the verifier. The observer then synchronizeson these linearization points. Our implementation then automatically checks, using our novel technique based on fragment abstraction, that a supplied C-like description of a concurrent data structure is a correct linearizable implementation of a stack, queue, or set.

The fact that our fragment abstraction has been able to automatically verify all supplied concurrent algorithms, also those that employ skiplists or arrays of SLLs, indicates that the fragment abstraction is a simple mechanism for capturing both the local and global information about heap cells that is necessary for verifying correctness, in particular for concurrent algorithms where an unbounded number of threads interact via a shared heap.

Here goes the outline

Related Work [14] presents a thread-modular shape analysis for multi-threaded programs in which locks protect portions of the heap, and threads have to acquire a lock before accessing the corresponding portion of the heap. Fine-grained concurrent algorithms do not follow this pattern.

Thread-modular approaches for verifying fine-grained concurrent algorithms have been presented in several works. [1] introduce observers to specify the semantics of data structures. It verifies program using the thread-modular approach, where heaps are specified by reachability constraints between cells pointed to by program variables. The work is applied to concurrent stack and queue implementaitons based on singly-linked lists. [23] present a more efficient way to handle the expensive interference steps for common programming idioms in lock-free data structures. [2] extend this approach to be able to verify linearizability for a large class of concurrent data structures based on singly-linked lists. It used a specifically designed finitary abstraction of unbounded singly-linked list segments.

[We should say something about to CAVE]

[We should have a paragraph on verification of skiplists, and on verification of the TS queue/stack]

Below is related work from the SAS paper

Much previous work has been devoted to the *manual* verification of linearizability for concurrent programs. Examples include [25, 37]. In [30], O'Hearn *et al.* define a *hindsight lemma* that provides a non-constructive evidence for linearizability. The lemma is used to prove linearizability of an optimistic variant of the lazy set algorithm. Vafeiadis [39] uses forward and backward simulation relations together with history or prophecy variables to prove linearizability. These approaches are manual, and without tool implementations. *Mechanical* proofs of linearizability, using interactive theorem provers, have been reported in [6, 7, 32, 31]. For instance, Colvin *et al.* [6] verify the lazy set algorithm in PVS, using a combination of forward and backward simulations.

Several techniques for verifying linearizability are based on establishing commutation properties between atomic actions inside each method. Such methods can be partly automated [12, 34] or part of automated program analysis [24].

There are several works on *automatic* verification of linearizability. In [40], Vafeiadis develops an automatic tool for proving linearizability that employs instrumentation to verify logically pure executions. However, this work can handle non-fixed LPs only for read-only methods, i.e, methods that do not modify the heap. This means that the method cannot handle algorithms like the *Elimination* queue [29], *HSY* stack [19], *CCAS* [16], *RDCSS* [16] and *HM* set [22] that we consider in this paper. In addition, their shape abstraction is not powerful enough to handle algorithms like *Harris* set [15] and *Michael* set [26] that are also handled by our method. Chakraborty *et al.* [20] describe an "aspect-oriented" method for modular verification of concurrent queues that they use to prove linearizability of the Herlihy/Wing queue. Bouajjani et al. [5] extended this work to show that verifying linearizability for certain fixed abstract data types, including queues and stacks, is reducible to control-state reachability. We can incorporate this technique into our framework by a suitable construction of observers. The method can not be applied to sets. The most recent work of Zhu *et al.* [45] describe a tool that is applied for specific set, queue, and stack algorithms. For queue algorithms, their technique can handle queues with helping mechanism except for *HW* queue [21] which is handled by our paper. For set algorithms, the authors can only handle those that perform an optimistic contains (or lookup) operation by applying the *hindsight lemma* from [30]. Hindsight-based proofs provide only *non-constructive* evidence of linearizability. Furthermore, some algorithms (e.g., the unordered list algorithm considered in Sec. 8 of this paper) do not contain the code patterns required by the hindsight method. Algorithms with non-optimistic contains (or lookup) operation like *HM* [22], *Harris* [15] and *Michael* [26] sets cannot be verified by their technique. Vechev *et al.* [43] check linearizability with user-specified non-fixed LPs, using a tool for finite-state verification. Their method assumes a bounded number of threads, and they report state space explosion when having more than two threads. Dragoi *et al.* [11] describe a method for proving linearizability that is applicable to algorithms with non-fixed LPs. However, their method needs to rewrite the implementation so that all operations have linearization points within the rewritten code. Černý *et al* [41] show decidability of a class of programs with a bounded number of threads operating on concurrent data structures. Finally, the works [1, 4, 38] all require fixed linearization points.

We have not found any report in the literature of a verification method that is sufficiently powerful to automatically verify the class of concurrent set implementations based on sorted and non-sorted singly-linked lists having non-optimistic contains (or lookup) operations we consider. For instance the lock-free sets of *HM* [22], *Harris* [15], or *Michael* [26], or unordered set of [44],

2 Overview

In this section, we illustrate our technique by using it to prove correctness, in the sense of linearizability, of a concurrent data structure implementation which uses skiplists. We consider an implementation of a set data structure, which operates on a shared heap which represents a skiplist. Here, we consider the Lock-Free Concurrent Skiplist, described in [22, Section 14.4]. To the best of our knowledge, no existing automated verification technique has succeeded in verifying functional correctness of concurrent skiplist algorithms. We have automatically verified both lock-based and lock-free skiplist-based algorithms using fragment abstraction, as reported in Section 8.

2.1 The Skiplist Algorithm

To illustrate our approach, we use the lock-free concurrent lock-free skiplist algorithm [22, Section 14.4]. Informally, a skiplist consists of a collection of sorted linked lists, each of which is located at a *level*, ranging from 1 up to a maximum value. Each skiplist node has a key value and participates in the lists at levels 1 up to its *height*. The skiplist has sentinel head and tail nodes with maximum heights and key values $-\infty$ and $+\infty$, respectively. The lowest-level list (at level 1) constitutes an ordered list of all nodes in the skiplist. Higher-level lists are increasingly sparse sublists of the lowest-level list, and serve as shortcuts into lower-level lists.

There are three main methods of the algorithm namely `add`, `contains` and `remove`. All methods rely on a method `find` to search for a given key. In this section, we shortly describe the `find` and `add` methods. Figure 1 shows code for these methods.

```
struct Node { int data; int height; Node next[]; boolean marked[];}
```

```
boolean find(int x,Node preds[],Node succs[])
boolean marked[] = {false};
boolean s;
Node* pred = null,curr = null,succ = null;
retry:
while (true)
  pred = head;
  for (int i = MAXHEIGHT; i >= 1; i--)
    curr = pred.next[i];
    while (true)
      atomic { succ = curr.next[i];
               marked[i] = curr.marked[i];}
      while (marked[0])
        s=CAS(pred.next[i],curr,succ,false
                                     false);
        if (!s) goto retry;
        curr = pred.next[i];
        atomic { succ = curr.next[i];
                 marked[i] = curr.marked[i];}
      if (curr.key < x)
        pred = curr;
        curr = succ;
      else break;
    preds[i] = pred;
    succs[i] = curr;
  return (curr.key == x);
```

```
boolean add (int x):
int h = randomLevel;
Node* preds[]; succs[]
while (true);
  if find(x,preds,succs) ●
   return false;
  else
   Node* n = new Node(x, h);
   for(int i = 1;i <= h; i++)
    Node* succ = succs[i];
    atomic { new.next[i] = succ;
             new.marked[i] = false; }
   Node* pred = preds[1];
   Node* succ = succs[1];
   atomic { new.next[1] = succ;
            new.marked[1] = false;}
   if !CAS(pred.next[1],succ,n,false,false)
    goto 5;
   else ●
    for (int i = 2; i <= h; i++)
     while (true);
      pred = preds[i];
      succ = succs[i];
      if CAS(pred.next[i],succ,n,false,false)
       break;
      find(x,preds,succs);
return true;
```

Fig. 1: Code for the `find` and `add` methods of the skiplist algorithm.

Each heap cell has a `key` field, a `marked` field which is true if the node has been logically removed, and an array of `next` pointers, indexed from 1 upto its `height`. The `find` method traverses the list at decreasing levels, starting from the maximum level. During the traversal, the two pointer variables pred and curr are advanced until pred points to a node with the largest key on that level smaller than x, recording the obtained values of pred and curr into the arrays preds and succs at the current level, and thereafter continuing one level below. During the traversal, the method removes marked nodes at the current level (lines 15 to 19) using a compareAndSet command. The compareAndSet statement tests the expected reference and mark values, and if both tests succeed, replaces them with updated reference and mark values. The compareAndSet command returns a Boolean indicating whether the value changed.

The add method uses find to check whether a node with key x is already in the list. If an unmarked node with key x is found in the bottom-level list, the add method returns false. If no node is found, then the next step is to try to add the new node into the list. It is performed by linking it into the bottom-level list between the preds[0] and succs[0] nodes returned by find. using a compareAndSet to set the reference while validating that these nodes still refer one to the other and have not been removed from the list (line 16). If compareAndSet fails, the method call will restart. Otherwise, the node is added into the list. Thereafter, the add proceeds with linking the new node into increasingly higher levels.

2.2 Specifying the Correctness Criterion of Linearizability

In our verification, we establish that the skiplist algorithm of Figure 1 is correct in the sense that it is a linearizable implementation of a set data structure. Linearizability intuitively states that each operation on the data structure can be considered as being performed atomically at some point (called the *linearization point (LP)*) between its invocation and return [21]. We specify linearizability by the technique of *observers* [1], including its subsequent extensions [5, 20, 2]. In the case of our skiplist algorithm, LPs can be associated to fixed statements in the code. The user then instruments these statements to announce the corresponding set operation. For instance, the LP of a successful add operation is at line 16 of the add method, whereas the linearization point of an unsuccessful add operation is at line 7 of the add method. [Put these LPs as dots in the code] Having instrumented methods at LPs, we then consider an arbitrary concurrent program consisting of an arbitrary collection of threads, each of which executes some method call. We must check that the concurrent execution of such a program generates (through its instrumentation) a sequence of operations which satisfies the semantics of the set data structure. This check is performed by an *observer*, which monitors the sequence of operations that is announced by the instrumentation, and reports when it violates the semantics of the set data structure by moving to an accepting "error state". Observers are described in more detail in Section 3.3.

To verify that no execution of the program may cause the observer to accept, we form the cross-product of the program and the corresponding observer, in which the observer synchronizes with the program on the operations that are announced at LPs. This reduces the problem of checking linearizability to a reachability problem: checking that, in the cross-product, the observer cannot reach an accepting state.

In conclusion, verifying linearizability for skiplist-based set implementations requires the user to associate linearization points with statements in the methods of the implementation: the remaining steps are then automated. We remark that this user annotation is not necessary when verifying linearizability of stack and queue implementation. This is achieved by exploiting recent results saying that linearizability for stacks and queues can be precisely specified by observers that process the sequence of call and return actions of methods, instead of the user-supplied LPs [5, 20]. We use this technique for stacks and queues in Section XXX.

2.3 Verification by Fragment Abstraction

In the actual verification, we must compute a symbolic representation of an invariant that is satisfied by all reachable configurations of the cross-product of the program and an observer. The verification must address the challenges of an unbounded domain of data values, an unbounded number of concurrently executing threads, and an unbounded heap. For this, we have developed a novel shape representation, called *fragment abstraction*, which we combine with data abstraction and thread abstraction. Let us illustrate how fragment abstraction applies to the skiplist algorithm.

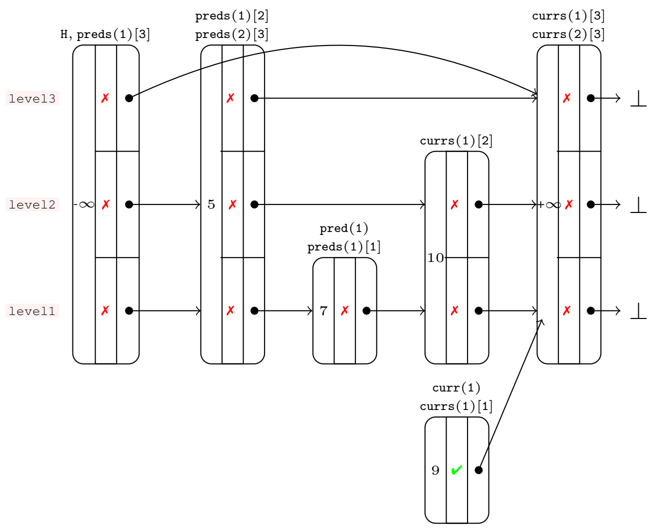


Fig. 2: A concrete shape of 3-level skiplist with three threads

Figure 2 shows an example state of the heap of the skiplist algorithm with three levels. Each heap cell is shown with the values of its fields. [Add a legend, showing the layout of fields, as in previous papers] In addition, each cell is labeled by the pointer variables that point to it; we use `lvar`(i) to denote the local variable `lvar` of thread $\mathtt{th}_i$. In the heap state of Figure 2, thread $\mathtt{th}_1$ is trying to add a new node with key 9, and has reached line 8 of the `add` method. Thread $\mathtt{th}_2$ is also doing the same thing as $\mathtt{th}_1$ and it has done its first iteration of the loop in `find` method in line 25.

One of the main properties of the algorithm is that the `pred` variable arrives at the bottom-level list at a node before, and never after, the target node. If the node is physically removed before the `find` starts, then it will not be found.

Our verification technique is based on a symbolic representation of program configurations, which combines a thread abstraction, a data abstraction, and a shape abstraction.

- Our *thread abstraction* adapts the thread-modular approach by representing only the view of single, but arbitrary, thread `th`. Such a view consists of (i) the local state of thread `th`, (ii) the part of the heap that is accessible to thread `th` via pointer variables (local to `th` or global), and (iii) the state of the observer.
- We use a novel *shape abstraction*, which represents the part of the heap that is accessible to `th` by a set of *fragments*. A fragment consists of a pair of heap cells that are connected by a pointer. For each of these cells, the fragment represents (i) the values of its non-pointer fields, (ii) the pointer variables (either local to `th` or global) that point to the cell, and (iii) global reachability information, which expresses how each node in the pair can reach to and be reached from (by following a chain of pointers) a finite set of globally significant heap cells. A cell is globally significant if it is pointed to by a global pointer variable or if its `data` field has the same value as the register of the set observer. In our fragment abstraction, a set of fragments represents the set of heap structures in which each pair of pointer-connected nodes is represented by some fragment in the set. Intuitively, a set of fragments describes the set of heaps that can be formed by "piecing together" fragments in the set. This "piecing together" must be both locally consistent (appending only fragments that agree on their common node), and globally consistent (respecting the global reachability information).
- A natural data abstraction is applied to the non-pointer local variables of the thread, the non-pointer fields of each fragment, as well as the observer registers. For our skiplist algorithm, fields that range over small finite domains are represented by their concrete values, whereas the observer registers and fields that range over the domain of data values are represented by constraints over their relative ordering (wrp. to the order $<$).

Our combination of fragment abstraction, thread abstraction, and data abstraction results in a finite abstract domain, thereby guaranteeing termination of our analysis.

