Fast Concurrent Data Structures Through Timestamping

Dipl. Ing. Andreas Haas, Bakk. techn.

Fachbereich Computerwissenschaften

Paris-Lodron-Universität Salzburg

Dissertation zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

an der Naturwissenschaftlichen Fakultät der
Paris-Lodron-Universität Salzburg
unter der Betreuung von
Univ.-Prof. Dipl.-Inform. Dr.-Ing. Christoph M. Kirsch

Salzburg July 2015

Acknowledgements

I would like to thank my advisor Christoph M. Kirsch and my co-advisor Ana Sokolova for their support and guidance. I would also like to thank all members of the Computational Systems Group, and all my co-authors, for their hard work.

This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF): S11404-N23 and S11411-N23).

Abstract

We present timestamping of concurrent ordered data structures such as stacks, queues, and double-ended queues (deques) to improve their scalability on multicore hardware. The key idea of timestamping is to timestamp elements upon insertion and then use these timestamps to determine the order in which elements are removed. With timestamping, element insertion works thread-locally without any synchronization and is therefore fast. Remove operations are slower because they first have to search for an element and then synchronize on the actual removal. We show how to trade-off the performance of insert and remove operations to optimize overall performance. The idea is to assign the same timestamp to concurrently inserted elements so that these elements can be removed later in parallel by concurrent remove operations. We have developed timestamped versions of a stack, a queue, and a deque, and show in experiments that our implementations are competitive with other state-of-the-art data structure implementations. For example, our TS (timestamped) stack outperforms the Treiber stack, the standard lock-free concurrent stack algorithm, by a factor of 6, and the elimination backoff stack, the previously fastest concurrent stack algorithm, by a factor of 2. Timestamping presents a new challenge for correctness proofs of concurrent data structures. We propose a new proof technique to establish correctness of concurrent stack algorithms and show how to apply it to our TS stack. Additionally, we analyze experimentally how much operations of concurrent queue implementations are reordered between their invocation and the time they take effect. In our experiments, queue implementations with timestamping do not show more reordering than other queue implementations.

Table of contents

Chapter 1

Introduction

We are interested in the design and implementation of concurrent data structures that provide high performance and scalability on multicore hardware. This means that we are interested in executing a high number of operations per time unit on a data structure, and we want this number of operations to scale up with the number of threads which access the data structure concurrently.

In this thesis we are especially interested in the design of concurrent ordered data structures like stacks, queues, and double-ended queues (deques), where the order in which elements are removed depends on the order in which they were inserted. The challenge in the design of ordered data structures is to maintain information about the element which is to be removed next. We call this element the candidate element. For example, the candidate element of a stack is the last pushed element in the stack.

In many implementations of ordered data structures, all information about the candidate element is maintained in the memory layout of the data structure. For

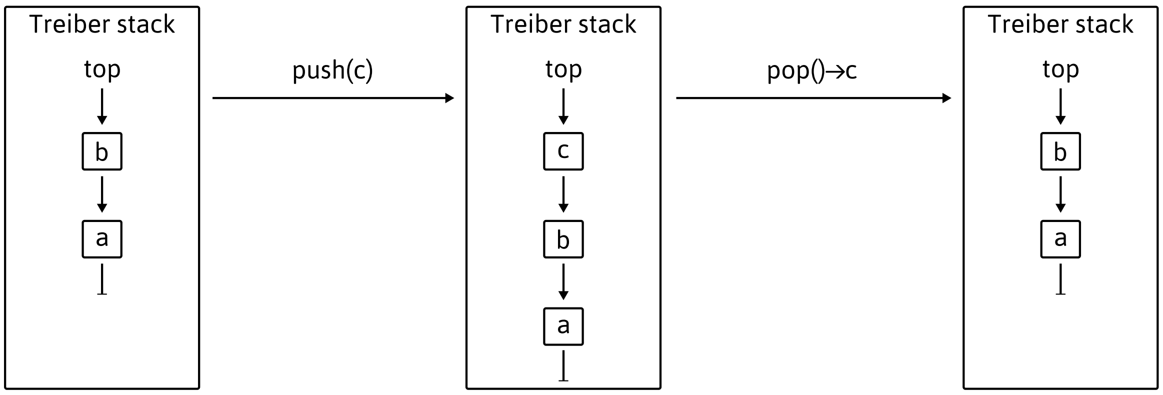


Figure 1.1: Treiber stack [Tre86] push and pop operations.

