Fragment Abstraction for Concurrent Shape Analysis

No Institute Given

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 [10, 24]. Consequently, significant research efforts have been directed towards developing techniques to verify correctness of such algorithms, in particular 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 such techniques require significant *manual* effort for constructing a proof of correctness (e.g., [22, 35]), in some cases with the support of an interactive theorem prover (e.g., [33, 7, 8, 29, 28]). Development of automated verification techniques is a difficult challenge.

A major difficulty is that a successful verification technique 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 is the problem of handling dynamically allocated memory. Consequently, all existing techniques for automatically proving correctness of such concurrent algorithms restrict attention to the case where the heap represents shared data byp singly-linked lists [1, 16, 2, 30, 36]. Many of these techniques impose additional restrictions on the considered problem, such as bounding the number of accessing threads [3, 40, 38]. [COMMENT: Add other restrictions] However, many concurrent data structure implementations employ more sophisticated structures, such as skip lists [13, 21, 31], trees, and arrays of of singly-linked lists [9]. There are no techniques that have been applied to automatically verify concurrent algorithms that operate on such data structures.

The last list and its description 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 employ skip lists, singly linked lists, as well as arrays of singly linked lists, 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 way is able to represent different kinds of unbounded heap structures, such as arrays of singly linked lists and skip lists. Its main idea is to represent a set of heap states by a set of *fragments*. Each fragment is simply a pair of node types (called *tags*) that are connected by a pointer. A tag can be seen as a finitary abstraction of a heap node, which summarizes both local information about values of its data fields as well as global information about its position in the heap, including how it 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 describes how pairs of pointer-connected nodes can be "pieced together" to form heap structures. In other words, 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. By construction, our fragment abstraction is finitary, since there is a bounded set of tags.

Fragment abstraction can, in a natural way, be combined with other abstractions for handling unbounded data domains and for handling an unbounded number of threads. Our fragment abstraction technique copes with an unbounded data domain by letting the definition of tags incorporate a suitable data abstraction to the data fields in heap nodes. We cope with the challenge of an unbounded number of threads by incorporating the successful thread-modular approach [4]; this is done simply by letting each set of fragments represent only the heap cells that are accessible to an arbitrary single thread.

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, 19]; 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

This related work remains to be written

Related Work Much previous work has been devoted to the *manual* verification of linearizability for concurrent programs. Examples include [22, 33]. In [27], 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 [35] 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 [7, 8, 29, 28]. For instance, Colvin *et al.* [7] verify the lazy set algorithm in PVS, using a combination of forward and backward simulations.

There are several works on *automatic* verification of linearizability. In [36], 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 [26], *HSY* stack [18], *CCAS* [15], *RDCSS* [15] and *HM* set [21] that we consider in this paper. In addition, their shape abstraction is not powerful enough to handle algorithms like *Harris* set [14] and *Michael* set [23] that are also handled by our method. Chakraborty *et al.* [19] 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.* [42] 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 [20] 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 [27]. 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* [21], *Harris* [14] and *Michael* [23] sets cannot be verified by their technique. Vechev *et al.* [40] 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.* [12] 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* [38] show decidability of a class of programs with a bounded number of threads operating on concurrent data structures. Finally, the works [1, 4, 34] 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* [21], *Harris* [14], or *Michael* [23], or unordered set of [41],

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, namely the Timestamped Stack of [9]. This algorithm operates on a shared heap which represents an array of singly linked lists. The verification of correctness for this algorithm and the related Timestamped queue, also of [9], has been challenging. Only recently has the TS stack been verified by non-automated techniques [6] using a non-trivial extension of forward simulation, and the TS queue by been verified manually by a new technique based on partial orders [9]. We have verified both these algorithms automatically using fragment abstraction, as reported in Section 8.

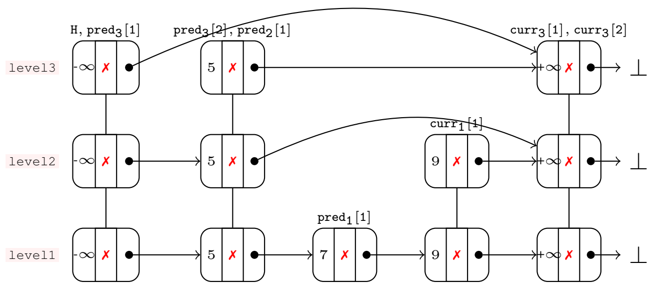


Fig. 1: A concrete shape of 3-level skipl-list with two threads

2.1 The Timestamped Stack Algorithm

Figure 3 shows a simplified version of the Timestamped Stack (TS stack), where we have omitted the check for emptiness in the `pop` method, and the optimization using `push`-`pop` elimination. These features are included in the full version of the algorithm, described in Appendix XXX, that we have verified automatically.

The algorithm uses an array of singly-linked lists (SLLs), one for each thread, accessed via the thread-indexed array `pools`[`maxThreads`] of pointers to the first cell of each list. The `init` method initializes each of these pointers to `null`. Each list cell contains a data value, a timestamp value, a `next` pointer, and a boolean flag `mark` which

```
struct Node { int data; 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;
int k;
for(int i=0; i<maxThreads; i++)
    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);
```

Fig. 2: Description of the find function of the Skip-list algorithm

indicates whether the node is logically removed from the stack. Each thread pushes elements to "its own" list, and can pop elements from any list.

A push method for inserting a data element d works as follows: first, a new cell with element d and minimal timestamp -1 is inserted at the beginning of the list indexed by the calling thread (line 1-3). After that, a new timestamp is created and assigned (via the variable t) to the ts field of the inserted cell (line 4-5). Finally, the method unlinks (i.e., physically removes) all cells that are reachable (through a sequence of next pointers) from the inserted cell and whose mark field is true; these cells are already logically removed. This is done by redirecting the next pointer of the inserted cell to the first cell with a false mark field, which is reachable from the inserted cell.

A pop method first traverses all lists, finding in each list the first cell whose mark field is false(line 8), and letting the variable youngest point to the most recent such cell (i.e., with the largest timestamp) (line 1-11). A compare-and-swap (CAS) is used to set the mark field of this youngest cell to true, thereby logically removing it. This procedure will restart if the CAS fails. After the youngest cell has been removed, the method will unlink all cells, whose mark field is true, that appear before (line 17-19) or after (line 20-23) the removed cell. Finally, the method returns the data value of the removed cell.

```
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):
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 pop():
boolean success = false;
int maxTS = -1;
Node* youngest, myTop, n = null;
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(maxTS < n.ts)
      maxTS = n.ts;
      youngest = n;
      k = i;
  if (youngest != null)
    success= CAS(youngest.mark,false,true);
      if (success)
        CAS(pools[k], myTop, youngest);
        if (myTop != youngest);
          myTop.next = youngest;
        pools[k].next = youngest.next;
        Node* next=youngest.next
        while (next.next != next & next.mark);
          next = next.next;
        youngest.next = next;
return youngest.data;
```

Fig. 3: Description of the Timestamped stack algorithm, with some simplifications.

It would here be nice with some intuitive argument why the algorithm works

2.2 Specifying the Correctness Criterion of Linearizability

In our verification, we establish that the TS stack algorithm of Figure 3 is is correct in the sense that it is a linearizable implementation of a stack 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 [20]. We specify linearizability using the technique of *observers* [1]. Observers act like monitors that synchronize with the monitored concurrent programs over designated actions, and report violations of the linearizability criterion. In our verification, we use observers 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, 19]; 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, typically by affixing linearization points to particular statements, or in more complex cases by light-weight instrumentation using the approach of controllers [2]; this puts a small burden on the verifier. In our work, we

use this technique to specify linearizability of set implementations (see Section XXX).

In our verification of TS stack, we use the first approach.

In the case of a stack, we use observers to check constraints that intuitively express that

1. a data item must not be popped before it is pushed,
2. pop must not return "empty" if some data element was pushed but not popped,
3. a pushed data item must not be popped twice, and
4. any two data items must be popped in last in first out stack order.

For a sequential data structure, it is straight-forward to construct observers that check each of these constraints. For concurrent data structures, one must be careful to expressed them exactly in terms of allowed sequences of calls and returns of different methods. For completeness, we here present how this can be done for a stack implementation, as shown in [5].