Figure 3 shows a set of fragments that represents the part of the heap that is accessible to $\mathtt{th}_1$ in the configuration in Figure 2. We assume that the value of the observer register is 9. There are 10 fragments, named $\mathtt{v}_1, \ldots, \mathtt{v}_{10}$. Two of these ($\mathtt{v}_5$ and $\mathtt{v}_9$) consist of a tag that points to $\perp$, and the other consist of a pair of pointer-connected tags. The fragments $\mathtt{v}_1, \ldots, \mathtt{v}_6$ are abstracted from the level 1, whereas the other fragments are abstracted from higher levels. As can be seen, the property where the `pred`, in bottom level, never refers to an unmarked node whose key is greater than or equal to `x` is preserved by the abstraction. The data field of the tag which contains `pred` in fragments $\mathtt{v}_9$ is smaller than `x`. In addition, there is no tag whose marked field is `true` and contains `x` in its reach to information, it means that the abstraction preservers the property where the node which is physically removed can not be found by `find` method. [The below if for TS Stack. Should it be replaced or removed?]

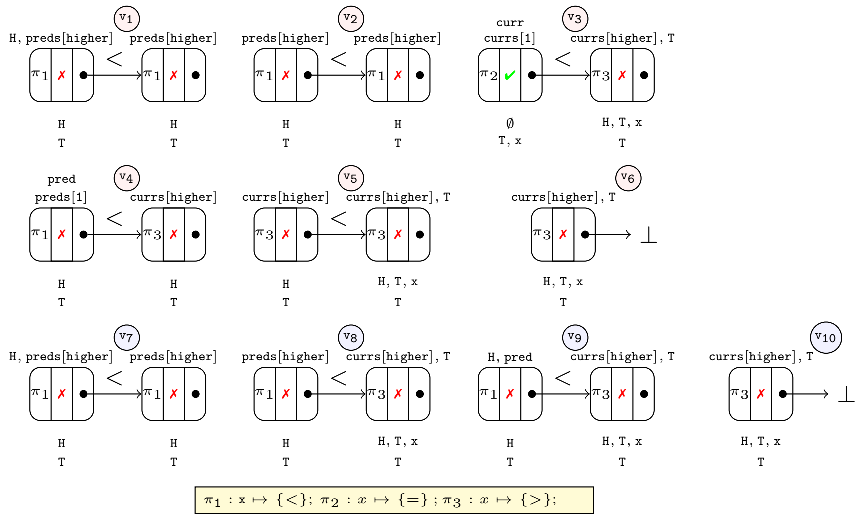


Fig. 3: Fragment abstraction of skiplist algorithm

- The `data` field of the tag (to the left) abstracts the data value 2 to the set of observer registers with that value: in this case x_2.
- The `ts` field (at the top) abstracts the timer value 15 to the possible relations with `ts`-fields of heap cells with the same data value as each observer registers. Recall that observer registers x_1 and x_2 have values 4 and 2, respectively. There are three heap cells with `data` field value 4, all with a `ts` value less than 15. There is one heap cell with `data` field value 2, having `ts` value 15. Consequently, the abstraction of the `ts` field maps x_1 to $\{>\}$ and x_2 to $\{=\}$: this is shown as the mapping λ_4 in Figure **??**.
- The `mark` field assumes values from a small finite domain and is represented precisely as in concrete heap cells

Above the top, the tag contains the thread-local and global pointer variables that point to the cell, in this case `youngest` and `n`. At the bottom of the tag, the first row contains the global variables pointing to cells from which the cell can be reached, in this case `pools`[3], as well as observer registers whose value is equal to the `data` field of a cell from which the cell can be reached, in this case x_2 (since the cell itself has the same data value as x_2). The second row contains dual information: now for cells that can be reached from the cell itself (this is again x_2).

Fix the issue with `pools`[3]

Each cell in the heap state of Figure **??** now satisfies some tag in Figure **??**. Moreoever, each pair of pointer-connected cells (where the pointed-to “cell” can also be $\perp$) satisfies some fragment in Figure **??** in the obvious way. Conversely, the set of fragments in Figure **??** represents the set of heaps in which each pair of pointer-connected cells satisfies one of its fragments. For instance, the list pointed to by `pools`[3] is represented by the sequence of fragments $v_4 v_5 v_6 v_7$.

Maybe we should omit the rest of this section?

In order to obtain a complete representation of reachable program configurations, we must also represent the local states of a thread. This is done in a standard manner, by applying the same data abstraction as for heap cells. For instance, the local state of thread $\mathtt{th}_2$ corresponding to Figure **??** is represented by a *local symbolic configuration* that contains the values of the program counter and variable `success`, abstracts the value of `k` into the set $\{\mathtt{me}, \mathtt{ot}\}$ and applies the timestamp abstraction to `maxTS`.

In the verification, we must compute a symbolic representation that is satisfied by all reachable program configurations (recall that program configurations include the state of the observer). This invariant is obtained by an abstract-interpretation-based fixpoint procedure, which starts from a representation of the set of initial configurations, and thereafter repeatedly performs postcondition computations that extend the symbolic representation by the effect of any execution step of the program, until convergence. This procedure is presented in Section 4.1.

3 Concurrent Data Structure Implementations

In this section, we introduce our representation of concurrent data structure implementations, we define the correctness criterion of linearizability, we introduce observers and how to use them for specifying linearizability.

3.1 Concurrent Data Structure Implementations

We first introduce (sequential) data structures. A *data structure* `DS` is a pair $\langle \mathbb{D}, \mathbb{M} \rangle$, where $\mathbb{D}$ is a (possibly infinite) *data domain* and $\mathbb{M}$ is an alphabet of *method names*. An *operation op* is of the form $\mathtt{m}(d^{in}, d^{out})$ where $\mathtt{m} \in \mathbb{M}$ is a method name and d^{in} are the *input* resp. *output* values, each of which is either in $\mathbb{D}$ or in some fixed finite domain (such as the booleans). For some method names, the input or output value is absent from the operation. A *trace* of `DS` is a sequence of operations. The (sequential) semantics of a data structure `DS` is given by a set $[\![\mathtt{DS}]\!]$ of allowed traces.

Quy: replace the below by an example of a set trace.

For example, for the `Stack` data structure, the method names are `push` and `pop`. The operation $\mathtt{push}(3)$, where 3 is an input value, pushes the value 3, whereas $\mathtt{pop}(4)$, where 4 is an output value, pops the value 4.

A *concurrent data structure implementation* operates on a shared state consisting of shared global variables and a shared heap. It assigns, to each method name, a method which performs operations on the shared state. Each data structure implementation also comes with an initialization method, named `init`, which initializes its shared state.

We assume that all global variables are pointer variables. Heap cells have a fixed set $\mathcal{F}$ of fields, namely data fields that assume values in $\mathbb{D}$ or $\mathbb{F}$, and possibly lock fields. We use the term $\mathbb{D}$-field for a data field that assumes values in $\mathbb{D}$, and the terms $\mathbb{F}$-field and lock field with analogous meaning. Furthermore, each cell has one or several named pointer fields. For instance, in data structure implementations based on singly linked lists each heap cell has a pointer field named `next`; in implementations based on skiplists there is an array of pointer fields named `next[k]` where `k` ranges from 1 to the maximum level of the skiplist.

Each method declares local variables and a method body. The set of local variables include the input parameter of the method and the program counter `pc`. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. Variables are either pointer variables (to heap cells) or data variables, assuming values from $\mathbb{D}$ or from some finite set $\mathbb{F}$ that includes the boolean values. The body is built in the standard way from atomic commands, using standard control flow constructs (sequential composition, selection, and loop constructs). Method execution is terminated by executing a `return` command, which may return a value. Atomic commands include assignments between data variables, pointer variables, or fields of cells pointed to by a pointer variable. The command `new Node()` allocates a new structure of type `Node` on the heap, and returns a reference to it. The compare-and-swap command `CAS(&a,b,c)` atomically compares the values of `a` and `b`. If equal, it assigns the value of `c` to `a` and returns `true`, otherwise, it leaves `a` unchanged and returns `false`. [Quy: You may want instead to describe Compare-and-set] We assume a memory management mechanism, which automatically collects garbage, and ensures that a new cell is fresh, i.e., has not been used before; this avoids the so-called ABA problem (e.g., [28]).

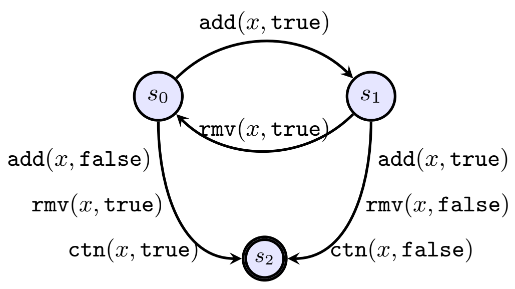
We define a *program* (over a concurrent data structure) to consist of an arbitrary number of concurrently executing threads, each of which executes a method that performs an operation on the data structure. We assume concurrent execution according to sequentially consistent memory model. We assume that the data structure has been initialized by the `init` method prior to the start of program execution.

3.2 Linearizability

In a concurrent data structure implementation, we represent the calling of a method by a *call action* $\mathtt{call}_{\mathtt{o}}\ \mathtt{m}\left(d^{in}\right)$, and the return of a method by a *return action* $\mathtt{ret}_{\mathtt{o}}\ \mathtt{m}\left(d^{out}\right)$, where $\mathtt{o} \in \mathbb{N}$ is an *action identifier*, which links the call and return of each method invocation. A *history* h is a sequence of actions such that (i) different occurrences of return actions have different action identifiers, and (ii) for each return action a_2 in h there is a unique *matching* call action a_1 with the same action identifier and method name, which occurs before a_2 in h. A call action which does not match any return action in h is said to be *pending*. A history without pending call actions is said to be *complete*. A *completed extension* of h is a complete history h' obtained from h by appending (at the end) zero or more return actions that are matched by pending call actions in h, and thereafter removing the call actions that are still pending. For action identifiers $\mathtt{o}_1, \mathtt{o}_2$, we write $\mathtt{o}_1 \preceq_{\mathtt{h}} \mathtt{o}_2$ to denote that the return action with identifier $\mathtt{o}_1$ occurs before the call action with identifier $\mathtt{o}_2$ in h. A complete history is *sequential* if it is of the form $a_1 a'_1 a_2 a'_2 \cdots a_n a'_n$ where a'_i is the matching action of a_i for all $i : 1 \leq i \leq n$, i.e., each call action is immediately followed by the matching return action. We identify a sequential history of the above form with the corresponding trace $op_1 op_2 \cdots op_n$ where $op_i = \mathtt{m}(d_i^{in}, d_i^{out})$, $a_i = \mathtt{call}_{\mathtt{o}_i}\ \mathtt{m}\left(d_i^{in}\right)$, and $a_i = \mathtt{ret}_{\mathtt{o}_i}\ \mathtt{m}\left(d_i^{out}\right)$, i.e., we merge each call action together with the matching return action into one operation. A complete history h' is a *linearization* of h if (i) h' is a permutation of h, (ii) h' is sequential, and (iii) $\mathtt{o}_1 \preceq_{\mathtt{h}'} \mathtt{o}_2$ if $\mathtt{o}_1 \preceq_{\mathtt{h}} \mathtt{o}_2$ for each pair of action identifiers $\mathtt{o}_1$ and $\mathtt{o}_2$. A sequential history h' is *valid* wrt. DS if the corresponding trace is in $[\![\mathtt{DS}]\!]$. We say that h is *linearizable* wrt. DS if there is a completed extension of h, which has a linearization that is valid wrt. DS. We say that a program $\mathcal{P}$ is linearizable wrt. DS if, in each possible execution, the sequence of call and return actions is *linearizable* wrt. DS.

3.3 Specification by Observers

To verify correctness of a data structure implementation, we must verify that any history, i.e., sequence of call and return actions, of any program execution satisfies the linearizability criterion. This is equivalent to requiring that each operation on the data structure can be considered as being performed atomically at some point (called the *linearization point (LP)*) between its invocation and return. To check linearizability of set implementations, the user first instruments each method so that it generates a corresponding operation precisely when the method executes its LP, using the technique of linearization policies [2]. In the case of the skiplist algorithm, this instrumentation simply consists of associating LPs to fixed statements in the code. [Quy: Please illustrate here how this is done for the skiplist] Thereafter, it should be checked that the concurrent execution of any program generates (through its instrumentation) a sequence of operations which satisfies the semantics of the data structure. This check is performed by an *observer*, which monitors the sequence of operations that is announced by the instrumentation, and report when it violates the semantics of the set data structure.



$\mathtt{add}(x, \mathtt{true})$
s_0
s_1
$\mathtt{rmv}(x, \mathtt{true})$
$\mathtt{add}(x, \mathtt{false})$
$\mathtt{rmv}(x, \mathtt{true})$
$\mathtt{ctn}(x, \mathtt{true})$
$\mathtt{add}(x, \mathtt{true})$
$\mathtt{rmv}(x, \mathtt{false})$
$\mathtt{ctn}(x, \mathtt{false})$
s_2

Fig. 4: Set observer.

Formally, an observer $\mathcal{O}$ is a tuple $\langle S^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{init}}, \mathtt{X}^{\mathcal{O}}, \Delta^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{acc}} \rangle$ where $S^{\mathcal{O}}$ is a finite set of *observer states* including the *initial state* $s^{\mathcal{O}}_{\mathtt{init}}$ and the *accepting state* $s^{\mathcal{O}}_{\mathtt{acc}}$, a finite set $\mathtt{X}^{\mathcal{O}}$ of *registers*, and $\Delta^{\mathcal{O}}$ is a finite set of *transitions*. Transitions are of the form $\langle s_1, \mathtt{m}(d^{in}, d^{out}), s_2 \rangle$, $\mathtt{m} \in \mathbb{M}$ is a method name and x^{in} and x^{out} are either registers or constants, i.e., transitions are labeled by operations whose input or output data may be parameterized on registers. The observer processes a sequence of operations one operation at a time. If there is a transition, whose label (after replacing registers by their values) matches the operation, such a transition is performed. If there is no such transition, the observer remains in its current state. The observer accepts a sequence if it can be processed in such a way that an accepting state is reached. The observer is defined in such a way that it accepts precisely those sequences that are *not* in $[\![\mathtt{DS}]\!]$. Fig. 4 depicts an observer for the set data structure.

For stacks and queues, it was recently established [5, 20] that the set of linearizable histories, i.e., sequences of call and return actions, can be exactly specified by an observer. Thus, we check linearizability for queue and stack implementations without any user-supplied instrumentation, by using an observer which accepts precisely the complement of the set of linearizable histories. We illustrate this in Section XXX.

It remains to take care of the criterion that the call of pop already has a return value

3.4 Some Notation

We conclude this section by introducing some terminology and notation that will be use in subsequent sections. We assume a program $\mathcal{P}$ and a specification given by an observer $\mathcal{O}$. We assume that each thread $\mathtt{th}$ executes one method denoted $\mathtt{Method}\,(\mathtt{th})$.