example, consider the standard lock-free concurrent stack algorithm illustrated in Figure 1.1, the Treiber stack [Tre86]. The Treiber stack maintains a `top` pointer which points to the candidate. Every time a new element is pushed, the new element becomes the candidate element and therefore the `top` pointer is redirected to the new element. Similarly, every time the candidate element is popped, the `top` pointer is redirected to the next candidate element. This means that every operation which accesses the Treiber stack modifies the `top` pointer. When an increasing number of threads accesses the Treiber stack concurrently, the contention on the `top` pointer increases, and eventually the Treiber stack does not scale anymore. In our benchmarks in Section 3.2 the Treiber stack does not scale beyond ten threads.

In this thesis we propose to use timestamping to avoid contention on a single memory location and thereby provide high performance and scalability. The idea is to attach timestamps to elements to maintain information about the next candidate element. Remove operations then iterate over the elements in the data structure and compare their timestamps to identify the candidate element and remove it.

With timestamps we do not have to maintain all information about the candidate element in the memory layout. Therefore we can use a more efficient memory layout and achieve better performance and scalability. However, with timestamping we also have to consider two new sources of overhead: (1) timestamp generation, and (2) identification of a candidate element. We show with our implementations that these sources of overhead can cost less than an inefficient memory layout. For example, in our benchmarks the TS (timestamped) stack [DHK15] is six times faster than the Treiber stack described above, and two times faster than the elimination backoff stack [HSY04], the previously fastest concurrent stack algorithm.

Timestamping also allows fast implementations of concurrent queues and deques. The TS deque [DHK14] is significantly faster than a flat-combining concurrent deque implementation [HIST10], and only the LCRQ queue [AM13] is faster than the TS queue [DHK14] in our experiments.

Implementation. The design of our timestamped data structures is guided by the requirements of the standard correctness condition for concurrent data structures, linearizability [HW90]. Informally, linearizability requires that sequentially executed operations take effect in the order they were executed. Concurrently executed operations, however, can take effect in arbitrary order.

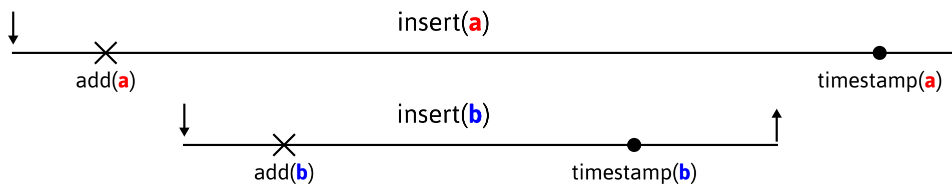


Figure 1.2: An execution of two insert operations, insert(a) and insert(b). Element a is added before element b, but b is timestamped before a. Therefore insert(a) and insert(b) run concurrently.

We show that the definition of linearizability allows to implement operations of timestamped data structures non-atomically. This means that we can add an element to a linked list and in a separate step timestamp it. The reason is illustrated in Figure 1.2, where horizontal lines represent the timeline, $\downarrow$ represents invocations, $\uparrow$ represents responses, $\times$ represents the time when elements are added to a linked list, and $\bullet$ represents the time they get timestamped. If elements are added in a different order than they are timestamped, then their surrounding insert operations executed concurrently. Therefore these insert operations can legitimately take effect in arbitrary order, and both the order in which the elements were added and the order they were timestamped are correct. We can also prove that the identification of a candidate element and its removal can be done in separate steps. Therefore we can optimize element insertion and removal independent from timestamp generation and the identification of candidate elements. Additionally, this allows to provide lock-free implementations, which means that at any point in time at least one thread can access our data structures successfully [HS08].