Let `call push(d)` and `ret push(d)` denote the action of calling or returning from a `push` method with data parameter `d`. Let `call pop(d)` and `ret pop(d)` denote the action of calling or returning from a `pop` method which returns the value `d`. Constraints on the ordering of such actions can be checked by an *observer*. Observers are finite automata extended with a finite set of *registers* whose values range over the domain of data values. At initialization, the registers are nondeterministically assigned arbitrary distinct data values, which never change during a run of the observer. Transitions are labeled by actions parameterized by registers. The observer processes a sequence of actions, one action at a time. If there is a transition, whose label, after replacing registers by their values, matches the action, such a transition is performed; otherwise the observer remains in its current state. The observer accepts a sequence of actions if, for some initial assignment of data values to its registers, the sequence can be processed in such a way that an accepting state is reached. The observer for a particular constraint is defined in such a way that it accepts precisely those sequences that do *not* satisfy the constraint.

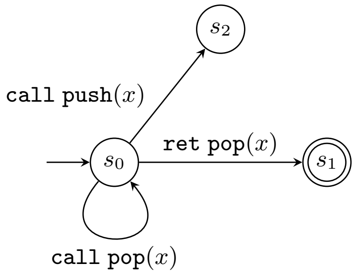


Fig. 4: An observer recognizing sequences in which `pop` for some data value returns although no corresponding `push` has been previously called. The observer has one register `x`.

As an example, Figure 4 shows an observer for Constraint 1, which checks that a pop method must not return before any push method with the same data value has been called. Observers for Constraints 2 and 3 are similar to that for Constraint 1. The observer for Constraint 4 is more complex, and shown in Figure 5. It has two registers $\mathtt{x}_1$ and $\mathtt{x}_2$. Let $\mathtt{d}_1$ and $\mathtt{d}_2$ denote their initial values. In a sequence of actions that leads to the accepting state s_6, a $\mathtt{push}(\mathtt{d}_1)$ must be linearized before a $\mathtt{push}(\mathtt{d}_2)$, since $\mathtt{push}(\mathtt{d}_1)$ returns before $\mathtt{push}(\mathtt{d}_2)$ is called; however a $\mathtt{pop}(\mathtt{d}_1)$ returns in a situation where the number of returned $\mathtt{push}(\mathtt{d}_2)$ methods exceeds the number of called $\mathtt{pop}(\mathtt{d}_2)$ methods, thereby violating LIFO ordering.

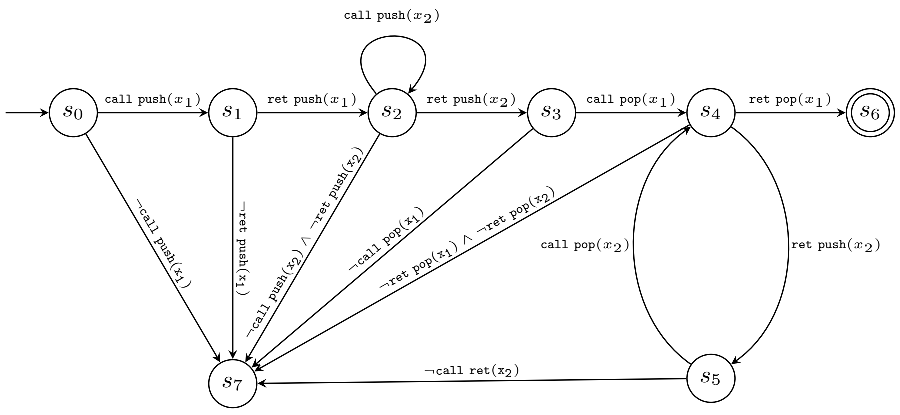


Fig. 5: An observer recognizing LIFO violations. The registers are $\mathtt{x}_1$ and $\mathtt{x}_2$. For each action $\mathtt{act}$ parameterized by $\mathtt{x}_1$ or $\mathtt{x}_2$, the label $\neg\mathtt{act}(\mathtt{x}_\mathtt{i})$ is the union of all actions parameterized by $\mathtt{x}_1$ or $\mathtt{x}_2$ except for $\mathtt{act}(\mathtt{x}_\mathtt{i})$.

To verify that TS Stack is a correct linearizable implementation of a stack, we must show that any concurrent execution of method calls satisfies the above four constraints (1) – (4). To this end, define a *program* to consist of an arbitrary number of concurrently executing threads, each of which executes a push or pop method with some data value as parameter. We assume that the data structure has been initialized by the `init` method prior to the start of program execution. We must verify that any execution of such a program satisfies the four constraints above. To verify a constraint, we form, as in the automata-theoretic approach [37], also adopted in [1], the cross-product of the program and the corresponding observer, where the observer synchronizes with the program on call and return actions that are recognized by the observer. This reduces the problem of checking each of the above constraints to the problem of checking that the corresponding observer cannot reach an accepting state.

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

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 can also be combined with data abstraction and thread abstraction. Let us illustrate how fragment abstraction to the TS stack algorithm.

Figure 6 shows an example heap state of TS stack. The heap consists of a set of singly linked lists (SLLs), each of which is accessed from a pointer in the array `pools`[`maxThreads`] in a configuration when it is accessed concurrently by three threads $\mathtt{th}_1$, $\mathtt{th}_2$, and $\mathtt{th}_3$. The heap consists of three SLLs accessed from the three pointers `pools`[1], `pools`[2], and `pools`[3] respectively. Each heap cell is shown with the values of its fields, using the layout shown to the right in Figure 6. 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 6, thread $\mathtt{th}_1$ is trying to push a new node with data value 4, pointed by its local variable `new`, having reached line 3. Thread $\mathtt{th}_3$ has just called the `push` method. Thread $\mathtt{th}_2$ has reached line 12 in the execution of the `pop` method, and has just assigned `youngest` to the first node in the list pointed to by `pools`[3] which is not logically removed (in this case it is the last node of that list). Not shown in Figure 6 is the configuration of the observer. In this case, the observer is the one in Figure 5, which has two registers $\mathtt{x}_1$ and $\mathtt{x}_2$, which are assigned the values 4 and 2, respectively.

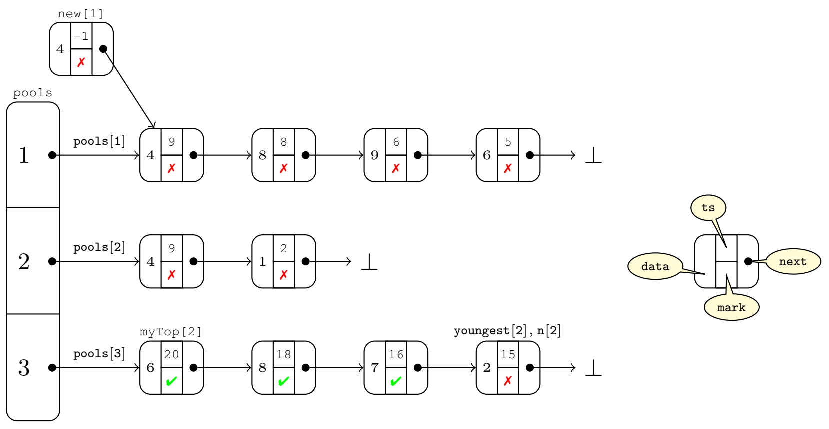


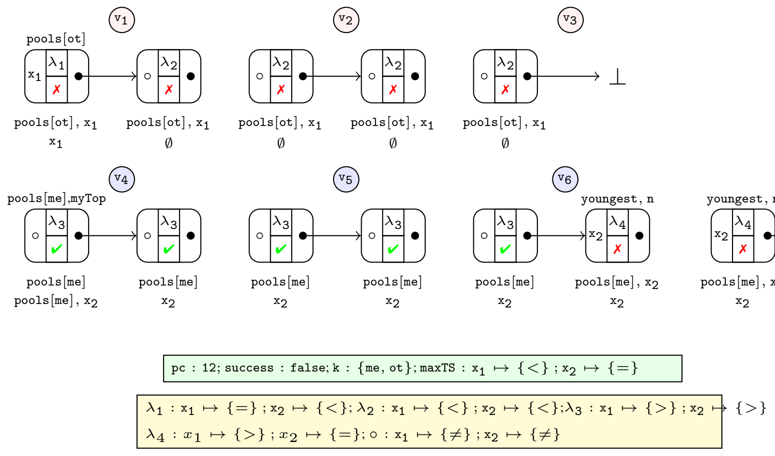
Fig. 6: A possible heap state of TS stack with three threads.