Heaps. A *heap (state)* consists of a finite set $\mathbb{C}$ of cells, including the two special cells $\mathtt{null}$ and $\perp$ (dangling). We represent each named field of the cell type by a function from the set of nontrivial cells $\mathbb{C}^{-}\mathbb{C} \setminus \{\mathtt{null}, \perp\}$ to the appropriate domain, e.g., $\mathbb{C}$ in the case of pointer fields. Thus, the values of $\mathtt{next}$-fields is represented by a function $\mathtt{next} : \mathbb{C}^{-} \mapsto \mathbb{C}$.

Threads. A *local state* $\mathtt{loc}$ of a thread $\mathtt{th}$ defines the values of its local variables, including the program counter $\mathtt{pc}$ and the input parameter for the method executed by $\mathtt{th}$. The behavior of a thread $\mathtt{th}$ can be formalized as a labeled transition relation $\rightarrow_{\mathtt{th}}$ on pairs $\langle \mathtt{loc}, \mathcal{H} \rangle$ consisting of a local state $\mathtt{loc}$ and a heap $\mathcal{H}$. Transitions are normally unlabeled, except when the executed statement is annotated as a linearization point, in which case it is labeled by the corresponding operation (of form $\mathtt{m}(d^{in}, d^{out})$).

Programs. A *configuration* of a program $\mathcal{P}$ is a tuple $\langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle$ where $\mathtt{T}$ is a set of threads, $\mathcal{H}$ is a heap, and $\mathtt{LOC}$ maps each thread $\mathtt{th} \in \mathtt{T}$ to its local state $\mathtt{LOC}\,(\mathtt{th})$ The behavior of a program $\mathcal{P}$ can be formalized by a transition relation $\rightarrow_{\mathcal{P}}$ where each step corresponds to one move of a single thread. I.e., there is a transition of form $\langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle \xrightarrow{\lambda}_{\mathcal{P}} \langle \mathtt{T}, \mathtt{LOC}[\mathtt{th} \leftarrow \mathtt{loc}'], \mathcal{H}' \rangle$ whenever the transition relation $\rightarrow_{\mathtt{th}}$ has a transition $\langle \mathtt{loc}, \mathcal{H} \rangle \xrightarrow{\lambda}_{\mathtt{th}} \langle \mathtt{loc}', \mathcal{H}' \rangle$.

We use $\mathcal{S} = \mathcal{P} \otimes \mathcal{O}$ to denote the cross-product of a program $\mathcal{P}$ and observer $\mathcal{O}$. Transitions of $\mathcal{S}$ are of the form $\left\langle c^{\mathcal{P}}, s \right\rangle, \rightarrow_{\mathcal{S}}, \left\langle c^{\mathcal{P}'}, s' \right\rangle$, obtained from a transition $c^{\mathcal{P}} \xrightarrow{\lambda}_{\mathcal{P}} c^{\mathcal{P}'}$ of the program with some (possibly empty) label λ, where the observer makes a transition if it can perform a matching transition.

This construction reduces the problem of verifying that the represented concurrent algorithm is a linearizable implementation of a data structure, whose semantics is captured by the observer, to checking that $\mathcal{S}$ cannot reach a configuration $\left\langle c^{\mathcal{P}}, s^{\mathcal{O}}_{\mathtt{acc}} \right\rangle$ with an accepting observer state.

We may have to say that we check the trivial conditions that were checked by the monitor in SAS 16

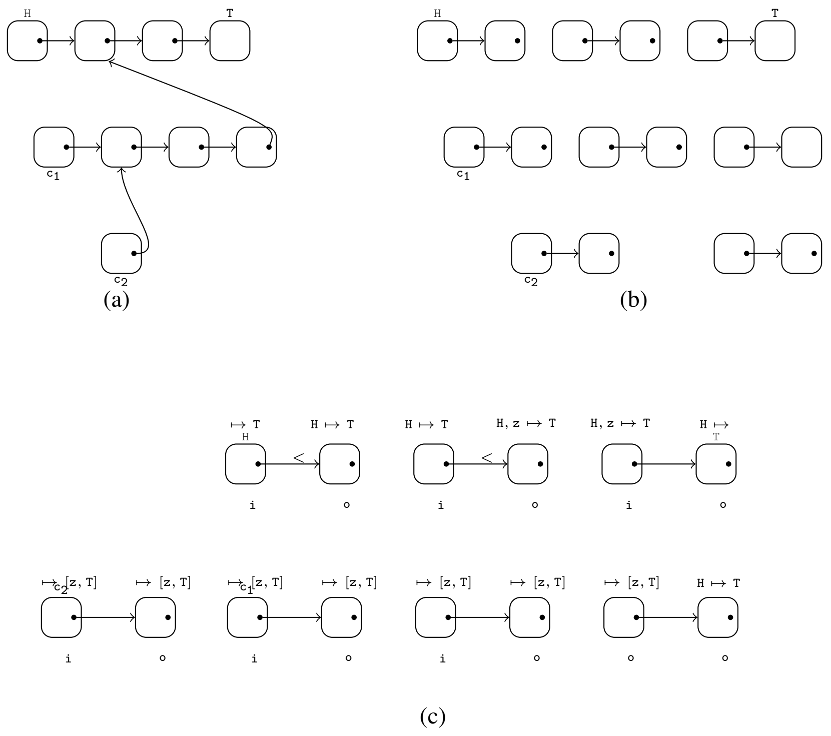


Fig. 5: Example of Fragment Abstraction

4 Fragment Abstraction: Singly Linked Lists

In the previous section, we reduced the problem of verifying linearizability to the problem of verifying that, in any execution of the cross-product of a program and an observer, the observer cannot reach an accepting state. In this section, we describe our technique for performing this verification. The main novel contribution is our fragment abstraction for representing the possible heap configurations of a concurrent program. In the following subsection, we describe our symbolic representation and verification technique, using fragment abstraction, for programs that operate on singly-linked lists (SLLs). This representation is also the basis for our representation for programs operating on skiplists, described in Subsection **??** and programs operating on arrays of SLLs, in Subsection **??**.

4.1 Symbolic Representations

In this subsection, we present our fragment abstraction for programs that operate on singly-linked lists. This means that each cell has exactly one pointer field, named `next`. In order to simplify the presentation in this section, we assume that each heap cell has at most one $\mathbb{D}$-field, named `data`. Our verification technique is based on a combination of a thread abstraction, a shape abstraction, and a data abstraction.

Assume a configuration $c^{\mathcal{S}}$ of $\mathcal{S} = \mathcal{P} \otimes \mathcal{O}$ and thread $\mathtt{th}$. We say that a heap cell $\mathbb{c}$ is *accessible* to a thread $\mathtt{th}$ in $c^{\mathcal{S}}$, if $\mathbb{c}$ is reachable (directly or via sequence of `next`-pointers) from a global pointer variable or local pointer variable of $\mathtt{th}$. For a pointer variable p, let a p-cell be the cell to which p points. For an observer register $\mathtt{x}_i$, define a $\mathtt{x}_i$*-cell* to be a heap cell whose `data` field has the same value as $\mathtt{x}_i$. [Maybe skip the following] A heap cell is *globally significant* if it is either a p-cell for a global pointer variable p, or a $\mathtt{x}_i$-cell for some observer register $\mathtt{x}_i$.

- Our *thread abstraction* adapts the thread-modular approach by representing only the view of a single, but arbitrary, thread $\mathtt{th}$. Such a view consists of (i) the local state of thread $\mathtt{th}$, (ii) the state of the observer, (iii) the part of the heap (including global variables) that is accessible to thread $\mathtt{th}$, and (iv) a predicate, which for each heap cell accessible to $\mathtt{th}$ says whether it is *private* to $\mathtt{th}$, i.e., not accessible to any other thread.
- We use a novel *shape abstraction*, which represents the part of the heap that is accessible to $\mathtt{th}$ by a set of *fragments*. A fragment consists of a pair of heap cells that are connected by a pointer. For each of these cells, the fragment represents (i) the values of its data fields, (ii) the pointer variables (either local to $\mathtt{th}$ or global) that point to it, and (iii) global pointer variables p and observer registers $\mathtt{x}_i$ such that the cell can either reach to or be reached from (by a chain of `next`-pointers) a p-cell or a $\mathtt{x}_i$-cell, respectively. A set of fragments represents the set of heaps in which each pair of pointer-connected nodes is represented by some fragment in the set. Thus, a set of fragments describes the set of heaps that can be formed by "piecing together" fragments, in a way that is locally consistent (connecting only fragments that agree on their common node) and globally consistent (respecting the reachability information wrp. to global pointer variables and observer registers).
- We apply a natural *data abstraction* to the local state of a thread, the state of the observer, and to each fragment. This abstraction represents small finite domains exactly, and abstracts a set of data values in $\mathbb{D}$ by their relative ordering.

Let us now describe our symbolic representation in concrete detail. Our data abstraction is conveniently represented by defining an abstract domain for each concrete domain of data values.

- For small concrete domains (including that of the program counter, and of the observer location), the abstract domain is the same as the concrete one.
- For locks, the abstract domain is $\{me, other, free\}$, mening that the lock is held by $\mathtt{th}$, held by some other thread, or is free, respectively.
- For the concrete domain $\mathbb{D}$ of data values, the abstract domain is the set of mappings from observer registers and local variables ranging over $\mathbb{D}$ to subsets of $\{<, =, >\}$.

An element in this abstract domain represents a concrete data value $\mathtt{d}$ if it maps each local variable and observer register with a value $\mathtt{d}' \in \mathbb{D}$ to a set which includes a relation $\sim$ such that $\mathtt{d} \sim \mathtt{d}'$.

In our shape abstration, we represent each fragment by a pair of *tags*. A *tag* is a tuple $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, where

- $\mathtt{dabs}$ is a mapping from non-pointer fields to their corresponding abstract domains,
- $\mathtt{pvars}$ is a set of (global or local) pointer variables,
- $\mathtt{reachfrom}$ and $\mathtt{reachto}$ are sets of global pointer variables and observer registers,
- $\mathtt{private}$ is a boolean value.

For a heap cell $\mathbb{c}$ that is accessible to thread $\mathtt{th}$ in a configuration $c^{\mathcal{S}}$, and a tag $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, we write $\mathbb{c} \lhd_{\mathtt{th}}^{c^{\mathcal{S}}} \mathtt{tag}$ to denote that

- $\mathtt{dabs}$ is an abstraction of the concrete values of the non-pointer fields of $\mathbb{c}$,
- $\mathtt{pvars}$ is the set of pointer variables (global or local to $\mathtt{th}$) that point to $\mathbb{c}$,
- $\mathtt{reachfrom}$ is the set of (i) global pointer variables from which $\mathbb{c}$ is reachable via a (possibly empty) sequence of $\mathtt{next}$ pointers, and (ii) observer registers $\mathtt{x}_i$ such that $\mathbb{c}$ is reachable from some $\mathtt{x}_i$-cell.
- $\mathtt{reachto}$ is the set of (i) global pointer variables pointing to a cell that is reachable from $\mathbb{c}$, and (ii) observer registers $\mathtt{x}_i$ such that some $\mathtt{x}_i$-cell is reachable from $\mathbb{c}$.
- $\mathtt{private}$ is $\mathtt{true}$ only if $\mathbb{c}$ is private to $\mathtt{th}$ in $c^{\mathcal{S}}$ [Should we add: "is not accessible to any other thread than $\mathtt{th}$"]

Definition 1 (SLL-fragment). *An* SLL-fragment $\mathtt{v}$ *(or just fragment) is a triple of form* $\langle \mathtt{v.i}, \mathtt{v.o}, \mathtt{v.}\phi \rangle$*, of form* $\langle \mathtt{v.i}, \mathtt{null} \rangle$*, or of form* $\langle \mathtt{v.i}, \bot \rangle$*, where* $\mathtt{v.i}$ *and* $\mathtt{v.o}$ *are tags and* $\mathtt{v.}\phi$ *is a subset of* $\{<, =, >\}$.

For a cell $\mathbb{c}$ which is accessible to thread $\mathtt{th}$, and a fragment $\mathtt{v}$ of form $\langle \mathtt{v.i}, \mathtt{v.o}, \mathtt{v.}\phi \rangle$, let $\mathbb{c} \lhd_{\mathtt{th}}^{c^{\mathcal{S}}} \mathtt{v}$ denote that the $\mathtt{next}$ field of $\mathbb{c}$ points to a cell $\mathbb{c}'$ such that (i) $\mathbb{c} \lhd_{\mathtt{th}}^{c^{\mathcal{S}}} \mathtt{v.i}$, (ii) $\mathbb{c}' \lhd_{\mathtt{th}}^{c^{\mathcal{S}}} \mathtt{v.o}$, and (iii) $\mathbb{c}.\mathtt{data} \sim \mathbb{c}'.\mathtt{data}$ for some $\sim \in \mathtt{v.}\phi$. For a fragment of form $\mathtt{v} = \langle \mathtt{v.i}, \mathtt{null} \rangle$, let $\mathbb{c} \lhd \mathtt{v}$ denote that $\mathbb{c} \lhd \mathtt{v.i}$ and $\mathtt{next}(\mathbb{c}) = \mathtt{null}$. Define $\mathbb{c} \lhd \mathtt{v}$ for $\mathtt{v}$ of form $\langle \mathtt{v.i}, \bot \rangle$ analogously.

Let V be a set of fragments. A global configuration $c^{\mathcal{S}}$ satisfies a set V of fragments wrp. to $\mathtt{th}$, denoted $c^{\mathcal{S}} \models_{\mathtt{th}}^{heap} V$, if for any cell $\mathbb{c}$ that is accessible to $\mathtt{th}$, there is a fragment $\mathtt{v} \in V$ such that $\mathbb{c} \lhd_{\mathtt{th}}^{c^{\mathcal{S}}} \mathtt{v}$.

We are now ready to define our abstract symbolic representation, which is defined as a partial mapping from combinations of abstract observer and local thread states to sets of fragments.

Define a *local symbolic configuration* as a mapping from local variables (including the program counter) to their corresponding abstract domains. We let $c^{\mathcal{S}} \models_{\mathtt{th}}^{loc} \sigma$ denote that in the global configuration $c^{\mathcal{S}}$, the local configuration of thread $\mathtt{th}$ satisfies the local symbolic configuration σ, defined in the natural way. For an observer location s, we let $c^{\mathcal{S}} \models_{\mathtt{th}}^{loc} \langle \sigma, s \rangle$ denote that $c^{\mathcal{S}} \models_{\mathtt{th}}^{loc} \sigma$ and that the observer location of $c^{\mathcal{S}}$ is s.

Definition 2. *A symbolic representation Ψ is a partial mapping from pairs of local symbolic configurations and observer locations to sets of fragments. A system* *configuration $c^{\mathcal{S}}$ satisfies a symbolic representation Ψ, denoted $c^{\mathcal{S}} \,\mathrm{sat}\, \Psi$, if for each thread* `th`*, the domain of Ψ contains a pair $\langle \sigma, s \rangle$ such that (i) $c^{\mathcal{S}} \models^{loc}_{\mathtt{th}} \langle \sigma, s \rangle$, and (ii) $c^{\mathcal{S}} \models^{heap}_{\mathtt{th}} \Psi(\langle \sigma, s \rangle)$.*

Do we need an example here?