Our fragment abstraction represents a set of heap states by a set of *fragments*. Each fragment consists of a pair of node types (called *tags*) that are connected by a pointer. A tag is an abstraction of a heap cell, which contains both

(i) local information about its non-pointer fields, obtained using data abstraction if necessary, and about variables pointing to the cell, and

(ii) global information, saying how it can reach to and be reached from (by following a chain of pointers) other heap cells that are pointed to by global variables, or cells whose data field has the same value as some observer register. This global information describes how the cell is positioned relative to "important" cells in the heap. Note that cells which carry the the same data value as some observer register are also considered "important", since it is necessary to track their position in order to verify ordering constraints that are checked by the observer. [COMMENT: Do we need to explain this better?]

Our fragment abstraction incorporate the thread-modular approach by letting tags consider local variables of only one arbitrary single thread, and only requiring that fragments represent heap cells that can be accessed by that thread.



v_1 v_2 v_3 v_4 v_5 v_6 v_7
pools[ot]
pools[ot], x_1
x_1
$\emptyset$
pools[me],myTop
pools[me]
pools[me], x_2
x_2
youngest, n
λ_1 λ_2 λ_3 λ_4
$\perp$
pc : 12; success : false; k : {me, ot}; maxTS : $x_1 \mapsto \{<\}$; $x_2 \mapsto \{=\}$
$\lambda_1 : x_1 \mapsto \{=\}$; $x_2 \mapsto \{<\}$; $\lambda_2 : x_1 \mapsto \{<\}$; $x_2 \mapsto \{<\}$;$\lambda_3 : x_1 \mapsto \{>\}$; $x_2 \mapsto \{>\}$
$\lambda_4 : x_1 \mapsto \{>\}$; $x_2 \mapsto \{=\}$;$\circ : x_1 \mapsto \{\neq\}$; $x_2 \mapsto \{\neq\}$

Fig. 7: Fragment abstraction

Figure 7 shows a set of fragments that is satisfied wrp. to $\mathtt{th}_2$ by the configuration in Figure 6. There are 7 fragments, named $v_1, \ldots, v_2$. Two of these (v_3 and v_7) consist of a tag that points to $\perp$, and the other consist of a pair of pointer-connected tags. Consider the tag which occurs in fragment v_7; it is the bottom-rightmost tag. This tag is an abstraction of the bottom-rightmost heap cell in Figure 6, using the same layout for fields as Figure 6. The different non-pointer fields are represented as follows.

– 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 7.

– 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 6 now satisfies some tag in Figure 6. Moreoever, each pair of pointer-connected cells (where the pointed-to "cell" can also be $\perp$) satisfies some fragment in Figure 6 in the obvious way. Conversely, the set of fragments in Figure 6 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 $v_4 v_5 v_6 v_7$.

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 th_2 corresponding to Figure 6 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 $\{me, ot\}$ and applies the timestamp abstraction to `maxTS`.

We are now ready to present our symbolic representation of a set of reachable program configurations. Our symbolic representation is a mapping from a set of local symbolic configurations, which maps each local symbolic configuration in its domain to a set of fragments. A global configuration satisfies a symbolic representation Ψ if the local state of each thread `th` satisfies some local symbolic configuration in the domain of Ψ, which is mapped to a set of fragments, which is satisfied by heap wrp. to thread `th`.

Our symbolic representation represents the reachable local states of a thread by a finite set of *local symbolic configurations*, using our data abstraction. It maps each local symbolic configuration to a set of fragments, which is a symbolic representation of the set of corresponding reachable heaps. Thus, our symbolic representation is a mapping from a set of local symbolic configurations, which maps each local symbolic configuration in its domain to a set of fragments. A global configuration satisfies a symbolic representation Ψ if the local stat of each thread `th` satisfies some local symbolic configuration in the domain of Ψ, which is mapped to a set of fragments, which represents the heap that is accessible to `th` in the global configuration. In the following, we explain more precisely the local symbolic configurations, and our fragment abstraction.

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

Variables are either pointer variables (to heap cells) or data variables, assuming values from $\mathbb{B}$ 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 begin by introducing (sequential) data structures. A *data structure* $\mathtt{DS}$ 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 set of booleans). For some method names, the input or output value is absent from the operation.

Here, we should give an example of a stack trace. And maybe omit the rest of this subsubsection

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 *trace* of $\mathtt{DS}$ is a sequence of operations. The standard (sequential) semantics of a data structure $\mathtt{DS}$ is provided by a set $[\![\mathtt{DS}]\!]$ of allowed traces, called the *legal* traces of $\mathtt{DS}$.

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.

Each method declares local variables and a method body. 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). We assume that the set of local variables include the input parameter of the method in addition to the program counter `pc`. Method execution is terminated by executing a `return` command, which may return a value. 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. 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. 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 skip lists, there is an array of pointer fields named `next[k]` where `k` ranges from 1 to the maximum level of the skip list. 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. 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`. 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., [25]).

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 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 pending call actions. 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 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 $op_i = \mathtt{m}(d_i^{in}, d_i^{out})$, $a_i = \mathtt{call}_{\mathtt{o}_\mathtt{i}}\ \mathtt{m}\left(d_i^{in}\right)$, and $a_i = \mathtt{ret}_{\mathtt{o}_\mathtt{i}}\ \mathtt{m}\left(d_i^{out}\right)$, 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. $\mathtt{DS}$ if the corresponding trace is in $[\![\mathtt{DS}]\!]$. We say that h is *linearizable* wrt. $\mathtt{DS}$ if there is a completed extension of h, which has a linearization that is valid wrt. $\mathtt{DS}$. We say that a program $\mathcal{P}$ is linearizable wrt. $\mathtt{DS}$, in each possible execution, the sequence of call and return actions is *linearizable* wrt. $\mathtt{DS}$.

Should we mention LP method: The most common technique is to annotate methods with LPs. More precisely, each method is instrumented to announce precisely when the linearization point occurs during each method invocation. In many cases, the linearization point can be associated with a particular statement in the method code (so called *fixed linearization points*). Linearizability can then be verified by checking that the occurrence of linearization points is allowed by the sequential semantics of the particular data structure that is implemented.

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. For stacks and queues, it was recently established [5, 19] that the linearizability criterion is equivalent to a small number of simple ordering constraints of the following form

> for one or two *arbitrary* data values, the subsequence of call and return actions with these data values as parameters is in a particular regular set.

The complement of each such constraint can be expressed by an *observer*, as introduced in [1]. Observers are finite automata extended with a finite set of *registers* that assume values in $\mathbb{D}$. At initialization, the registers are nondeterministically assigned arbitrary distinct values, which never change during a run of the observer. Formally, an observer $\mathcal{O}$ is a tuple $\left\langle S^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{init}}, \mathtt{X}^{\mathcal{O}}, \Delta^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{acc}} \right\rangle$ 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 $\left\langle s_1, \mathtt{call}\ \mathtt{m}\left(x^{in}, x^{out}\right), s_2 \right\rangle$ or $\left\langle s_1, \mathtt{ret}\ \mathtt{m}\left(x^{in}, x^{out}\right), s_2 \right\rangle$ where 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 history one action at a time. If there is a transition, whose label (after replacing registers by their values) matches the action, such a transition is performed. If there is no such transition, the observer remains in its current state. The observer accepts a history 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 histories that do *not* satisfy the constraint represented by the observerer. [COMMENT: In Figure XXX we showed observers for the stack]

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

3.4 Formal Semantics

For preciseness, we formalize the semantics of programs and observers. Below, we assume a program $\mathcal{P}$ with a set $\mathtt{X}^{\mathtt{gl}}$ of global variables, and an ordering constraint specified by an observer $\mathcal{O}$. We assume that each thread $\mathtt{th}$ executes one method denoted $\mathtt{Method}\left(\mathtt{th}\right)$.

For a function $f : A \mapsto B$ from a set A to a set B, we use $f[a_1 \leftarrow b_1, \ldots, a_n \leftarrow b_n]$ to denote the function f' such that $f'(a_i) = b_i$ and $f'(a) = f(a)$ if $a \notin \{a_1, \ldots, a_n\}$.

Heaps. A *heap (state)* is a tuple $\mathcal{H} = \left\langle \mathbb{C}, \mathtt{ptr}_1, \ldots, \mathtt{ptr}_k, \mathtt{Val}^{\mathtt{gl}}, \mathtt{Val}^{\mathbb{C}} \right\rangle$, where (i) $\mathbb{C}$ is a finite set of cells, including the two special cells $\mathtt{null}$ and $\perp$ (dangling); we define $\mathbb{C}^- = \mathbb{C} \setminus \{\mathtt{null}, \perp\}$, (ii) for each pointer field $\mathtt{ptr}_i$ there is a total function $\mathtt{ptr}_i : \mathbb{C}^- \to \mathbb{C}$ that defines where that pointer field points, (iii) $\mathtt{Val}^{\mathtt{gl}} : \mathtt{X}^{\mathtt{gl}} \to \mathbb{C}$ maps the global (pointer) variables to their values, and (iv) $\mathtt{Val}^{\mathbb{C}} : \mathbb{C} \times \mathcal{F} \to \mathbb{F} \cup \mathbb{D}$ maps data and lock fields of each cell to their values. We let $\mathcal{H}_{\mathtt{init}}$ denote the initial heap produced by the `init` method.

Threads. A *local state* $\mathtt{loc}$ of a thread $\mathtt{th}$ wrt. a heap $\mathcal{H}$ defines the values of its local variables, including the program counter `pc` and the input parameter for the method executed by $\mathtt{th}$. In addition, there is the special initial state $\mathtt{idle}$, and terminated state $\mathtt{term}$. We formalize the behavior of a thread $\mathtt{th}$ by 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 $\mathcal{H}$, with three types of transitions:

1. $\langle \mathtt{idle}, \mathcal{H} \rangle \xrightarrow{\mathtt{call}_{\mathtt{th}}\ \mathtt{m}\left(d^{in}\right)}_{\mathtt{th}} \langle \mathtt{loc}^{\mathtt{th}}_{\mathtt{init}}, \mathcal{H} \rangle$ activates thread $\mathtt{th}$ through a transition labeled by a call action with $\mathtt{th}$ as the action identifier and $\mathtt{m} = \mathtt{Method}\,(\mathtt{th})$, taking $\mathtt{th}$ to an initial local state $\mathtt{loc}^{\mathtt{th}}_{\mathtt{init}}$ where d^{in} is the value of its input parameter, the value of `pc` is the label of the first statement of the method, and the other local variables are undefined.
2. $\langle \mathtt{loc}, \mathcal{H} \rangle \rightarrow_{\mathtt{th}} \langle \mathtt{loc}', \mathcal{H}' \rangle$ denotes execution of method statements, which are unlabeled; these are defined in the standard way for each statement form.
3. $\langle \mathtt{loc}, \mathcal{H} \rangle \xrightarrow{\mathtt{ret}_{\mathtt{th}}\ \mathtt{m}\left(d^{out}\right)}_{\mathtt{th}} \langle \mathtt{term}, \mathcal{H} \rangle$ terminates thread $\mathtt{th}$ through execution of its `return` command, labeled by a return action with $\mathtt{th}$ as action identifier, $\mathtt{m} = \mathtt{Method}\,(\mathtt{th})$, and d^{out} as the returned value.

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 $\mathtt{LOC}\,(\mathtt{th})$ wrt. $\mathcal{H}$. The initial configuration $c^{\mathcal{P}}_{\mathtt{init}}$ is the pair $\langle \mathtt{LOC}_{\mathtt{init}}, \mathcal{H}_{\mathtt{init}} \rangle$, where $\mathtt{LOC}_{\mathtt{init}}\,(\mathtt{th}) = \mathtt{idle}$ for each $\mathtt{th} \in \mathtt{T}$, A program $\mathcal{P}$ induces 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 $\langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle \xrightarrow{\varepsilon}_{\mathcal{P}} \langle \mathtt{T}, \mathtt{LOC}[\mathtt{th} \leftarrow \mathtt{loc}'], \mathcal{H}' \rangle$ whenever the transition relation $\rightarrow_{\mathtt{th}}$ has a transition $\langle \mathtt{loc}, \mathcal{H} \rangle \xrightarrow{\lambda}_{\mathtt{th}} \langle \mathtt{loc}', \mathcal{H}' \rangle$, where the label λ is either a call or return action or the empty label. Note that the only visible transitions are those corresponding to call and return actions.

Cross-Product of Program and Observer We use $\mathcal{S} = \mathcal{P} \otimes \mathcal{O}$ to denote the cross-product obtained by running $\mathcal{P}$ and $\mathcal{O}$ together. The initial configuration of $\mathcal{S}$ is $\left\langle c^{\mathcal{P}}_{\mathtt{init}}, s^{\mathcal{O}}_{\mathtt{init}} \right\rangle$. 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 $c^{\mathcal{P}} \xrightarrow{l}_{\mathcal{P}} c^{\mathcal{P}'}$ of the program with some (possibly empty) label l, and a corresponding synchronizing transition $\langle s, \mathtt{ret}\ \mathtt{m}\,(l)\,, s' \rangle$ of the monitor. The verification problem is now to check that $\mathcal{S}$ cannot reach a configuration $\left\langle c^{\mathcal{P}}, s^{\mathcal{O}}_{\mathsf{acc}} \right\rangle$ with an accepting observer state.

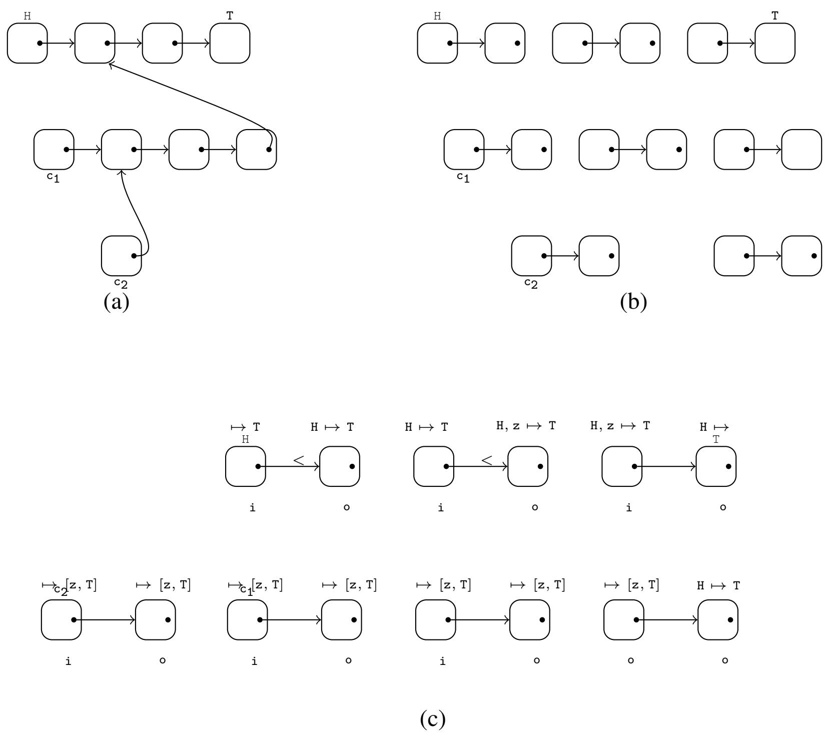


Fig. 8: Example of Fragment Abstraction

4 Fragment Abstraction: Singly Linked Lists

In this section, we describe in more detail our fragment abstraction for concurrent programs that operate on a shared heap. We consider a program with global variables $\mathtt{X}^{\mathtt{gl}}$ and thread-local variables $\mathtt{X}^{\mathtt{loc}}$. We assume that all global variables are pointer variables. We describe our fragment abstractions for three classes of heap structures: in the following subsection, we consider programs operating on singly-linked lists, in Subsection **??**, we consider programs operating on skiplists, and in Subsection **??**, we consider programs operating arrays of singly linked lists.

4.1 Fragment Abstraction for Singly-Linked List-Based Programs

In this subsection, we describe our symbolic representation, 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 **??**.

[COMMENT: Say that for now, we ignore timers?]

We assume that we must analyze the product of a program and an observer. The program operates on a heap, where each cell has exactly one pointer field, named `next`, and at most one `data` field, which assumes values from the same domain $\mathbb{D}$ as observer registers.

[COMMENT: Introduce notation for the set of local $\mathbb{D}$-variables?. Maybe defined which threads we talk about? For now, we skip timestamps.]

We first define our *data abstraction*. For each thread-local variable, and each non-pointer cell field, which ranges over some concrete domain, we define a corresponding abstract domain, as follows.