Example: Let us show an example of how a singly linked list (SLL) is split into a set of fragments. Fig. 5(a) shows an example of a concrete shape of a singly linked list. Each cell contains the values of `val`, `mark`, and `lock` from top to bottom, where ✓ denotes `true`, and ✗ denotes `false` (or `free` for `lock`) and the value of `val` is denoted by a pair of the observer register `z` and a subset of $\{<, =, >\}$. There are two threads 1 and 2 with two local variables $\mathtt{c}_1$ and $\mathtt{c}_2$. There are two global variables `H` and `T` pointing to the head and tail of the list. The observer register `z` has value 8. Fig. 5(b) shows the result of splitting the list in Fig. 5(a) into fragments where each heap fragment is a small list of two nodes. Fig. 5(c) shows the abstraction of fragments in Fig. 5(b) where for each cell $\mathbb{c}$, the value of val is abstracted to a subset in $\{< \mathtt{z}, = \mathtt{z}, > \mathtt{z}\}$, the reachability relation between $\mathbb{c}$ and global variables and observer registers is abstracted to the predicate $\mathtt{X} \mapsto \mathtt{Y}$ where `X`, `Y` are sets of global variables and observer registers, `i` or `o` states that $\mathbb{c}$ is an input or output cell.

We ust always say that we consider the cross-product of a program and an observer

Symbolic Postcondition Computation In the verification, we must compute an invariant in the form of a symbolic representation which is satisfied by all reachable program configurations. Such an invariant is obtained by an abstract interpretation-based fixpoint procedure, which starts from a representation of the set of initial configurations, and thereafter repeatedly performs postcondition computations that extend the symbolic representation by the effect of any execution step of the program, until convergence. In this subsection, we describe the symbolic postcondition computation.

Polish the inclusion of the observer!

The symbolic postcondition computation must ensure that the symbolic representation of the reachable configurations of a program is closed under execution of a statement by some thread. So, assume a symbolic representation Ψ. The symbolic postcondition operation on Ψ produces an extension Ψ' of Ψ, such that whenever $c^{\mathcal{S}}$ is a configuration with $c^{\mathcal{S}} \,\mathrm{sat}\, \Psi$, then any step from $c^{\mathcal{S}}$ must lead to a configuration $c^{\mathcal{S}'}$ with $c^{\mathcal{S}'} \,\mathrm{sat}\, \Psi'$. Let `th` be an arbitrary thread. Then $c^{\mathcal{S}} \,\mathrm{sat}\, \Psi$ implies that $Dom(\Psi)$ contains a pair $\langle \sigma, s \rangle$ such that $c^{\mathcal{S}} \models^{loc}_{\mathtt{th}} \langle \sigma, s \rangle$ and $c^{\mathcal{S}} \models^{heap}_{\mathtt{th}} \Psi(\langle \sigma, s \rangle)$. Our symbolic postcondition computation must ensure that $Dom(\Psi')$ contains a pair $\langle \sigma', s' \rangle$ such that $c^{\mathcal{S}'} \models^{loc}_{\mathtt{th}} \langle \sigma', s' \rangle$ and $c^{\mathcal{S}'} \models^{heap}_{\mathtt{th}} \Psi(\langle \sigma', s' \rangle)$. In the thread-modular approach, there are two cases to consider, depending on which thread causes the step from $c^{\mathcal{S}}$ to $c^{\mathcal{S}'}$.

- *Local Steps:* The step is caused by `th` itself executing a statement which may change its local state, the location of the observer, and the state of the heap. In this case, we first compute a local symbolic configuration σ' and observer location

s', and a set V' of fragments such that $c^{\mathcal{S}'} \models^{loc}_{\mathtt{th}} \langle \sigma', s' \rangle$ and $c^{\mathcal{S}'} \models^{heap}_{\mathtt{th}} V'$, and then (if necessary) extend Ψ so that $\langle \sigma', s' \rangle \in Dom(\Psi)$ and $V' \subseteq \Psi(\langle \sigma', s' \rangle)$.

– *Interference Steps:* The step is caseud by Another thread $\mathtt{th}_2$, which may change the location of the observer (to s') and the state of the heap. By assumption, there is a local symbolic configuration σ_2 with $\langle \sigma_2, s \rangle \in Dom(\Psi)$ such that $c^{\mathcal{S}} \models_{\mathtt{th}_2} \langle \sigma_2, s, \Psi(\sigma_2) \rangle$. We then compute a set V' of fragments such that the resulting configuration $c^{\mathcal{S}'}$ satisfies $c^{\mathcal{S}'} \models^{heap}_{\mathtt{th}} V'$ and make sure that $V' \in \Psi(\langle \sigma, s' \rangle)$. To do this, we first combine the the local symbolic configurations σ and σ_2 and the sets of fragments $\Psi(\sigma)$ and $\Psi(\sigma_2)$, using an operation called *intersection*, into a joint local symbolic configuration of th and $\mathtt{th}_2$ and a set $V_{1,2}$ of fragments that represents the cells accessible to either th or $\mathtt{th}_2$. We thereafter symbolically compute the postcondition of the statement execution by $\mathtt{th}_2$, in the same was as for local steps, and finally project the set of resulting fragments back onto th in the natural way, to obtain V'.

In the following, we first describe the symbolic postcondition computation for local steps, and thereafter the intersection operation.

Symbolic Postcondition Computation for Local Steps Let th be an arbitrary thread, and assume that $\langle \sigma, s \rangle \in Dom(\Psi)$. For each statement that th can execute in a configuration c with $c^{\mathcal{S}} \models^{loc}_{\mathtt{th}} \langle \sigma, s \rangle$ and $c^{\mathcal{S}} \models^{heap}_{\mathtt{th}} \Psi(\langle \sigma, s \rangle)$, we must compute a local symbolic configuration σ', a new observer location s' and a set V' of fragments such that such that the resulting configuration $c^{\mathcal{S}'}$ satisfies $c^{\mathcal{S}'} \models^{loc}_{\mathtt{th}} \langle \sigma', s' \rangle$ and $c^{\mathcal{S}'} \models^{heap}_{\mathtt{th}} V'$. This computation has do be done differently for each statement. For statements that do not affect the heap or pointer variables, this computation is standard, and affects only the local symbolic configuration, the observer location, and the dabs component of tags. We therefore here describe how to compute the effect of statements that update pointer variables or pointer fields of heap cells, since these are the most interesting cases.

Let us fist consider a statement of form g := p, which assigns the value of a local pointer variable p to a global pointer variable g. It is easy to update the pvars field of tags to contain the variable g if and only if it contained the variable p before the statement. The difficulty is to update the reachability information provided by the fields reachfrom and reachto in each tag of a fragment, in particular whether g should be in such a set after the statement (if p would have been a global variable, the update would simply reflect that g and p are now aliases, just as in the update of pvars). In order to compute the postcondition with sufficient precision, we therefore investigate whether the set $\Psi(\langle \sigma, s \rangle)$ of fragments allows to form a heap in which a p-cell can reach or be reached from each tag of a fragment. It also checks whether a heap can be formed in which a p-cell can not reach or be reached from such tags. For each case that succeeds, the set V' will contain fragment with corresponding content of reachto and reachfrom fields.

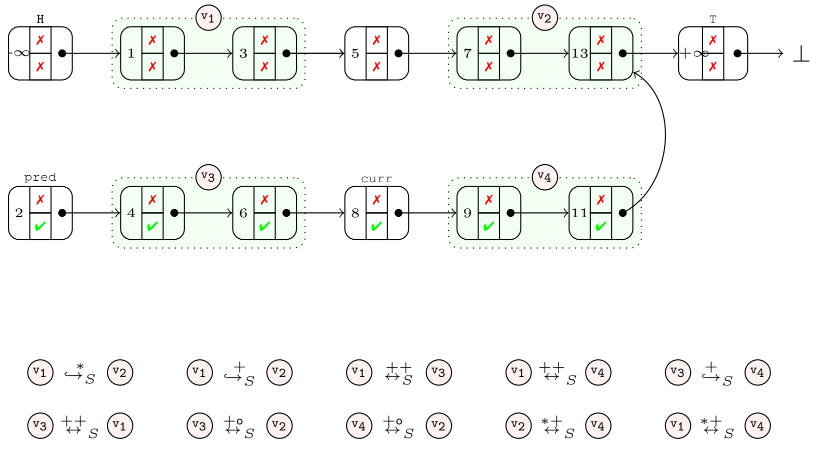
Our actual postcondition computation performs this investigation in a systematic manner, by computing a set of transitive closure-like relations between fragments. First, define two tags tag and tag$'$ to be *consistent* if the concretizations of their dabs-fields overlap, and if the other fields agree. Thus, tag and tag$'$ are consistent if there can exist a cell $\mathbb{c}$ accessible to $\mathtt{th}$ in some heap, with $\mathbb{c} \lhd^{c^S}_{\mathtt{th}}$ $\mathtt{tag}$ and $\mathbb{c} \lhd^{c^S}_{\mathtt{th}}$ $\mathtt{tag}'$. Then, for two fragments $\mathtt{v}_1$ and $\mathtt{v}_2$ in a set V of fragments,

- let $\mathtt{v}_1 \hookrightarrow_V \mathtt{v}_2$ denote that $\mathtt{v}_1.\mathtt{o}$ and $\mathtt{v}_2.\mathtt{i}$ are consistent, and
- let $\mathtt{v}_1 \leftrightarrow_V \mathtt{v}_2$ denote that $\mathtt{v}_1.\mathtt{o} = \mathtt{v}_2.\mathtt{o}$ are consistent, and that either $\mathtt{v}_1.\mathtt{i}.\mathtt{pvars} \cap \mathtt{v}_2.\mathtt{i}.\mathtt{pvars} = \emptyset$ or that the global variables in $\mathtt{v}_1.\mathtt{i}.\mathtt{reachfrom}$ are disjoint from those in $\mathtt{v}_2.\mathtt{i}.\mathtt{reachfrom}$. [Question: Is this correct?]

Intuitively, $\mathtt{v}_1 \hookrightarrow_V \mathtt{v}_2$ denotes that it is possible that $\mathtt{next}(\mathbb{c}_1) = \mathbb{c}_2$ for some cells with $\mathbb{c}_1 \lhd \mathtt{v}_1$ and $\mathbb{c}_2 \lhd \mathtt{v}_2$. Intuitively, $\mathtt{v}_1 \leftrightarrow_V \mathtt{v}_2$ denotes that it is possible that $\mathtt{next}(\mathbb{c}_1) = \mathtt{next}(\mathbb{c}_2)$. for different cells $\mathbb{c}_1$ and $\mathbb{c}_2$ with $\mathbb{c}_1 \lhd \mathtt{v}_1$ and $\mathbb{c}_2 \lhd \mathtt{v}_2$. Note that the above definitions also work when some view(s) contain $\mathtt{null}$ or $\perp$. We use these relations to define several derived relations:

- $\overset{+}{\hookrightarrow}_V$ denotes the transitive closure and $\overset{*}{\hookrightarrow}_V$ the reflexive transitive closure of $\hookrightarrow_V$,
- $\mathtt{v}_1 \overset{**}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{*}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{*+}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{*\circ}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$,
- $\mathtt{v}_1 \overset{++}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{+*}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{*}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{+\circ}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$.

We say that $\mathtt{v}_1$ and $\mathtt{v}_2$ are *compatible* if $\mathtt{v}_x \overset{*}{\hookrightarrow} \mathtt{v}_y$ or $\mathtt{v}_y \overset{*}{\hookrightarrow} \mathtt{v}_x$ or $\mathtt{v}_x \overset{**}{\leftrightarrow} \mathtt{v}_y$. Intuitively, if $\mathtt{v}_1$ and $\mathtt{v}_2$ are satisfied by two cells in the same heap state, then they must be compatible. [Do we need to update the following example?] Figure 6 illustrates the above relations


H
$-\infty$
v1
1
3
5
v2
7
13
T
$+\infty$
$\perp$
pred
2
v3
4
6
curr
8
v4
9
11
$\mathtt{v}_1 \overset{*}{\hookrightarrow}_S \mathtt{v}_2$
$\mathtt{v}_1 \overset{+}{\hookrightarrow}_S \mathtt{v}_2$
$\mathtt{v}_1 \overset{++}{\leftrightarrow}_S \mathtt{v}_3$
$\mathtt{v}_1 \overset{++}{\leftrightarrow}_S \mathtt{v}_4$
$\mathtt{v}_3 \overset{+}{\hookrightarrow}_S \mathtt{v}_4$
$\mathtt{v}_3 \overset{++}{\leftrightarrow}_S \mathtt{v}_1$
$\mathtt{v}_3 \overset{+\circ}{\leftrightarrow}_S \mathtt{v}_2$
$\mathtt{v}_4 \overset{+\circ}{\leftrightarrow}_S \mathtt{v}_2$
$\mathtt{v}_2 \overset{*+}{\leftrightarrow}_S \mathtt{v}_4$
$\mathtt{v}_1 \overset{*+}{\leftrightarrow}_S \mathtt{v}_4$
Fig. 6: Illustration of some transitive closure-like relations between fragments

for a heap state with 13 heap cells. The figure shows 4 fragments that are satisfied by heap cells, as denoted by green boxes, and how the relationship between heap cells is reflect by relations between the corresponding fragments.

We can now describe how to perform the symbolic postcondition computations for statements that assign to a pointer variable.

Consider a statement of form $\mathtt{x} := \mathtt{y}$, where x and y are global or local (to thread th) pointer variables. We must compute a set V' of fragments which are satisfied by the configuration after the statement, implying that any cell $\mathbb{c}$ which is accessible to th after the statement satisfies some fragment in V'. The cell $\mathbb{c}$ must satisfy some fragment v in V before the statement, and must be in the same heap state as the cell pointed to by y. This implies that v must be compatible with some fragment $\mathtt{v}_y \in V$ such that $\mathtt{y} \in \mathtt{v}_y.\mathtt{i.pvars}$. This means that we can make a case analysis on the possible relationships between v and any such $\mathtt{v}_y$. Thus, for each fragment $\mathtt{v}_y \in V$ such that $\mathtt{y} \in \mathtt{v}_y.\mathtt{i.pvars}$ we let V' contain the fragments obtained by the following transformations on fragments in V.