- For small concrete domains (including that of the program counter), the abstract domain is the same as the concrete one.
- For locks variables and lock fields, the abstract domain is $\{me, other, free\}$. [COMMENT: Should we explain?]
- For the concrete domain $\mathbb{D}$ of data values, the abstract domain is the set of mappings from local variables ranging over $\mathbb{D}$ and observer registers to the set $\{<, =, >\}$. An element in the abstract domain represents a concrete data value $\mathtt{d}$ if it maps each local variable and observer register with value $\mathtt{d}'$ to a set which includes a relation $\sim$ such that $\mathtt{d} \sim \mathtt{d}'$.

Define a *local symbolic configuration* as a mapping from local variables (including the program counter) to their corresponding abstract domains. We use $c \models^{loc}_{\mathtt{th}} \sigma$ to denote that in the global configuration c, the local configuration of thread $\mathtt{th}$ satisfies the local symbolic configuration σ, defined in the natural way.

As partial motivation for the definition of tags, we observe that in order to verify ordering properties represented by the observer, e.g., that data elements are popped in LIFO order, it is necessary to track the relative positions of heap cells whose `data` field has the same value as some observer register. Thus, in a given configuration, for an observer register $\mathtt{x}_i$, define a $\mathtt{x}_i$*-cell* to be a heap cell whose `data` has the same value as $\mathtt{x}_i$. [COMMENT: Move this paragraph?]

Next, define a *tag* as a tuple $\texttt{tag} = \langle \texttt{dabs}, \texttt{pvars}, \texttt{reachfrom}, \texttt{reachto}, \texttt{private} \rangle$, where

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

Assume some global configuration c. We say that a heap cell $\mathbb{c}$ be *accessible* to a thread `th` if $\mathbb{c}$ is reachable (directly or via sequence of next-pointers) from a global pointer variable or local pointer variable of `th`. For a cell $\mathbb{c}$ which is accessible to thread `th`, and a tag $\texttt{tag} = \langle \texttt{dabs}, \texttt{pvars}, \texttt{reachfrom}, \texttt{reachto}, \texttt{private} \rangle$, we write $\mathbb{c} \lhd_{\texttt{th}} \texttt{tag}$ to denote that

- `dabs` represents the concrete values of the non-pointer fields of $\mathbb{c}$,
- `pvars` is the set of global pointer variables and local pointer variables of `th` that point to $\mathbb{c}$,
- `reachfrom` is the set of (i) global pointer variables from which $\mathbb{c}$ is reachable via a (possibly empty) sequence of `next` pointers, and (ii) observer registers $\texttt{x}_i$ such that $\mathbb{c}$ is reachable from some $\texttt{x}_i$-cell.
- `reachto` is the set of (i) global pointer variables that point to a cell which is reachable from $\mathbb{c}$, and (ii) observer registers $\texttt{x}_i$ such that some $\texttt{x}_i$-cell is reachable from $\mathbb{c}$.
- `private` is `true` if $\mathbb{c}$ has never been published (by assigning a heap pointer-field or global pointer variable) by its creating thread `th`.

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

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

Let V be a set of fragments. A global configuration satisfies a set V of fragments wrp. to `th`, denoted $c \models_{\texttt{th}}^{heap} V$, if for any cell $\mathbb{c}$ that is accessible to `th`, there is a fragment $\texttt{v} \in V$ such that $\mathbb{c} \lhd_{\texttt{th}} \texttt{v}$. For a local symbolic configuration σ and set V of fragments, we write $c \models_{\texttt{th}} \langle \sigma, V \rangle$ to denote that that $c \models_{\texttt{th}}^{loc} \sigma$ and $c \models_{\texttt{th}}^{heap} V$.

A *symbolic representation* Ψ is a mapping from local symbolic configurations to sets of fragments. A configuration c of a program satisfies a symbolic representation if for each thread `th`, the domain of Ψ contains a local symbolic configuration σ such that $c \models_{\texttt{th}} \langle \sigma, \Psi(\sigma) \rangle$.

Skip the following example. Do we need one?

Example: Let us show an example of how a singly linked list (SLL) is split into a set of fragments. Fig. 8(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. 8(b) shows the result of splitting the list in Fig. 8(a) into fragments where each heap fragment is a small list of two nodes. Fig. 8(c) shows the abstraction of fragments in Fig. 8(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.

Computing Postconditions In the verification, we must compute a symbolic representation that is satisfied by all reachable program configurations. 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. In this subsection, we describe the symbolic postcondition computation, which is the key step in this procedure.

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. More precisely, assume that a global configuration c satisfies a symbolic representation Ψ. Let `th` be an arbitrary thread. Assume that there is a local symbolic configuration $\sigma \in Dom(\Psi)$ such that $c \models_{\mathtt{th}} \langle \sigma, \Psi(\sigma) \rangle$. We must ensure that this property still hols after any execution of a statement by some thread. In the thread-modular approach, we must consider two cases:

- *Local Steps:* The thread `th` itself executes some statement, which may change its local state and the state of the heap. In this case, we compute a local symbolic configuration σ' and set V' such that the resulting configuration c' satisfies $c' \models_{\mathtt{th}} \langle \sigma', V' \rangle$, and (if necessary) extend Ψ so that $\sigma' \in Dom(\Psi)$ and $V' \in \Psi(\sigma')$.
- *Interference Steps:* Another thread $\mathtt{th}_2$, which satisfies a local symbolic configuration σ_2 in $Dom(\Psi)$ with $c \models_{\mathtt{th}_2} \langle \sigma_2, \Psi(\sigma_2) \rangle$ performs a computation step, which affects the state of the heap in such a way that makes it necessary to extend $\Psi(\sigma)$. We must then compute a set V' of fragments such that the resulting configuration c' satisfies $c' \models_{\mathtt{th}}^{heap} V'$ and make sure that $V' \in \Psi(\sigma)$. 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 of $\mathtt{th}_2$, as in the local case, finally project back the set of fragments 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 $\sigma \in Dom(\Psi)$ with For each statement that th can execute in a configuration c with $c \models_{\texttt{th}} \langle \sigma, \Psi(\sigma) \rangle$, we must compute a local symbolic configuration σ' and a set V' of fragments such that such that the resulting configuration c' satisfies $c' \models_{\texttt{th}} \langle \sigma', V' \rangle$. 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 and data abstraction part of fragments. We therefore here describe how to compute the effect of statements that update pointer variables or the heap, since these are the most interesting cases.

The main difficulty in the postcondition computation is to update the reachability information provided in the fields reachfrom and reachto in each tag of a fragment. For instance, considerthat a statement g := p, which assigns the value of a local pointer variable p to a global pointer variable g. In the postcondition computation, we must for each fragment determine how to update the field reachfrom in its tags, and in particular whether g should be in this set after the statement (the same problem occurs for the set reachto), If v would have been a global variable, this information could be obtained by checking whether v is in the set before the operation. However, since the reachfrom field does not include local variables, we start the postcondition computation by computing a number of transitive-closure-like relations between fragments, which will allow to determine whether g should be in the reachfrom field after the statement with rather good accuracy. Note that if our procedure can not determine whether g should be in a reachfrom field, then it generates fragments for both possibilities.

First, we say that two tags $\texttt{tag} = \langle \texttt{dabs}, \texttt{pvars}, \texttt{reachfrom}, \texttt{reachto}, \texttt{private} \rangle$ and $\texttt{tag}' = \langle \texttt{dabs}', \texttt{pvars}', \texttt{reachfrom}', \texttt{reachto}', \texttt{private}' \rangle$ are *consistent* if if there is some concrete valuation of non-pointer fields represented by both dabs and $\texttt{dabs}'$, and if $\texttt{pvars} = \texttt{pvars}'$, $\texttt{reachfrom} = \texttt{reachfrom}'$, $\texttt{reachto} = \texttt{reachto}'$, and $\texttt{private} = \texttt{private}'$. Intuitively, tag and $\texttt{tag}'$ are consistent if there can exist a cell $\mathbb{c}$ accessible to th with $\mathbb{c} \lhd_{\texttt{th}} \texttt{tag}$ and $\mathbb{c} \lhd_{\texttt{th}} \texttt{tag}'$.

To Quy: Can you really require that the data abstrations must be the same?

Let $\texttt{v}_1$ and $\texttt{v}_2$ be two fragments in a set V of fragments.

- Let $\texttt{v}_1 \hookrightarrow_V \texttt{v}_2$ denote that $\texttt{v}_1\texttt{.o}$ and $\texttt{v}_2\texttt{.i}$ are consistent.
- Let $\texttt{v}_1 \leftrightarrow_V \texttt{v}_2$ denote that $\texttt{v}_1\texttt{.o} = \texttt{v}_2\texttt{.o}$ are consistent, and that either $\texttt{v}_1\texttt{.i.pvars} \cap \texttt{v}_2\texttt{.i.pvars} = \emptyset$ or that the global variables in $\texttt{v}_1\texttt{.i.reachfrom}$ are disjoint from those in $\texttt{v}_2\texttt{.i.reachfrom}$.

  Question: Is this correct?

Intuitively, $\texttt{v}_1 \hookrightarrow_V \texttt{v}_2$ denotes that it is possible that $\texttt{next}(\mathbb{c}_1) = \mathbb{c}_2$ for some cells with $\mathbb{c}_1 \lhd \texttt{v}_1$ and $\mathbb{c}_2 \lhd \texttt{v}_2$. Intuitively, $\texttt{v}_1 \leftrightarrow_V \texttt{v}_2$ denotes that it is possible that $\texttt{next}(\mathbb{c}_1) = \texttt{next}(\mathbb{c}_2)$. for different cells $\mathbb{c}_1$ and $\mathbb{c}_2$ with $\mathbb{c}_1 \lhd \texttt{v}_1$ and $\mathbb{c}_2 \lhd \texttt{v}_2$. Note that the above definitions also work for the cases that the output tag is null or $\perp$.

We use the above relations to define several derived relations:

- Let $\overset{+}{\hookrightarrow}_V$ denote the transitive closure of $\hookrightarrow_V$, and $\overset{*}{\hookrightarrow}_V$ the reflexive transitive closure of $\hookrightarrow_V$.

- Let $\mathtt{v}_1 \overset{**}{\leftrightarrow}_V \mathtt{v}_2$ denote that there are $\mathtt{v}_1', \mathtt{v}_2' \in V$ with $\mathtt{v}_1' \leftrightarrow_V \mathtt{v}_2'$ such that $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}_1'$ and $\mathtt{v}_2 \overset{*}{\hookrightarrow}_V \mathtt{v}_2'$.
- Let $\mathtt{v}_1 \overset{*+}{\leftrightarrow}_V \mathtt{v}_2$ denote that there are $\mathtt{v}_1', \mathtt{v}_2' \in V$ with $\mathtt{v}_1' \leftrightarrow_V \mathtt{v}_2'$ such that $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}_1'$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}_2'$.
- Let $\mathtt{v}_1 \overset{*\circ}{\leftrightarrow}_V \mathtt{v}_2$ denote that is a $\mathtt{v}_1' \in V$ with $\mathtt{v}_1' \leftrightarrow_V \mathtt{v}_2$ such that $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}_1'$.
- Let $\mathtt{v}_1 \overset{++}{\leftrightarrow}_V \mathtt{v}_2$ denote that there are $\mathtt{v}_1', \mathtt{v}_2' \in V$ with $\mathtt{v}_1' \leftrightarrow_V \mathtt{v}_2'$ such that $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}_1'$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}_2'$.
- Let $\mathtt{v}_1 \overset{+*}{\leftrightarrow}_V \mathtt{v}_2$ denote that there are $\mathtt{v}_1', \mathtt{v}_2' \in V$ with $\mathtt{v}_1' \leftrightarrow_V \mathtt{v}_2'$ such that $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}_1'$ and $\mathtt{v}_2 \overset{*}{\hookrightarrow}_V \mathtt{v}_2'$.
- Let $\mathtt{v}_1 \overset{+\circ}{\leftrightarrow}_V \mathtt{v}_2$ denote that is a $\mathtt{v}_1' \in V$ with $\mathtt{v}_1' \leftrightarrow_V \mathtt{v}_2$ such that $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}_1'$.

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, this means that $\mathtt{v}_1$ and $\mathtt{v}_2$ can be satisfied by two cells in the same heap state. Figure 9 illustrates the above relations for a heap state with 13 heap cells. The figure

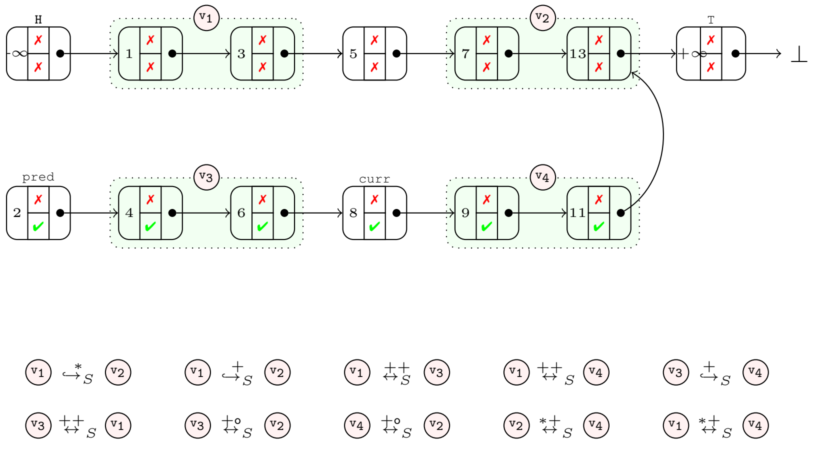


Fig. 9: Illustration of some transitive-closre-like relations between fragments

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 $\mathtt{x}$ and $\mathtt{y}$ are global or local (to thread $\mathtt{th}$) pointer variables. We must compute a set V' of fragments which are satisfied by the configuration after the statement. We must ensure that any cell $\mathbb{c}$ which is accessible to $\mathtt{th}$ after the statement satisfies some fragement in V'. The cell $\mathbb{c}$ must satisfy some fragment $\mathtt{v}$ in V, and must be in the same heap state as the cell pointed to by $\mathtt{y}$. This means that we can make a case analysis on the possible relationships between $\mathtt{v}$ and any fragment $\mathtt{v}_y \in V$ such that $\mathtt{y} \in \mathtt{v}_y.\mathtt{i.pvars}$. 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 $\mathtt{x}$ 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 $\mathtt{v}$ with $\mathtt{v} \hookrightarrow_V \mathtt{v}_y$, let V' contain $\mathtt{v}'$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
   - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \cup \{x\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \cup \{x\}$ if $\mathtt{x}$ is a global variable,
3. We perform analogous inclusions for fragments $\mathtt{v}$ 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 $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - $\mathtt{v}'.\mathtt{i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachfrom} = \mathtt{v.o.reachfrom} \setminus \{x\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$ if $\mathtt{x}$ 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 $\mathtt{x}$ and $\mathtt{y}$. 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 $\mathtt{v}$ which is in $\mathtt{v}$. This is done by an exhaustive case analysis on the possible relationship between $\mathtt{v}$, $\mathtt{v}_x$ 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 $\texttt{regset}$ of observer registers in $\texttt{v.i.reachfrom} \cap \texttt{v}_\texttt{x}\texttt{.i.reachfrom}$, we first create a fragment $\texttt{v}'$ which is same as $\texttt{v}$, except that $\texttt{v}'\texttt{.i.reachfrom} = (\texttt{v.i.reachfrom} \setminus \texttt{v}_\texttt{x}\texttt{.i.reachfrom}) \cup \texttt{regset}$.

(b) Thereafter, for each set $\texttt{regset}'$ of observer registers in $\texttt{v}'\texttt{.o.reachfrom} \cap \texttt{v}_x\texttt{.i.reachfrom}$, we let V' contain a fragment $\texttt{v}''$ which is same as $\texttt{v}'$, except that $\texttt{v}''\texttt{.o.reachfrom} = (\texttt{v}'\texttt{.o.reachfrom} \setminus \texttt{v}_x\texttt{.i.reachfrom}) \cup \texttt{regset}'$.

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 $c \models^{heap}_{\texttt{th}_i} V_i$ for $i = 1, 2$. In order to distinguish between local variables of $\texttt{th}_1$ and $\texttt{th}_2$, we assume that local variable $\texttt{x}$ of thread $\texttt{th}_i$ is named as $\texttt{x}[\texttt{i}]$. We must compute a set V which for each heap cell accessible to either $\texttt{th}_1$ or $\texttt{th}_2$, the set V must contain a fragment $\texttt{v}$ with $\mathbb{c} \lhd_{1,2} \texttt{v}$. [COMMENT: This notation to be defined] There are here two possibilities.