1. First, for the fragment $\mathtt{v}_y$ itself, we let V' contain $\mathtt{v}'$, which is the same as $\mathtt{v}_y$, except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v}_y.\mathtt{i.pvars} \cup \{x\}$ and $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$

   and furthermore, if x is a global variable, then
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$ and $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \cup \{x\}$,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$ and $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$.
2. for each fragment v with $\mathtt{v} \hookrightarrow_V \mathtt{v}_y$, let V' contain $\mathtt{v}'$ which is same as v except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
   - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$ if x is a global variable,
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$ if x is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$ if x is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$ if x is a global variable,
3. We perform analogous inclusions for fragments v with $\mathtt{v} \overset{+}{\hookrightarrow}_V \mathtt{v}_y$, $\mathtt{v}_y \overset{*}{\hookrightarrow}_V \mathtt{v}$, $\mathtt{v}_y \overset{*+}{\leftrightarrow}_V \mathtt{v}$, and $\mathtt{v}_y \overset{*\circ}{\leftrightarrow}_V \mathtt{v}$. For space reasons, we show only the case of $\mathtt{v}_y \overset{*+}{\leftrightarrow}_V \mathtt{v}$, in which case we let V' contain $\mathtt{v}'$ which is same as v except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$ if x is a global variable,
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$ if x is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \setminus \{x\}$ if x is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$ if x is a global variable,

The statement $\mathtt{x} := \mathtt{y.next}$ is handled rather similarly to the preceding case. Let us therefore describe the computation for statements of the form $\mathtt{x.next} := \mathtt{y}$. This is the most difficult statement, since it is a destructive update of the heap. The statement affects reachability relations for both x and y. This means that we can make a case analysis on how a fragment in V is related to some pair of compatible fragments $\mathtt{v}_x$, $\mathtt{v}_y$ in V such that $\mathtt{x} \in \mathtt{v}_x.\mathtt{i.pvars}$, $\mathtt{y} \in \mathtt{v}_\mathtt{y}.\mathtt{i.pvars}$. Thus, for each pair of compatible fragments $\mathtt{v}_x$, $\mathtt{v}_y$ in V such that $\mathtt{x} \in \mathtt{v}_x.\mathtt{i.pvars}$, $\mathtt{y} \in \mathtt{v}_\mathtt{y}.\mathtt{i.pvars}$, we let V' contain the fragments obtained by the following transformations on fragments in V.

1. First, let V' containt a new fragment $\mathtt{v}_{new}$ of form $\langle \mathtt{v}_{new}.\mathtt{i}, \mathtt{v}_{new}.\mathtt{o}, \mathtt{v}_{new}.\phi \rangle$ $\mathtt{v}_{new}.\mathtt{i.tag} = \mathtt{v}_x.\mathtt{i.tag}$ and $\mathtt{v}_{new}.\mathtt{o.tag} = \mathtt{v}_y.\mathtt{i.tag}$ except that $\mathtt{v}_{new}.\mathtt{o.reachfrom} = \mathtt{v}_y.\mathtt{i.reachfrom} \cup \mathtt{v}_x.\mathtt{i.reachfrom}$, and $\mathtt{v}_{new}.\phi = \{<, =, >\}$. Thereafter, we add all possible fragments that can result from a transformation of some fragment v which is in v. This is done by an exhaustive case analysis on the possible relationship between v, $\mathtt{v}_x$ and $\mathtt{v}_y$. Let us consider an interesting case, in which $\mathtt{v}_x \overset{*}{\hookrightarrow}_V \mathtt{v}$ and either $\mathtt{v} \overset{+}{\hookrightarrow}_V \mathtt{v}_y$ or $\mathtt{v}_y \overset{*+}{\leftrightarrow} \mathtt{v}$. In this case,
   (a) for each subset regset of observer registers in $\mathtt{v.i.reachfrom} \cap \mathtt{v_x.i.reachfrom}$, we first create a fragment $\mathtt{v}'$ which is same as v, except that $\mathtt{v}'.\mathtt{i.reachfrom} = (\mathtt{v.i.reachfrom} \setminus \mathtt{v_x.i.reachfrom}) \cup \mathtt{regset}$.
   (b) Thereafter, for each set $\mathtt{regset}'$ of observer registers in $\mathtt{v}'.\mathtt{o.reachfrom} \cap \mathtt{v}_x.\mathtt{i.reachfrom}$, we let V' contain a fragment $\mathtt{v}''$ which is same as $\mathtt{v}'$, except that $\mathtt{v}''.\mathtt{o.reachfrom} = (\mathtt{v}'.\mathtt{o.reachfrom} \setminus \mathtt{v}_x.\mathtt{i.reachfrom}) \cup \mathtt{regset}'$.

We should include an argument why the last case above is correct. Quy, could you produce one?

Symbolic Postcondition Computation for Interference Steps. The key step in this computation is to form the intersection of two sets of fragments V_1 and V_2, such that for the configuration c we have $c \models^{heap}_{\mathtt{th}_i} V_i$ for $i = 1, 2$. In order to distinguish between local variables of $\mathtt{th}_1$ and $\mathtt{th}_2$, we assume that local variable x of thread $\mathtt{th}_i$ is named as $\mathtt{x}[\mathtt{i}]$. We must compute a set V which for each heap cell accessible to either $\mathtt{th}_1$ or $\mathtt{th}_2$, the set V must contain a fragment v with $\mathbb{c} \lhd^{c^S}_{1,2} \mathtt{v}$. [This notation to be defined] There are here two possibilities.

- If $\mathbb{c}$ is accessible to both $\mathtt{th}_1$ and $\mathtt{th}_2$, then there are fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$ such that $\mathbb{c} \lhd^{c^S}_1 \mathtt{v}_1$ and $\mathbb{c} \lhd^{c^S}_2 \mathtt{v}_2$. We use the notation $\mathtt{v}_1 \sqcap \mathtt{v}_2$ to denote a set of views such that whenever $\mathbb{c} \lhd^{c^S}_1 \mathtt{v}_1$ and $\mathbb{c} \lhd^{c^S}_2 \mathtt{v}_2$ then $\mathbb{c} \lhd^{c^S}_{1,2} \mathtt{v}$ for some $\mathtt{v} \in (\mathtt{v}_1 \sqcap \mathtt{v}_2)$.
- If $\mathbb{c}$ is accessible to only one of $\mathtt{th}_1$ and $\mathtt{th}_2$, say $\mathtt{th}_1$, then V should contain some fragment $\mathtt{v}_1 \in V_1$ with $\mathbb{c} \lhd^{c^S}_{1,2} \mathtt{v}_1$ [Check that we need not change $\mathtt{v}_1$]

For a fragment v, define v.i.greachfrom as the set of global variables in v.i.reachfrom. Define v.i.greachto, v.o.greachfrom, v.o.greachto, v.i.gpvars, and v.o.gpvars analogously. Define v.o.gtag as the tuple $\langle$v.o.gpvars, v.o.dabs, v.o.greachfrom, v.o.greachto, v.o.private$\rangle$.

Question to Quy: You have also used the notation v.i.gdata. Will you need it, and if so what does it mean?

[ANSWER of Quy: Because in the data, I add data constraint between data fields and local data variable. When we do intersection, we do not need to care about this constraint because its local constraint]

Let us now describe how to compute $\mathtt{v}_1 \sqcap \mathtt{v}_2$ for two views $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$. Firstly, we consider the case where both $\mathtt{v}_1$ and $\mathtt{v}_2$ have size 2. Let us consider some different cases. They all take into account the observation that if a cell $\mathbb{c}$ satisfies $\mathbb{c} \lhd^{c^S}_1 \mathtt{v}_1$ and $\mathbb{c} \lhd^{c^S}_2 \mathtt{v}_2$, then the information about global variables in $\mathtt{v}_1$ and $\mathtt{v}_2$ must coincide.

- if $\mathtt{v}_1.\mathtt{i.greachfrom} \neq \emptyset$ and $\mathtt{v}_2.\mathtt{i.greachfrom} \neq \emptyset$ then the global information in $\mathtt{v}_1$ and $\mathtt{v}_2$ must coincide. We hence obtain:
  - if $\mathtt{v}_1.\mathtt{i.gtag} = \mathtt{v}_2.\mathtt{i.gtag}$ and $\mathtt{v}_1.\mathtt{o.gtag} = \mathtt{v}_2.\mathtt{o.gtag}$ then $\mathtt{v}_1 \sqcap v_2 = \{\mathtt{v}_{12}\}$ where $\mathtt{v}_{12}$ is identical to $\mathtt{v}_1$ except that
    - $\mathtt{v}_{12}.\mathtt{i.pvars} = \mathtt{v}_1.\mathtt{i.pvars} \cup \mathtt{v}_2.\mathtt{i.pvars}$
    - $\mathtt{v}_{12}.\mathtt{o.pvars} = \mathtt{v}_1.\mathtt{o.pvars} \cup \mathtt{v}_2.\mathtt{o.pvars}$
    - $\mathtt{v}_{12}.\mathtt{i.reachfrom} = \mathtt{v}_1.\mathtt{i.reachfrom} \cup \mathtt{v}_2.\mathtt{i.reachfrom}$
    - $\mathtt{v}_{12}.\mathtt{o.reachfrom} = \mathtt{v}_1.\mathtt{o.reachfrom} \cup \mathtt{v}_2.\mathtt{o.reachfrom}$

      Question to Quy: Why do this union only for `pvars` and `reachfrom`, and not for `reachto` and `dabs`?

      [ANSWER of Quy: we fixed this in the disscussion]

    Question to Quy: Should we not have an "else" here, with $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \emptyset$?

    [ANSWER of Quy: we fixed this in the disscussion]
- if $\mathtt{v}_1.\mathtt{i.greachfrom} = \emptyset$, $\mathtt{v}_2.\mathtt{i.greachfrom} = \emptyset$, $\mathtt{v}_1.\mathtt{o.greachfrom} \neq \emptyset$ and $\mathtt{v}_2.\mathtt{o.greachfrom} \neq \emptyset$ then
  - if $\mathtt{v}_1.\mathtt{o.gtag} = \mathtt{v}_2.\mathtt{o.gtag}$, $\mathtt{v}_1.\mathtt{i.private} = \mathtt{false}$ and $\mathtt{v}_2.\mathtt{i.private} = \mathtt{false}$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}'_1, \mathtt{v}'_2, \mathtt{v}_{12}\}$ where
    - $\mathtt{v}'_1$ is same as $\mathtt{v}_1$ except that
      - $\mathtt{v}'_1.\mathtt{o.pvars} = \mathtt{v}_1.\mathtt{o.pvars} \cup \mathtt{v}_2.\mathtt{o.pvars}$
      - $\mathtt{v}'_1.\mathtt{o.reachfrom} = \mathtt{v}_1.\mathtt{o.reachfrom} \cup \mathtt{v}_2.\mathtt{o.reachfrom}$
    - $\mathtt{v}'_2$ is same as $\mathtt{v}_2$ except that
      - $\mathtt{v}'_2.\mathtt{o.pvars} = \mathtt{v}_1.\mathtt{o.pvars} \cup \mathtt{v}_2.\mathtt{o.pvars}$
      - $\mathtt{v}'_2.\mathtt{o.reachfrom} = \mathtt{v}_1.\mathtt{o.reachfrom} \cup \mathtt{v}_2.\mathtt{o.reachfrom}$
    - $\mathtt{v}_{12}$ is as in the previous case. [to Quy: I added this, is it correct?]
  - if $\mathtt{v}_1.\mathtt{o.gtag} = \mathtt{v}_2.\mathtt{o.gtag}$ and $\mathtt{v}_1.\mathtt{i.private} = \mathtt{true}$ or $\mathtt{v}_2.\mathtt{i.private} = \mathtt{true}$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}'_1, \mathtt{v}'_2\}$ where $\mathtt{v}'_1$ and $\mathtt{v}'_2$ are as above.
- if $\mathtt{v}_1.\mathtt{i.greachfrom} = \emptyset$, $\mathtt{v}_2.\mathtt{i.greachfrom} = \emptyset$, $\mathtt{v}_1.\mathtt{o.greachfrom} = \emptyset$ and $\mathtt{v}_2.\mathtt{o.greachfrom} = \emptyset$ then
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$, $\mathtt{v}_1.\mathtt{i.private} = false$, $\mathtt{v}_1.\mathtt{o.private} = false$, $\mathtt{v}_1.\mathtt{o.private} = false$ and $\mathtt{v}_2.\mathtt{o.private} = false$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}'_1, \mathtt{v}'_2, \mathtt{v}_{12}\}$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$, ($\mathtt{v}_1.\mathtt{i.private} = true$ or $\mathtt{v}_2.\mathtt{i.private} = true$) and $\mathtt{v}_1.\mathtt{o.private} = false$ and $\mathtt{v}_2.\mathtt{o.private} = false$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}'_1, \mathtt{v}'_2\}$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$ and ($\mathtt{v}_1.\mathtt{o.private} = true$ or $\mathtt{v}_1.\mathtt{o.private} = true$) then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2\}$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) \neq \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2\}$

5 Fragment Abstraction for Skip-Lists

Here goes a description of the fragment abstraction for skip lists

- In the fragment abstraction, `tag` is define exactly same as `tag` in SLL abstraction where `reachfrom` and `reachto` is defined based on the main level of skip-list. It means that we do not keep the reachability information in higher levels.

– Same as timestamp stacks and queues, in skip-list we keep the main level and abstract all the higher levels. It means that we do not distinguish the differences between high levels. Hence, we have two types of fragments including main level fragments and higher level fragments which are defined same as SLL fragments.

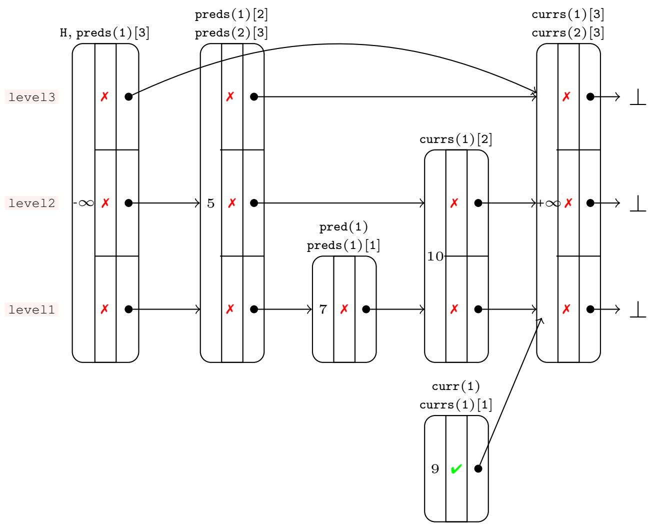


Fig. 7: A concrete shape of 3-level skipl-list with two threads

5.1 Abstract transformers for skip-lists

Let us show how to perform the abstract transformer for skip-list programs on the set of fragments V depending on the particular statement. We consider fragments of size 2 where $\mathtt{I_{inp}} = \{\mathtt{i}\}$, $\mathtt{I_{out}} = \{\mathtt{o}\}$, and $\mathtt{next(i)} = \mathtt{o}$ and fragments of size 1 where $\mathtt{I_{inp}} = \{\mathtt{i}\}$, $\mathtt{I_{out}} = \emptyset$, and $\mathtt{next(i)} = \mathtt{null}$ or $\mathtt{next(i)} = \bot$. For each fragment $\mathtt{v}$, let $\mathtt{v.level} \in \{1, 2\}$ be the level of $\mathtt{v}$.

Local Abstract Transformers: First, let us show the abstract transformer on the set of fragment V in the fragment of the concurrent thread. Let V_1 be set of fragments of level 1 in V, V_2 be set of fragments of level 2 in V. For each program statement, let V_{post} be the set of fragments after executing the statement. Let V_{post} be initialized as the empty set. Let $\mathtt{R}$ be the set of pairs of fragments. Intuitively, in each element in $\mathtt{R}$, the second fragment is the transformation of the first fragment. Let $\mathtt{R}$ be initialized as the empty set.