– If $\mathbb{c}$ is accessible to both $\texttt{th}_1$ and $\texttt{th}_2$, then there are fragments $\texttt{v}_1 \in V_1$ and $\texttt{v}_2 \in V_2$ such that $\mathbb{c} \lhd_1 \texttt{v}_1$ and $\mathbb{c} \lhd_2 \texttt{v}_2$. We use the notation $\texttt{v}_1 \sqcap \texttt{v}_2$ to denote a set of views such that whenever $\mathbb{c} \lhd_1 \texttt{v}_1$ and $\mathbb{c} \lhd_2 \texttt{v}_2$ then $\mathbb{c} \lhd_{1,2} \texttt{v}$ for some $\texttt{v} \in (\texttt{v}_1 \sqcap \texttt{v}_2)$.
– If $\mathbb{c}$ is accessible to only one of $\texttt{th}_1$ and $\texttt{th}_2$, say $\texttt{th}_1$, then V should contain some fragment $\texttt{v}_1 \in V_1$ with $\mathbb{c} \lhd_{1,2} \texttt{v}_1$ [COMMENT: Check that we need not change $\texttt{v}_1$]

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

Question to Quy: You have also used the notation $\texttt{v.i.gdata}$. 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 $\texttt{v}_1 \sqcap \texttt{v}_2$ for two views $\texttt{v}_1 \in V_1$ and $\texttt{v}_2 \in V_2$. Firstly, we consider the case where both $\texttt{v}_1$ and $\texttt{v}_2$ have size 2. Let us consider some different cases. They all take into account the observation that if a cell $\mathbb{c}$ satisfies $\mathbb{c} \lhd_1 \texttt{v}_1$ and $\mathbb{c} \lhd_2 \texttt{v}_2$, then the information about global variables in $\texttt{v}_1$ and $\texttt{v}_2$ must coincide.

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

        * $\mathtt{v_{12}.o.reachfrom} = \mathtt{v_1.o.reachfrom} \cup \mathtt{v_2.o.reachfrom}$

          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 $\mathtt{v_1.i.greachfrom} = \emptyset$, $\mathtt{v_2.i.greachfrom} = \emptyset$, $\mathtt{v_1.o.greachfrom} \neq \emptyset$ and $\mathtt{v_2.o.greachfrom} \neq \emptyset$ then
    - if $\mathtt{v_1.o.gtag} = \mathtt{v_2.o.gtag}$, $\mathtt{v_1.i.private} = \mathtt{false}$ and $\mathtt{v_2.i.private} = \mathtt{false}$ 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
            · $\mathtt{v'_1.o.pvars} = \mathtt{v_1.o.pvars} \cup \mathtt{v_2.o.pvars}$
            · $\mathtt{v'_1.o.reachfrom} = \mathtt{v_1.o.reachfrom} \cup \mathtt{v_2.o.reachfrom}$
        * $\mathtt{v'_2}$ is same as $\mathtt{v_2}$ except that
            · $\mathtt{v'_2.o.pvars} = \mathtt{v_1.o.pvars} \cup \mathtt{v_2.o.pvars}$
            · $\mathtt{v'_2.o.reachfrom} = \mathtt{v_1.o.reachfrom} \cup \mathtt{v_2.o.reachfrom}$
        * $\mathtt{v_{12}}$ is as in the previous case. [COMMENT: to Quy: I added this, is it correct?]
    - if $\mathtt{v_1.o.gtag} = \mathtt{v_2.o.gtag}$ and $\mathtt{v_1.i.private} = \mathtt{true}$ or $\mathtt{v_2.i.private} = \mathtt{true}$ 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 $\mathtt{v_1.i.greachfrom} = \emptyset$, $\mathtt{v_2.i.greachfrom} = \emptyset$, $\mathtt{v_1.o.greachfrom} = \emptyset$ and $\mathtt{v_2.o.greachfrom} = \emptyset$ then
    - if $\mathtt{gtag}(\mathtt{v_1}, \mathtt{o}) = \mathtt{gtag}(\mathtt{v_2}, \mathtt{o})$, $\mathtt{v_1.i.private} = false$, $\mathtt{v_1.o.private} = false$, $\mathtt{v_1.o.private} = false$ and $\mathtt{v_2.o.private} = false$ then $\mathtt{v_1} \sqcap \mathtt{v_2} = \{\mathtt{v_1}, \mathtt{v_2}, \mathtt{v'_1}, \mathtt{v'_2}, \mathtt{v_{12}}\}$
    - if $\mathtt{gtag}(\mathtt{v_1}, \mathtt{o}) = \mathtt{gtag}(\mathtt{v_2}, \mathtt{o})$, ($\mathtt{v_1.i.private} = true$ or $\mathtt{v_2.i.private} = true$) and $\mathtt{v_1.o.private} = false$ and $\mathtt{v_2.o.private} = false$ 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 ($\mathtt{v_1.o.private} = true$ or $\mathtt{v_1.o.private} = true$) 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.

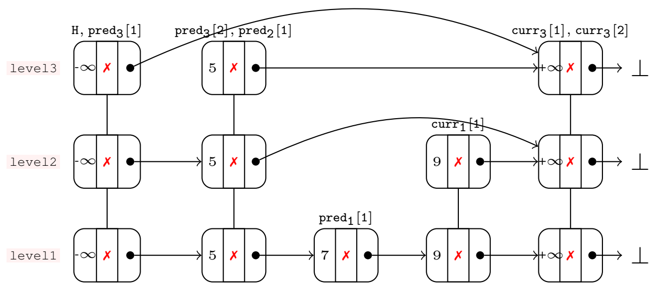
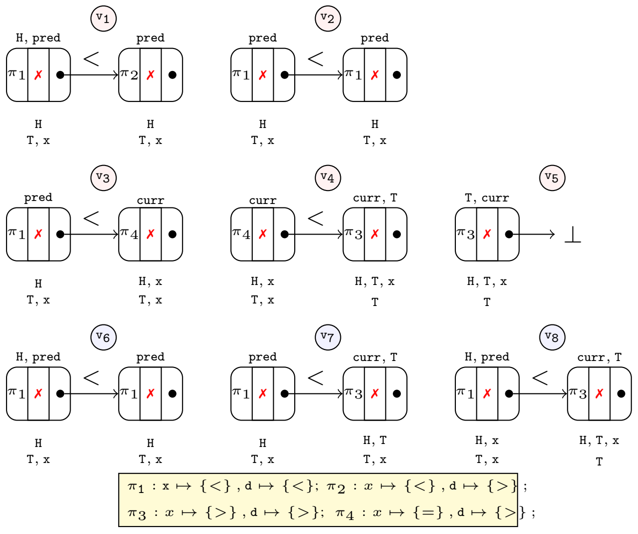


Fig. 10: A concrete shape of 3-level skipl-list with two threads



π_1 : x $\mapsto$ {<} , d $\mapsto$ {<}; π_2 : x $\mapsto$ {<} , d $\mapsto$ {>} ;
π_3 : x $\mapsto$ {>} , d $\mapsto$ {>}; π_4 : x $\mapsto$ {=} , d $\mapsto$ {>} ;

Fig. 11: skipl-list fragments [ANSWER of Quy: i am working with this figure]

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 $\mathtt{next(i)} = \perp$. 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 $\mathtt{y} \in \mathtt{v_y.i.pvars}$,
  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
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
     - if
       $gvarofx$ 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 $\mathtt{R}$
  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
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
     - if
       $gvarofx$ 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 $\mathtt{R}$
  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
     - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
     - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
     - if
       $gvarofx$ 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 V'

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

   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 $\mathtt{v_y} \overset{*\circ}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$, create $\mathtt{v'}$ which is same as $\mathtt{v}$ except that
   - $\mathtt{v'.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v'.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - if
     $gvarofx$ is a global variable
     * $\mathtt{v'.i.reachfrom} = \mathtt{v.i.reachfrom} \setminus \{x\}$,
     * $\mathtt{v'.i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$,
     * $\mathtt{v'.o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
     * $\mathtt{v'.o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$,

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

   then add $\mathtt{v'}$ to V_{post}, and $(\mathtt{v}, \mathtt{v'})$ to 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 $\mathtt{i_1} \in \{\mathtt{v_1.i}, \mathtt{v_1.o}\}$, $\mathtt{i_2} \in \{\mathtt{v_2.i}, \mathtt{v_2.o}\}$, $\mathtt{tag}(\mathtt{v}, \mathtt{i}) = \mathtt{tag}(\mathtt{v_1}, \mathtt{i_1})$, 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
   - $\mathtt{v'.i.pvars} = \mathtt{v'_1}.i_1.vars$,
   - $\mathtt{v'.o.pvars} = \mathtt{v'_2}.i_2.vars$,
   - $\mathtt{v'.i.reachfrom} = \mathtt{v'_1}.i_1.reachfrom$,
   - $\mathtt{v'.o.reachfrom} = \mathtt{v'_2}.i_2.reachfrom$,
   - $\mathtt{v'.i.reachto} = \mathtt{v'_1}.i_1.reachto$,
   - $\mathtt{v'.o.reachto} = \mathtt{v'_2}.i_2.reachto$,