– $\mathtt{x} := \mathtt{y}$: The transformer is performed as follows: For each fragment $\mathtt{v_y} \in \mathtt{V_1}$ where $\mathtt{y} \in \mathtt{v_y.i.pvars}$,

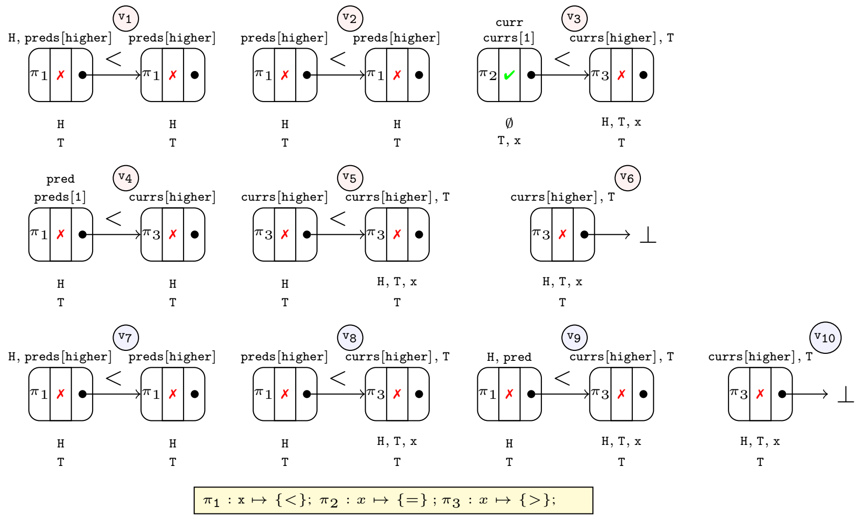


Fig. 8: skipl-list fragments [ANSWER of Quy: i am working with this figure]

1. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v} \hookrightarrow_{\mathtt{V}} \mathtt{v_y}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
   - if

     $gvarofx$ is a global variable
     * $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
2. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v} \overset{+}{\hookrightarrow}_{\mathtt{V}} \mathtt{v_y}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - if

     $gvarofx$ is a global variable
     * $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
3. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v_y} \overset{*}{\hookrightarrow}_{\mathtt{V}} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,

  - if
    $gvarofx$ is a global variable
      * $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{i}.\mathtt{reachfrom} \cup \{x\}$,
      * $\mathtt{v}'.\mathtt{i}.\mathtt{reachto} = \mathtt{v}.\mathtt{i}.\mathtt{reachto} \setminus \{x\}$,
      * $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{o}.\mathtt{reachfrom} \cup \{x\}$,
      * $\mathtt{v}'.\mathtt{o}.\mathtt{reachto} = \mathtt{v}.\mathtt{o}.\mathtt{reachto} \setminus \{x\}$,

  then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to V'
4. for each fragment $\mathtt{v}$ where $\mathtt{v}_\mathtt{y} \overset{*\dagger}{\leftrightarrow}_\mathtt{V} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i}.\mathtt{pvars} = \mathtt{v}.\mathtt{i}.\mathtt{pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{pvars} = \mathtt{v}.\mathtt{o}.\mathtt{pvars} \setminus \{x\}$,
   - if
     $gvarofx$ is a global variable
       * $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{i}.\mathtt{reachfrom} \setminus \{x\}$,
       * $\mathtt{v}'.\mathtt{i}.\mathtt{reachto} = \mathtt{v}.\mathtt{i}.\mathtt{reachto} \setminus \{x\}$,
       * $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{o}.\mathtt{reachfrom} \setminus \{x\}$,
       * $\mathtt{v}'.\mathtt{o}.\mathtt{reachto} = \mathtt{v}.\mathtt{o}.\mathtt{reachto} \setminus \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to V'
5. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v}_\mathtt{y} \overset{*\circ}{\leftrightarrow}_\mathtt{V} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i}.\mathtt{pvars} = \mathtt{v}.\mathtt{i}.\mathtt{pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{pvars} = \mathtt{v}.\mathtt{o}.\mathtt{pvars} \setminus \{x\}$,
   - if
     $gvarofx$ is a global variable
       * $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{i}.\mathtt{reachfrom} \setminus \{x\}$,
       * $\mathtt{v}'.\mathtt{i}.\mathtt{reachto} = \mathtt{v}.\mathtt{i}.\mathtt{reachto} \setminus \{x\}$,
       * $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{o}.\mathtt{reachfrom} \cup \{x\}$,
       * $\mathtt{v}'.\mathtt{o}.\mathtt{reachto} = \mathtt{v}.\mathtt{o}.\mathtt{reachto} \setminus \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to V'
6. create $\mathtt{v}'$ which is same as $\mathtt{v}_\mathtt{y}$ except that
   - $\mathtt{v}'.\mathtt{i}.\mathtt{pvars} = \mathtt{v}.\mathtt{i}.\mathtt{pvars} \cup \{x\}$,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{pvars} = \mathtt{v}.\mathtt{o}.\mathtt{pvars} \setminus \{x\}$,
   - if
     $gvarofx$ is a global variable
       * $\mathtt{v}'.\mathtt{i}.\mathtt{reachto} = \mathtt{v}.\mathtt{i}.\mathtt{reachto} \cup \{x\}$,
       * $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{i}.\mathtt{reachfrom} \cup \{x\}$,
       * $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{o}.\mathtt{reachfrom} \cup \{x\}$,
       * $\mathtt{v}'.\mathtt{o}.\mathtt{reachto} = \mathtt{v}.\mathtt{o}.\mathtt{reachto} \setminus \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to V'
7. for each fragment $\mathtt{v} \in \mathtt{V}_2$ we do as follows. For each $(\mathtt{v}_1, \mathtt{v}'_1)$, $(\mathtt{v}_2, \mathtt{v}'_2) \in \mathtt{R}$, for each pair of indices $\mathtt{i}_1$, $\mathtt{i}_2$ such that $\mathtt{i}_1 \in \{\mathtt{v}_1.\mathtt{i}, \mathtt{v}_1.\mathtt{o}\}$, $\mathtt{i}_2 \in \{\mathtt{v}_2.\mathtt{i}, \mathtt{v}_2.\mathtt{o}\}$, $\mathtt{tag}(\mathtt{v}, \mathtt{i}) = \mathtt{tag}(\mathtt{v}_1, \mathtt{i}_1)$, and $\mathtt{tag}(\mathtt{v}, \mathtt{o}) = \mathtt{tag}(\mathtt{v}_2, \mathtt{i}_2)$. Create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i}.\mathtt{pvars} = \mathtt{v}'_1.i_1.vars$,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{pvars} = \mathtt{v}'_2.i_2.vars$,
   - $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}'_1.i_1.reachfrom$,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}'_2.i_2.reachfrom$,

  - $\mathtt{v'.i.reachto} = \mathtt{v}'_1.i_1.reachto$,
  - $\mathtt{v'.o.reachto} = \mathtt{v}'_2.i_2.reachto$,

  then add $\mathtt{v}'$ to V_{post}.

- $\mathtt{x} := \mathtt{y.next1}$: The local abstract transformer is quite similar to the previous case with slightly differences. For each fragment $\mathtt{v_y} \in \mathtt{S}$ where $\mathtt{y} \in \mathtt{vars(i)}$,

  1. for each fragment $\mathtt{v} \in \mathtt{V_1}$ where $\mathtt{v} \overset{*}{\hookrightarrow}_{\mathtt{V}} \mathtt{v_y}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
     - if $\mathtt{x}$ is a global variable
       * $\mathtt{v'.i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
       * $\mathtt{v'.i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$,
       * $\mathtt{v'.o.reachfrom} = \mathtt{v.o.reachfrom} \setminus \{x\}$,
       * $\mathtt{v'.o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,

     then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
  2. for each fragment $\mathtt{v} \in \mathtt{V_1}$ where $\mathtt{v_y} \hookrightarrow_{\mathtt{V}} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ then
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \cup \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
     - if $\mathtt{x}$ is a global variable
       * $\mathtt{v'.i.reachfrom} = \mathtt{v.i.reachfrom} \cup \{x\}$,
       * $\mathtt{v'.o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,
       * $\mathtt{v'.o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
       * $\mathtt{v'.o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$,

     then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
  3. for each fragment $\mathtt{v} \in \mathtt{V_1}$ where $\mathtt{v_y} \leftrightarrow_{\mathtt{V}} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
     - if $\mathtt{x}$ is a global variable
       * $\mathtt{v'.i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
       * $\mathtt{v'.i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$,
       * $\mathtt{v'.o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
       * $\mathtt{v'.o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,

     then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
  4. for each fragment $\mathtt{v} \in \mathtt{V_1}$ where $\mathtt{v_y} \overset{+}{\hookrightarrow}_{\mathtt{V}} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
     - if $\mathtt{x}$ is a global variable
       * $\mathtt{v'.i.reachfrom} = \mathtt{v.i.reachfrom} \cup \{x\}$,
       * $\mathtt{v'.i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$,
       * $\mathtt{v'.o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
       * $\mathtt{v'.o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$,

     then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
  5. for each fragment $\mathtt{v} \in \mathtt{V_1}$ where $\mathtt{v_y} \overset{*\circ}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that

   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - if $\mathtt{x}$ is a global variable
     * $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
6. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v}_\mathtt{y} \overset{*+}{\leftrightarrow}_\mathtt{V} \mathtt{v}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\{$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - if $\mathtt{x}$ is a global variable
     * $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
7. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v} \overset{+\circ}{\leftrightarrow}_\mathtt{V} \mathtt{v}_\mathtt{y}$, create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - if $\mathtt{x}$ is a global variable
     * $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
8. create $\mathtt{v}'$ which is same as $\mathtt{v}_\mathtt{y}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
   - if $\mathtt{x}$ is a global variable
     * $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
     * $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
9. for each fragment $\mathtt{v} \in \mathtt{V}_2$ we do as follows. For each $(\mathtt{v}_1, \mathtt{v}_1')$, $(\mathtt{v}_2, \mathtt{v}_2') \in \mathtt{R}$, for each pair of indices $\mathtt{i}_1$, $\mathtt{i}_2$ such that $\mathtt{i}_1 \in \{\mathtt{v}_1.\mathtt{i}, \mathtt{v}_1.\mathtt{o}\}$, $\mathtt{i}_2 \in \{\mathtt{v}_2.\mathtt{i}, \mathtt{v}_2.\mathtt{o}\}$, $\mathtt{tag}(\mathtt{v}, \mathtt{i}) = \mathtt{tag}(\mathtt{v}_1, \mathtt{i}_1)$, and $\mathtt{tag}(\mathtt{v}, \mathtt{o}) = \mathtt{tag}(\mathtt{v}_2, \mathtt{i}_2)$. Create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v}_1'.i_1.vars$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v}_2'.i_2.vars$,
   - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v}_1'.i_1.reachfrom$,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v}_2'.i_2.reachfrom$,
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v}_1'.i_1.reachto$,

    - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v}_2'.i_2.reachto$,

  then add $\mathtt{v}'$ to V_{post}.

- $\mathtt{x.next1} := \mathtt{y}$: The local abstract transformer is performed by several steps as follows: For each pair of fragments $\mathtt{v}_x$, $\mathtt{v_y}$ in V_1 where $\mathtt{x} \in \mathtt{v_x.i.pvars}$, $\mathtt{y} \in \mathtt{v_y.i.pvars}$, and $\mathtt{v}_x \overset{*}{\hookrightarrow} \mathtt{v}_y$ or $\mathtt{v_y} \overset{*}{\hookrightarrow} \mathtt{v_x}$ or $\mathtt{v}_x \overset{**}{\leftrightarrow} \mathtt{v}_y$, let $\mathtt{R}_1$, $\mathtt{R}_2$, $\mathtt{R}_3$, $\mathtt{R}_4$, $\mathtt{R}_5$, $\mathtt{R}_6$ be initialized as $\mathtt{R}$,
  1. let $\mathtt{v_{new}}$ be the fragment of size 2 and of level 1 where $\mathtt{tag}(\mathtt{v_{new}}, \mathtt{i}) = \mathtt{tag}(\mathtt{v_x}, \mathtt{i})$ and $\mathtt{tag}(\mathtt{v_{new}}, \mathtt{o}) = \mathtt{tag}(\mathtt{v_y}, \mathtt{i})$ except that $\mathtt{v_{new}.o.reachfrom} = \mathtt{v_y.i.reachfrom} \cup \mathtt{v_x.i.reachfrom}$,
  2. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v} \overset{*}{\hookrightarrow}_{\mathtt{V}} \mathtt{v_x}$, we do as follows: For each subset $\mathtt{regset}$ of observer registers in $\mathtt{v.i.reachto} \cap \mathtt{v}_x\mathtt{.i.reachfrom}$
     - create $\mathtt{v}'$ which is same as $\mathtt{v}$, except that
       - $\mathtt{v}'.\mathtt{i.reachto} = (\mathtt{v.i.reachto} \cap \mathtt{v_x.i.reachfrom}) \cup \mathtt{reachtov_y.i} \cup \mathtt{regset}$.
       - $\mathtt{v}'.\mathtt{o.reachto} = (\mathtt{v.o.reachto} \cap \mathtt{v_x.i.reachfrom}) \cup \mathtt{reachtov_y.i} \cup \mathtt{regset}$.
     - add $\mathtt{v}'$ to V_{post}
     - add $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}_1$
  3. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v_y} \overset{*}{\hookrightarrow}_{\mathtt{V}} \mathtt{v}$ or $\mathtt{v_y} = \mathtt{v}$,
     - create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
       - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.i.reachfrom}$,
       - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.o.reachfrom}$,
     - add $\mathtt{v}'$ to V_{post},
     - add $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}_2$
  4. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v}_x \overset{**}{\leftrightarrow}_V v$ and either $\mathtt{v} \hookrightarrow_{\mathtt{V}} \mathtt{v_y}$ or $\mathtt{v_y} \overset{*o}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$,
     - create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.o.reachfrom}$,
     - add $\mathtt{v}'$ to V_{post},
     - add $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}_3$
  5. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v}_x \overset{*}{\hookrightarrow}_V v$ and either $\mathtt{v} \hookrightarrow_{\mathtt{V}} \mathtt{v_y}$ or $\mathtt{v_y} \overset{*o}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$,
     - create $\mathtt{v}'$ which is same as $\mathtt{v}$ then except that $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.o.reachfrom}$,
     - for each subset $\mathtt{regset}$ of observer registers in $\mathtt{v}'.\mathtt{i.reachfrom} \cap \mathtt{v}_x\mathtt{.i.reachfrom}$
       - create $\mathtt{v}''$ which is same as $\mathtt{v}'$, except that $\mathtt{v}''.\mathtt{i.reachfrom} = (\mathtt{v}'.\mathtt{i.reachfrom} \setminus \mathtt{v}_x\mathtt{.i.reachfrom}) \cup \mathtt{regset}$.
       - add $\mathtt{v}''$ to V_{post},
       - add $(\mathtt{v}, \mathtt{v}'')$ to $\mathtt{R}_4$
  6. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v}_x \overset{**}{\leftrightarrow}_V v$ and either $\mathtt{v} \overset{+}{\hookrightarrow}_{\mathtt{V}} \mathtt{v_y}$ or $\mathtt{v} \overset{*+}{\leftrightarrow}_{\mathtt{V}} \mathtt{v_y}$,
     - create $\mathtt{v}'$ which is same as $\mathtt{v}$
     - add $\mathtt{v}'$ to V_{post},
     - add $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}_5$
  7. for each fragment $\mathtt{v} \in \mathtt{V}_1$ where $\mathtt{v}_x \overset{*}{\hookrightarrow}_V v$ and either $\mathtt{v} \overset{+}{\hookrightarrow}_{\mathtt{V}} \mathtt{v_y}$ or $\mathtt{v_y} \overset{*+}{\leftrightarrow} \mathtt{v}$, then for each subset $\mathtt{regset}$ of observer registers in $\mathtt{v.i.reachfrom} \cap \mathtt{v_x.i.reachfrom}$,