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

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
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
   - $\mathtt{v}'\mathtt{.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'\mathtt{.o.pvars} = \mathtt{v.o.pvars} \cup \{x\}$,
   - 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 $\mathtt{R}$
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
   - $\mathtt{v}'\mathtt{.i.pvars} = \mathtt{v.i.pvars} \setminus \{x\}$,
   - $\mathtt{v}'\mathtt{.o.pvars} = \mathtt{v.o.pvars} \setminus \{x\}$,
   - if $\mathtt{x}$ is a global variable
     - $\mathtt{v}'\mathtt{.i.reachfrom} = \mathtt{v.i.reachfrom} \cup \{x\}$,
     - $\mathtt{v}'\mathtt{.i.reachto} = \mathtt{v.i.reachto} \setminus \{x\}$,
     - $\mathtt{v}'\mathtt{.o.reachfrom} = \mathtt{v.o.reachfrom} \cup \{x\}$,
     - $\mathtt{v}'\mathtt{.o.reachto} = \mathtt{v.o.reachto} \setminus \{x\}$,

   then add $\mathtt{v}'$ to V_{post}, and $(\mathtt{v}, \mathtt{v}')$ to $\mathtt{R}$
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 $\mathtt{v}_\mathtt{y} \overset{*\top}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$, 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 $\mathtt{v} \overset{\top\circ}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}_\mathtt{y}$, 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
     * $\mathtt{v'.i.reachto} = (\mathtt{v.i.reachto} \cap \mathtt{v_x.i.reachfrom}) \cup \mathtt{reachtov_y.i} \cup \mathtt{regset}$.
     * $\mathtt{v'.o.reachto} = (\mathtt{v.o.reachto} \cap \mathtt{v_x.i.reachfrom}) \cup \mathtt{reachtov_y.i} \cup \mathtt{regset}$.
   - 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
     * $\mathtt{v'.i.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.i.reachfrom}$,
     * $\mathtt{v'.o.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.o.reachfrom}$,
   - 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 $\mathtt{v_y} \overset{*\circ}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$,
   - create $\mathtt{v}'$ which is same as $\mathtt{v}$ except that $\mathtt{v'.o.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.o.reachfrom}$,
   - 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 $\mathtt{v_y} \overset{*\circ}{\leftrightarrow}_{\mathtt{V}} \mathtt{v}$,
   - create $\mathtt{v}'$ which is same as $\mathtt{v}$ then except that $\mathtt{v'.o.reachfrom} = \mathtt{v_x.i.reachfrom} \cup \mathtt{v.o.reachfrom}$,
   - for each subset $\mathtt{regset}$ of observer registers in $\mathtt{v'.i.reachfrom} \cap \mathtt{v}_x\mathtt{.i.reachfrom}$
     * create $\mathtt{v}''$ which is same as $\mathtt{v}'$, except that $\mathtt{v''.i.reachfrom} = (\mathtt{v'.i.reachfrom} \setminus \mathtt{v}_x\mathtt{.i.reachfrom}) \cup \mathtt{regset}$.
     * 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 $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = (\mathtt{v}.\mathtt{i}.\mathtt{reachfrom} \setminus \mathtt{v}_{\mathtt{x}}.\mathtt{i}.\mathtt{reachfrom}) \cup \mathtt{regset}$.
- for each set $\mathtt{regset}'$ of observer registers in $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} \cap \mathtt{v}_x.\mathtt{i}.\mathtt{reachfrom}$,
  - create $\mathtt{v}''$ which is same as $\mathtt{v}'$, except that $\mathtt{v}''.\mathtt{o}.\mathtt{reachfrom} = (\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} \setminus \mathtt{v}_x.\mathtt{i}.\mathtt{reachfrom}) \cup \mathtt{regset}'$.
  - add $\mathtt{v}''$ to V_{post}
  - add $(\mathtt{v}, \mathtt{v}'')$ to $\mathtt{R}_6$

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

– for each fragment $\mathtt{v} \in V_2$ then we do as follows. for each $(\mathtt{v}_1, \mathtt{v}_1') \in \mathtt{R}_\mathtt{i}$, $(\mathtt{v}_2, \mathtt{v}_2') \in \mathtt{R}_\mathtt{j}$ where $\mathtt{i} \neq \mathtt{j}$ and $\mathtt{1} \leq \mathtt{i}, \mathtt{j} \leq \mathtt{6}$. 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
  - $\mathtt{v}'.\mathtt{i}.\mathtt{pvars} = \mathtt{v}_1'.i_1.vars$,
  - $\mathtt{v}'.\mathtt{o}.\mathtt{pvars} = \mathtt{v}_2'.i_2.vars$,
  - $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}_1'.i_1.reachfrom$,
  - $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}_2'.i_2.reachfrom$,
  - $\mathtt{v}'.\mathtt{i}.\mathtt{reachto} = \mathtt{v}_1'.i_1.reachto$,
  - $\mathtt{v}'.\mathtt{o}.\mathtt{reachto} = \mathtt{v}_2'.i_2.reachto$,

  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

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. 12, 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.

I am working here

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.

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.

Algorithms	Time (s)
TIMESTAMPS	
TS stack [25]	176
TS queue [25]	101
SKIP-LISTS	
Lock-free skip-list [21]	1992
Optimistic skip-list [25]	500
Priority queue skip-list 1 [26]	1320
Priority queue skip-list 2 [26]	599
SINGLY-LINKED LISTS	
Treiber stack [32]	18
MS lock-free queue [25]	21
DGLM queue [11]	16
Vechev-CAS set [39]	86
Vechev-DCAS set [39]	16
Michael lock-free set [23]	178
Pessimistic set [21]	30
Optimistic set [21]	25
Lazy set [17]	34
O′Hearn set [27]	88
HM lock-free set [21]	120

Fig. 12: Experimental results for verifying concurrent programs

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. Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.O.: Proving linearizability using forward simulations. CoRR abs/1702.02705 (2017), `http://arxiv.org/abs/1702.02705`
7. 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)
8. 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)
9. 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`
10. 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)

11. 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)
12. 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)
13. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC'04. pp. 50–59. ACM (2004)
14. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. pp. 300–314 (2001)
15. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: DISC,. pp. 265–279 (2002)
16. 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)
17. 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)
18. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. J. Parallel Distrib. Comput. 70(1), 1–12 (2010)
19. Henzinger, T., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: CONCUR. LNCS, vol. 8052, pp. 242–256 (2013)
20. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
21. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
22. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI. pp. 459–470. ACM (2013)
23. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. pp. 73–82 (2002)
24. 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)
25. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC. pp. 267–275 (1996)
26. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA. pp. 253–262 (2005)
27. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC. pp. 85–94 (2010)
28. 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)
29. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: CAV. LNCS, vol. 7358, pp. 243–259 (2012)
30. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Abstract transformers for thread correlation analysis. In: APLAS. Lecture Notes in Computer Science, vol. 5904, pp. 30–46. Springer (2009)
31. 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)
32. Treiber, R.: Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Res. Ctr. (1986)
33. Turon, A.J., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: POPL '13. pp. 343–356 (2013)
34. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI. LNCS, vol. 5403, pp. 335–348 (2009)
35. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)

36. Vafeiadis, V.: Automatically proving linearizability. In: CAV. LNCS, vol. 6174, pp. 450–464 (2010)
37. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proc. of LICS'86. pp. 332–344 (June 1986)
38. 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)
39. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI. pp. 125–135 (2008)
40. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: SPIN. LNCS, vol. 5578, pp. 261–278 (2009)
41. Zhang, K., Zhao, Y., Yang, Y., Liu, Y., Spear, M.F.: Practical non-blocking unordered lists. In: DISC. pp. 239–253 (2013)
42. Zhu, H., Petri, G., Jagannathan, S.: Poling: SMT aided linearizability proofs. In: CAV. LNCS, vol. 9207, pp. 3–19 (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;
```

```
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;
```

```
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. 13: 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. 14: 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. 15: 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. 16: 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);
```

```
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 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;
```

```
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. 17: 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. 18: SkiplistBased Concurrent Priority Queues.