  - create $\mathtt{v}'$ which is same as $\mathtt{v}$, except that $\mathtt{v}'.\mathtt{i.reachfrom} = (\mathtt{v.i.reachfrom} \setminus \mathtt{v_x.i.reachfrom}) \cup \mathtt{regset}$.
  - for each set $\mathtt{regset}'$ of observer registers in $\mathtt{v}'.\mathtt{o.reachfrom} \cap \mathtt{v}_x.\mathtt{i.reachfrom}$,
    * create $\mathtt{v}''$ which is same as $\mathtt{v}'$, except that $\mathtt{v}''.\mathtt{o.reachfrom} = (\mathtt{v}'.\mathtt{o.reachfrom} \setminus \mathtt{v}_x.\mathtt{i.reachfrom}) \cup \mathtt{regset}'$.
    * add $\mathtt{v}''$ to V_{post}
    * add $(\mathtt{v}, \mathtt{v}'')$ to $\mathtt{R}_6$

8. add $\mathtt{v_{new}}$ to V_{post}

– for each fragment $\mathtt{v} \in V_2$ then we do as follows. for each $(\mathtt{v}_1, \mathtt{v}_1') \in \mathtt{R_i}$, $(\mathtt{v}_2, \mathtt{v}_2') \in \mathtt{R_j}$ where $\mathtt{i} \neq \mathtt{j}$ and $1 \leq \mathtt{i}, \mathtt{j} \leq 6$. For each pair of indices $\mathtt{i}_1$, $\mathtt{i}_2$ such that $\mathtt{i}_1 \in \{\mathtt{v}_1.\mathtt{i}, \mathtt{v}_1.\mathtt{o}\}$, $\mathtt{i}_2 \in \{\mathtt{v}_2.\mathtt{i}, \mathtt{v}_2.\mathtt{o}\}$, $\mathtt{tag}(\mathtt{v}, \mathtt{i}) = \mathtt{tag}(\mathtt{v}_1, \mathtt{i}_1)$, $\mathtt{tag}(\mathtt{v}, \mathtt{o}) = \mathtt{tag}(\mathtt{v}_2, \mathtt{i}_2)$. Create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
  - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v}_1'.i_1.vars$,
  - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v}_2'.i_2.vars$,
  - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v}_1'.i_1.reachfrom$,
  - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v}_2'.i_2.reachfrom$,
  - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v}_1'.i_1.reachto$,
  - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v}_2'.i_2.reachto$,

  then add $\mathtt{v}'$ to V_{post}.

Fragment Intersection: Let us describe the intersection of two fragments $\mathtt{v}_1 \in \mathtt{V}_1$ and $\mathtt{v}_2 \in \mathtt{V}_2$ denoted as $\mathtt{v}_1 \sqcap \mathtt{v}_2$. Firstly, we consider the case where both $\mathtt{v}_1$ and $\mathtt{v}_2$ have size 2.

- if $\mathtt{v}_1.\mathtt{greachfrom}(\mathtt{i}, \{1,2\}) \neq \emptyset$ and $\mathtt{v}_2.\mathtt{greachfrom}(\mathtt{i}, \{1,2\}) \neq \emptyset$ then
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{i}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{i})$ and $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_{12}\}$ where $\mathtt{v}_{12}$ is same as $\mathtt{v}_1$ except that, for all $\mathtt{l} \in \{1, 2\{1,2\}\}$
    - $\mathtt{v}_{12}.\mathtt{vars}(\mathtt{i}) = \mathtt{v}_1.\mathtt{vars}(\mathtt{i}) \cup \mathtt{v}_2.\mathtt{vars}(\mathtt{i})$
    - $\mathtt{v}_{12}.\mathtt{o}.\mathtt{pvars} = \mathtt{v}_1.\mathtt{o}.\mathtt{pvars} \cup \mathtt{v}_2.\mathtt{o}.\mathtt{pvars}$
    - $\mathtt{v}_{12}.\mathtt{reachfrom}(\mathtt{i}, \mathtt{l}) = \mathtt{v}_1.\mathtt{reachfrom}(\mathtt{i}, \mathtt{l}) \cup \mathtt{v}_2.\mathtt{reachfrom}(\mathtt{i}, \mathtt{l})$
    - $\mathtt{v}_{12}.\mathtt{reachfrom}(\mathtt{i}, \mathtt{l}) = \mathtt{v}_1.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l}) \cup \mathtt{v}_2.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l})$
- if $\mathtt{v}_1.\mathtt{greachfrom}(\mathtt{i}, \{1,2\}) = \emptyset$, $\mathtt{v}_2.\mathtt{greachfrom}(\mathtt{i}, \{1,2\}) = \emptyset$, $\mathtt{v}_1.\mathtt{greachfrom}(\mathtt{o}, \{1,2\}) \neq \emptyset$ and $\mathtt{v}_2.\mathtt{greachfrom}(\mathtt{o}, \{1,2\}) \neq \emptyset$ then
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$, $\mathtt{v}_1.\mathtt{private}(\mathtt{i}) = \mathtt{false}$ and $\mathtt{v}_2.\mathtt{private}(\mathtt{i}) = \mathtt{false}$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}'_1, \mathtt{v}'_2, \mathtt{v}_{12}\}$ where $\mathtt{v}'_1$ is same as $\mathtt{v}_1$ except that, for all $\mathtt{l} \in \{1, 2\{1,2\}\}$
    - $\mathtt{v}'_1.\mathtt{o}.\mathtt{pvars} = \mathtt{v}_1.\mathtt{o}.\mathtt{pvars} \cup \mathtt{v}_2.\mathtt{o}.\mathtt{pvars}$
    - $\mathtt{v}'_1.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l}) = \mathtt{v}_1.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l}) \cup \mathtt{v}_2.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l})$

    and $\mathtt{v}'_2$ is same as $\mathtt{v}_2$ except that
    - $\mathtt{v}'_2.\mathtt{o}.\mathtt{pvars} = \mathtt{v}_1.\mathtt{o}.\mathtt{pvars} \cup \mathtt{v}_2.\mathtt{o}.\mathtt{pvars}$
    - $\mathtt{v}'_2.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l}) = \mathtt{v}_1.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l}) \cup \mathtt{v}_2.\mathtt{reachfrom}(\mathtt{o}, \mathtt{l})$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$ and ($\mathtt{v}_1.\mathtt{private}(\mathtt{i}) = \mathtt{true}$ or $\mathtt{v}_2.\mathtt{private}(\mathtt{i}) = \mathtt{true}$) then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}'_1, \mathtt{v}'_2\}$
- if $\mathtt{v}_1.\mathtt{greachfrom}(\mathtt{i}, \{1,2\}) = \emptyset$, $\mathtt{v}_2.\mathtt{greachfrom}(\mathtt{i}, \{1,2\}) = \emptyset$, $\mathtt{v}_1.\mathtt{greachfrom}(\mathtt{o}, \{1,2\}) = \emptyset$ and $\mathtt{v}_2.\mathtt{greachfrom}(\mathtt{o}, \{1,2\}) = \emptyset$ then
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$, $\mathtt{v}_1.\mathtt{private}(\mathtt{i}) = \mathtt{false}$, $\mathtt{v}_1.\mathtt{o}.\mathtt{private} = \mathtt{false}$, $\mathtt{v}_1.\mathtt{o}.\mathtt{private} = \mathtt{false}$ and $\mathtt{v}_2.\mathtt{o}.\mathtt{private} = \mathtt{false}$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}'_1, \mathtt{v}'_2, \mathtt{v}_{12}\}$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$, ($\mathtt{v}_1.\mathtt{private}(\mathtt{i}) = \mathtt{true}$ or $\mathtt{v}_2.\mathtt{private}(\mathtt{i}) = \mathtt{true}$) and $\mathtt{v}_1.\mathtt{o}.\mathtt{private} = \mathtt{false}$ and $\mathtt{v}_2.\mathtt{o}.\mathtt{private} = \mathtt{false}$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}'_1, \mathtt{v}'_2\}$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) = \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$ and ($\mathtt{v}_1.\mathtt{o}.\mathtt{private} = \mathtt{true}$ or $\mathtt{v}_1.\mathtt{o}.\mathtt{private} = \mathtt{true}$) then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2\}$
  - if $\mathtt{gtag}(\mathtt{v}_1, \mathtt{o}) \neq \mathtt{gtag}(\mathtt{v}_2, \mathtt{o})$ then $\mathtt{v}_1 \sqcap \mathtt{v}_2 = \{\mathtt{v}_1, \mathtt{v}_2\}$

6 Timestamp Stack

7 Timestamp Abstraction

7.1 View Abstraction

For timestamp data structures we have to deal with timestamp ordering and unbound number of lists. The solutions are described as follows:

– About timestamp ordering, we add timestamp ordering information for each index of a view. The order of index `i` of view `v` is of the form $\texttt{v.i.ts} \diamond \texttt{x}$ where $\diamond \in \{<, =, >\}$ and `x` is an observer register. Intuitively, $\texttt{v.i.ts} \diamond \texttt{x}$ means that $\diamond$ is the order between the timestamp of `v.i` and timestamp of an index whose data is equal to `x`.
– To deal with the problem of unbounded number of lists. We use two kind of views which are c-views $\texttt{v}_\texttt{c}$ in a current list and o-views $\texttt{v}_\texttt{o}$ in other lists. Note that, current list is the list where the current thread is accessing to.

7.2 Post-computation

The post-computation is quite similar to singly-linked lists with several differences as follows: Before computing the post condition of this statement, we change all o-views in the other list to c-views and previous c-views to o-views.

7.3 Intersection

The intersection between two views are computed same as in the case of singly-linked lists with several differences as follows:

– Two views of push methods should not be intersected. The reason for it is that we do not have more concurrent pushes in a same list.
– We can intersect o-views and c-views, o-views and o-views as well as c-views and c-views

8 Experimental Results

I am working here

Based on our framework, we have implemented a tool in OCaml, and used it for verifying concurrent algorithms (both lock-based and lock-free) including timestamps stack and queue, skip-list sets and priority queues as well as singly-linked lists algorithms (stacks, queues, sets). The experiments were performed on a desktop 2.8 GHz processor with 8GB memory. The results are presented in Fig. 9, where running times are given in seconds. All experiments start from the initial heap, and end either when the analysis reaches the fixed point or when a violation of safety properties or linearizability is detected.

Running Times. As can be seen from the table, the running times vary in the different examples. This is due to the types of shapes that are produced during the analysis. For instance, skip-lists algorithm have much longer running times. This is due to the number of pointer variables and their complicated shapes. Whereas, other algorithms produce simple shape patterns and hence they have shorter running times.

Algorithms	Time (s)
TIMESTAMPS	
TS stack [28]	176
TS queue [28]	101
SKIP-LISTS	
Lock-free skip-list [22]	1992
Optimistic skip-list [28]	500
Priority queue skip-list 1 [29]	1320
Priority queue skip-list 2 [29]	599
SINGLY-LINKED LISTS	
Treiber stack [36]	18
MS lock-free queue [28]	21
DGLM queue [10]	16
Vechev-CAS set [42]	86
Vechev-DCAS set [42]	16
Michael lock-free set [26]	178
Pessimistic set [22]	30
Optimistic set [22]	25
Lazy set [18]	34
O′Hearn set [30]	88
HM lock-free set [22]	120

Fig. 9: Experimental results for verifying concurrent programs

Error Detection In addition to establishing correctness of the original versions of the benchmark algorithms, we tested our tool with intentionally inserted bugs. For example, we emitted setting time statement in line 5 of the `push` method in TS stack algorithm. The tool, as expected, successfully detected and reported the bug.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS. LNCS, vol. 7795, pp. 324–338 (2013)
2. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Automated verification of linearization policies. In: SAS. Lecture Notes in Computer Science, vol. 9837, pp. 61–83. Springer (2016)
3. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV'07. LNCS, vol. 4590, pp. 477–490 (2007)
4. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: CAV'08. LNCS, vol. 5123, pp. 399–413 (2008)
5. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. In: ICALP. LNCS, vol. 9135, pp. 95–107 (2015)
6. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: CAV. LNCS, vol. 4144, pp. 475–488 (2006)
7. Derrick, J., Dongol, B., Schellhorn, G., Tofan, B., Travkin, O., Wehrheim, H.: Quiescent consistency: Defining and verifying relaxed linearizability. In: FM. LNCS, vol. 8442, pp. 200–214 (2014)

8. Dodds, M., Haas, A., Kirsch, C.: A scalable, correct time-stamped stack. In: POPL. pp. 233–246. ACM (2015), `http://dl.acm.org/citation.cfm?id=2676726`
9. Doherty, S., Detlefs, D., Groves, L., Flood, C., Luchangco, V., Martin, P., Moir, M., Shavit, N., Steele Jr., G.: DCAS is not a silver bullet for nonblocking algorithm design. In: SPAA'04. pp. 216–224. ACM (2004)
10. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE'04. LNCS, vol. 3235, pp. 97–114 (2004)
11. Dragoi, C., Gupta, A., Henzinger, T.A.: Automatic linearizability proofs of concurrent objects with cooperating updates. In: CAV. LNCS, vol. 8044, pp. 174–190 (2013)
12. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: TACAS. LNCS, vol. 6015, pp. 296–311. Springer (2010)
13. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC'04. pp. 50–59. ACM (2004)
14. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: Proc. of PLDI'07. pp. 266–277. ACM (2007)
15. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. pp. 300–314 (2001)
16. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: DISC,. pp. 265–279 (2002)
17. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 393–412. Springer (2016)
18. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. pp. 3–16 (2005)
19. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. J. Parallel Distrib. Comput. 70(1), 1–12 (2010)
20. Henzinger, T., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: CONCUR. LNCS, vol. 8052, pp. 242–256. Springer (2013)
21. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
22. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
23. Holík, L., Meyer, R., Vojnar, T., Wolff, S.: Effect summaries for thread-modular analysis - sound analysis despite an unsound heuristic. In: SAS. LNCS, vol. 10422, pp. 169–191. Springer (2017)
24. Lesani, M., Millstein, T., Palsberg, J.: Automatic atomicity verification for clients of concurrent data structures. In: CAV. LNCS, vol. 8559, pp. 550–567. Springer (2014)
25. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI. pp. 459–470. ACM (2013)
26. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. pp. 73–82 (2002)
27. Michael, M., Scott, M.: Correction of a memory management method for lock-free data structures. Tech. Rep. TR599, University of Rochester, Rochester, NY, USA (1995)
28. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC. pp. 267–275 (1996)
29. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA. pp. 253–262 (2005)
30. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC. pp. 85–94 (2010)
31. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. 15(4), 31:1–31:37 (2014)

32. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: CAV. LNCS, vol. 7358, pp. 243–259. Springer (2012)
33. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Abstract transformers for thread correlation analysis. In: APLAS. LNCS, vol. 5904, pp. 30–46. Springer (2009)
34. Singh, V., Neamtiu, I., Gupta, R.: Proving concurrent data structures linearizable. In: ISSRE. pp. 230–240 (2016)
35. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5), 609–627 (May 2005)
36. Treiber, R.: Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Res. Ctr. (1986)
37. Turon, A.J., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: POPL '13. pp. 343–356 (2013)
38. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI. LNCS, vol. 5403, pp. 335–348 (2009)
39. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)
40. Vafeiadis, V.: Automatically proving linearizability. In: CAV. LNCS, vol. 6174, pp. 450–464 (2010)
41. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: CAV. LNCS, vol. 6174, pp. 465–479 (2010)
42. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI. pp. 125–135 (2008)
43. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: SPIN. LNCS, vol. 5578, pp. 261–278. Springer (2009)
44. Zhang, K., Zhao, Y., Yang, Y., Liu, Y., Spear, M.F.: Practical non-blocking unordered lists. In: DISC. pp. 239–253 (2013)
45. Zhu, H., Petri, G., Jagannathan, S.: Poling: SMT aided linearizability proofs. In: CAV. LNCS, vol. 9207, pp. 3–19. Springer (2015)

A Algorithms

In this section, we show several important algorithms including timestamp stack and queue, lazy set, skip-list sets and skip-list priority queue.

```
struct Node {
         int data;
         Timestamp ts;
         Node* next;
         bool mark;
      }

init() :
 Node* pools[maxThreads];
 for(int i=0; i<maxThreads; i++)
   pools[i].next = null;

void push(int d):
1  Node* new := new Node(d,-1,null,false);
   new.next = pools[myID];
   pools[myID] = new;
   Timestamp t = new Timestamp();
   new.ts = t;
   Node* next = new.next;
   while (next.next != next & !next.mark)
     next = next.next;
   new.next = next;
   return new;

Node remove(Node* pt, Node* t, Node* n)
   bool s = CAS(n.mark,false,true);
     if (s)
       CAS(pt, t, n);
       if (t != n);
         t.next = n;
       t.next = n.next;
       Node* next=n.next
       while (next.next != next && next.mark);
         next = next.next;
       n.next = next;
       return s;
```

```
int pop(Timestamp ts):
  boolean success = false;
  int maxTS = -1;
  Node* youngest, myTop, n = null;
  Node* empty[maxThreads];
  while (!success)
    int k;
    for(int i=0; i<maxThreads; i++)
      myTop = pools[i]; n = myTop;
      while (n.mark && n.next != n) n = n.next;
      if(n = null)
        empty[i] = pools[i];
        continue;
      if(st < n.ts)
        r = remove(pools[i], top, n);
        return n.data);
      if(maxTS < n.ts)
        maxTS = n.ts;
        youngest = n;
        k = i;
    if (youngest != null)
      success= remove(pools[k],myTop, youngest);
    if (youngest = null)
      for(int i=0; i<maxThreads; i++)
          if (pools[i] != empty[i]);
            return NonEmpty;
      return Empty;
  return youngest.data;
```

Fig. 10: Timestamp Stack.

```
struct Node {
         int data;
         Timestamp ts;
         Node* next;
         bool mark;
      }

init() :
 Node* pools[maxThreads];
 for(int i=0; i<maxThreads; i++)
   pools[i].next = null;

void enq(int d):
  Node* new := new Node(d,-1,null,false);
  new.next = pools[myID];
  pools[myID] = new;
  Timestamp t = new Timestamp();
  new.ts = t;
  Node* next = new.next;
  while (next.next != next & !next.mark)
    next = next.next;
  new.next = next;
  return new;
```

```
int deq(Timestamp ts):
  boolean success = false;
  int maxTS = 10000;
  Node* youngest, myTop, n = null;
  Node* empty[maxThreads];
  while (!success)
    int k;
    for(int i=0; i<maxThreads; i++)
      myTop = pools[i]; n = myTop;
      while (n.mark && n.next != n) n = n.next;
      if(n = null)
        empty[i] = pools[i];
        continue;
      if(maxTS > n.ts)
        maxTS = n.ts;
        youngest = n;
        k = i;
    if (youngest != null)
      success= remove(pools[k],myTop, youngest);
    if (youngest = null)
      for(int i=0; i<maxThreads; i++)
          if (pools[i] != empty[i]);
            return NonEmpty;
      return Empty;
    return youngest.data;
```

```
Node remove(Node* pt, Node* t, Node* n)
   bool s = CAS(n.mark,false,true);
     if (s)
       CAS(pt, t, n);
       if (t != n);
         t.next = n;
       t.next = n.next;
       Node* next=n.next
       while (next.next != next && next.mark);
         next = next.next;
       n.next = next;
       return s;
```

Fig. 11: Timestamp Queue.

```
struct Node(bool lock; int val; Node *next; bool mark);

 <Node, Node> locate(int d):
 local pred, curr
   while (true)
      pred := head;
      curr := pred.next;
      while (curr.val < d)
         pred := curr;
         curr := curr.next
      lock(pred); lock(curr);
      if (! pred.mark &&
           ! curr.mark&&
           pred.next=curr)
         return(pred,curr);
      else
         unlock(pred);
         unlock(curr);

bool ctn(int d):
local curr
  curr := Head;
  while (curr.val < d)
     curr := curr.next
  b := curr.mark
  if (! b && curr.val = d)
     return true;
  else return false;
```

```
bool add(int d):
   local pred, curr, n, r
   1 (pred,curr) := locate(d);
      if (curr.val <> d)
        n :=
        new Node(0,d,curr,false);
        pred.next := n;
        r := true;
      else r := false;
      unlock(pred);
      unlock(curr);
      return r;

bool rmv(int d):
local pred, curr, n, r
1 (pred,curr) = locate(d);
   if (curr.val = d)
      curr.mark = true;
      n = curr.next;
      pred.next = n;
      r = true;
   else r = false;
   unlock(pred);
   unlock(curr);
   return r;
```

Fig. 12: Lazy Set.

```
struct Node(int key; int topLayer; Node *next[]; bool marked; fullylinked;Lock lock);
```

```
 locate(int v,Node* preds[], Node* succs[]):
    local lfound, pred, curr;
      pred = H; lfound = -1
      for (int i = maxHeight; i >= 1; i--);
         curr = pred.next[i];
         while (curr.val < v)
            pred = curr;
            curr = curr.next[i]
         if (lfound = 1 && curr.val = v)
            lfound = i
         preds[i] = pred;
         currs[i] = curr
      return lfound

rmv(int v):
 Node* nodeToDelete = null;
 bool isMarked = false;
 int level = -1;
 Node* preds[MaxHeight], succs[MaxHeight];
 while (true)
   int lFound = findNode(v, preds, succs);
   if (isMarked || lFound != -1 &&
     (okToDelete (succs[lFound], lFound)))
     if (!isMarked)
       nodeToDelete = succs[lFound];
       topLayer = nodeToDelete.level;
       lock(nodeToDelete);
       if(nodeToDelete.marked);
         unlock(nodeToDelete);
       return false;
       nodeToDelete.marked = true;
       isMarked = true;
     int highestLocked = -1;
     try
       Node* pred, succ, prevPred = null;
       bool valid = true;
       for (int i = 0;valid &&li<=level; i++)
         pred = preds[i];
         succ = succs[i];
         if (pred != prevPred)
           lock(pred);
         highestLocked = i;
         prevPred = pred;
         valid = !pred.marked && pred.next[i]==succ;
       if (!valid) continue;
       for (int j = level; j >= 0; j--)
         preds[j].nexts[j] = nodeToDelete.nexts[j];
       unlock(nodeToDelete);
       return true;
     finally unlock(preds,highestLocked);
   else return false;
```

```
add(int v):
 int level = randomLevel(MaxHeight);
 Node* preds[MaxHeight], succs[MaxHeight];
 while(true)
   int lfound = findnode(v, preds, succs);
   if (lfound != 1)
     Node* nodeFound = succs[lfound];
     if (!nodeFound.marked)
       while (!nodeFound.fullyLinked)
       return false;
     continue;
     int highestLocked = -1;
     try
       Node* pred, succ, prevPred = null;
       bool valid = true;
       for (int i = 0; valid&&
           i<=level;i++)
         pred = preds[i];
         succ = succs[i];
         if (pred != prevPred)
           pred.lock();
           highestLocked = i;
           prevPred = pred;
         valid = !pred.marked && !succ.marked
                && pred.nexts[i]= succ;
       if (!valid ) continue;
       Node* newNode = new Node(v,level);
       for (int j = 0; j <= level; j++)
         newNode.nexts[j] = succs[j];
         preds[j].nexts[layer] = newNode;
       newNode.fullyLinked = true;
       return true;
     finally unlock(preds, highestLocked);


bool ctn(int v):
Node* preds[MaxHeight], succs[MaxHeight] ;
  int lFound = findNode ( v,preds,succs);
  return (lFound != -1
          && succs[lFound].fullyLinked
          && !succs[lFound].marked);


bool okToDelete(Node* candidate,int lFound):
   return (candidate.fullyLinked
         && candidate.topLayer=lFound
         && !candidate.marked);
```

Fig. 13: Optimistic Skiplist.

```
struct Node(int key; int topLayer; Node *next[]; bool marked);

 boolean find(int x, Node* preds[], Node* succs[]):
 int mLevel = 0;
 boolean marked[MAXLEVEL] = false;
 boolean snip;
 Node* pred = null, curr = null, succ = null;
 retry:
   while (true)
     pred = head;
     for (int i = MAXLEVEL; i >= mLevel; i--)
       curr = pred.next[i];
       while (true)
         succ = curr.next[i].get(marked);
         while (marked[0])
         s= CAS(pred.next[i],curr,succ,false,false);
           if (!s) continue retry;
           curr = pred.next[i];
           succ = curr.next[i].get(marked);
         if (curr.key < x)
           pred = curr; curr = succ;
         else
           break;
       preds[i] = pred;
       succs[i] = curr;
     return (curr.key == key);

  bool rmv(int x):
   Node* nodeToDelete = null;
   int mLevel = 0;
   Node* preds[MAXLEVEL+1];
   Node* succs[MAXLEVEL+1];
   Node* succ;
   while (true)
     boolean found = find(x, preds, succs);
     if (!found)
       return false;
     else
       Node* node = succs[mLevel];
       for (int i = node.topLevel; i >= mLevel+1; i--)
         boolean marked[MAXLEVEL+1] = false;
         succ = node.next[i].get(marked);
         while (!marked[0])
           node.next[i].attemptMark(succ, true);
           succ = node.next[i].get(marked);
       boolean marked[MAXLEVEL+1] = false;
       succ = node.next[mLevel].get(marked);
       while (true)
         boolean iMarkedIt =
         CAS(node.next[mLevel],succ, succ, false, true);
         succ = succs[mLevel].next[mLevel].get(marked);
         if (iMarkedIt)
           find(x, preds, succs);
           return true;
         else if (marked[0]) return false;
```

```
add(int v):
 int topLevel = randomLevel();
 int mLevel = 0;
 Node* preds[MAXLEVEL+1];
 Node* succs[MAXLEVEL+1];
 while (true)
   boolean found = find(x, preds, succs);
   if (found)
     return false;
   else
     Node* new = new Node(x, topLevel);
     for (int i= mLevel;level<=topLevel;i++)
       Node* succ = succs[i];
       new.next[i].set(succ, false);
     Node* pred = preds[mLevel];
     Node* succ = succs[mLevel];
     new.next[mLevel].set(succ, false);
     if (!CAS(pred.next[mLevel],succ,new,false,false))
       continue;
     for (int i=mLevel+1;i<=topLevel;i++)
       while (true)
         pred = preds[i];
         succ = succs[i];
         if (CAS(pred.next[i],succ,new,false,false))
           break;
         find(x,preds,succs);
     return true;

bool ctn(int x):
int bottomLevel = 0;
bool marked[MAXLEVEL] = false;
Node* pred = head, curr = null, succ = null;
for (int i = MAXLEVEL; i >= mLevel; i--)
  curr = pred.next[i];
  while (true)
    succ = curr.next[i].get(marked);
    while (marked[0])
      curr = pred.next[i];
      succ = curr.next[i].get(marked);
    if (curr.key < x)
      pred = curr;
      curr = succ;
    else
      break;
return (curr.key == v);
```

Fig. 14: Lock-free Skiplist.

```
struct Node(int key; int level; Node *next[]; value value; Timestamp timestamp );

 Node * getLock(node* node1,int key,int level):
 node2 = node1.next[level];
 while (node2.key < key)
   node1 = node2;
   node2 = node1.next[level];
 lock(node1, level);
 node2 = node1.next[level];
 while (node2.key < key);
   unlock(node1, level);
   node1 = node2;
   lock(node1, level);
   node2 = node1.next[level];
 return node1;


  int DeleteMin(value value)):
   time = getTime();
   node1 = head.next[1];
   while (node1 != tail)
     if (node1.timeStamp < time)
       marked = SWAP(node1.deleted, true);
       if (marked = FALSE) break;
       node1 = node1.next[1];
   if (node1 != tail)
     value = node1.value;
     key = node1.key;
     return EMPTY;
   node1 = head;
   for (i = MaxLevel; i > 0; i--)
     node2 = node1.next[i];
     while (node2.key > key)
       node1 = node2;
       node2 = node2.next[i];
     savedNodes[i] = node1;
   node2 = node1;
   while (node2.key != key)
     node2 = node2.next[1];
     lock(node2);
     for (i = node2.level; i > 0; i--)
     node1 = getLock(savedNodes[i], key, i);
     lock(node2, i);
     node1.next[i] = node2.next[i];
     node2.next[i] = node1;
     unlock(node2, i);
   unlock(node1, i);
   unlock(node2);
   return DELETE;
```

```
int Insert(key key, value value):
 node1 = head;
 for (int i= MaxLevel; i > 0; i--)
   node2 = node1.next[i];
   while (node2.key > key)
     node1 = node2;
     node2 = node2.next[i];
 savedNodes[i] = node1;
 node1 = getLock(node1, key, 1);
 node2 = node1.next[i];
 if (node2.key = key)
   node2.value = value;
   unlock(node1, 1);
   return UPDATED;
 level = randomLevel();
 newNode = newNode(level, key, value);
 newNode.timeStamp = MAXTIME;
 lock(newNode)
 for (i = 1; i <= level; i++)
   if (i != 1)
     node1 = getLock(savedNodes[i], key, i);
   newNode.next[i] = node1.next[i];
   node1.next[i] = newNode;
   unlock(node1, i);
 unlock(newNode);
 newNode.timeStamp = getTime();
 return INSERTED;
```

Fig. 15: SkiplistBased Concurrent Priority Queues.