For fast element insertion and removal we use a memory layout which stores elements in multiple linked lists, one linked list per thread (see Figure 1.3). This means that each thread adds elements into its own linked list. Thereby we can to avoid

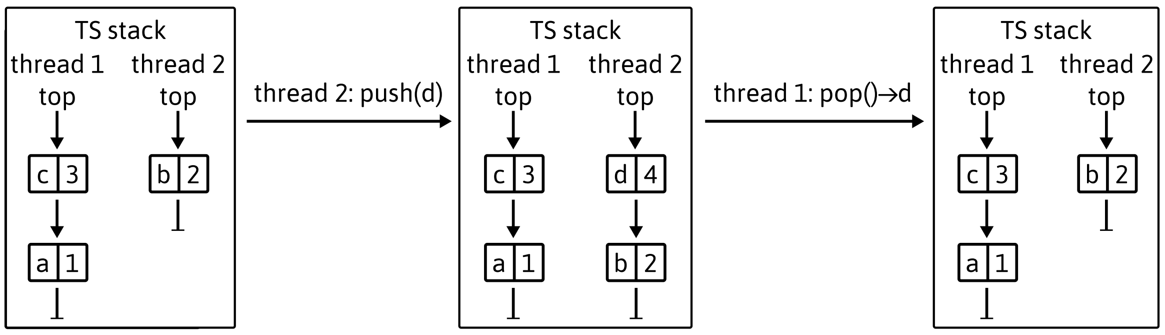


Figure 1.3: TS stack [DHK15] push and pop operations.

two expensive synchronisation patterns in the implementation of insert operations – atomic-write-after-read (AWAR) and read-after-write (RAW). These patterns are described in more details in [AGH+11], and we refer to them collectively as strong synchronisation. By avoiding strong synchronization in the insert operation, we can reduce the costs of insert operations radically. Element removal still requires strong synchronization. However, since elements are stored in multiple linked lists, contention can be distributed over these linked lists, which allows us to improve the performance of remove operations.

Timestamping does not only allow us to store elements in a more efficient memory layout, it also creates new opportunities in the management of candidate elements. As mentioned, linearizability only requires sequentially executed operations to take effect in the order they were executed. We take this requirement of linearizability literally and only guarantee that the timestamps of sequentially inserted elements are ordered. Concurrently inserted elements may get the same timestamp. We say that elements which are not ordered by their timestamps share a timestamp.

An important consequence of timestamp sharing is its effect on the performance of remove operations. Without timestamp sharing, there exists exactly one candidate element at a time. All concurrent remove operations contend on that candidate element, but only one can succeed to remove it. The other remove operations have to try to identify and remove the next candidate element. Similar to the Treiber stack, the contention on the candidate element will prevent the data structure from scaling.

With timestamp sharing there can exist multiple candidate elements at the same time. Concurrent remove operations can remove these candidate elements in parallel. This reduces contention on each candidate element and decreases the number of remove operations which have to retry removing another candidate element. Timestamp sharing can therefore improve the performance of timestamped data structures significantly and is an important aspect in the design of timestamping algorithms.

A natural approach to implement a timestamping algorithm is to maintain a monotonically increasing counter and to generate a timestamp by reading the current value of that counter. In our experiments we use an atomic counter based on the atomic fetch-and-increment instruction, a stuttering counter based on Lamport clocks [Lam78], and a hardware counter which can be read by the `RDTSCP` instruction on x86. All these timestamping algorithms are fast but provide little to no timestamp sharing and are therefore not optimal for remove performance.

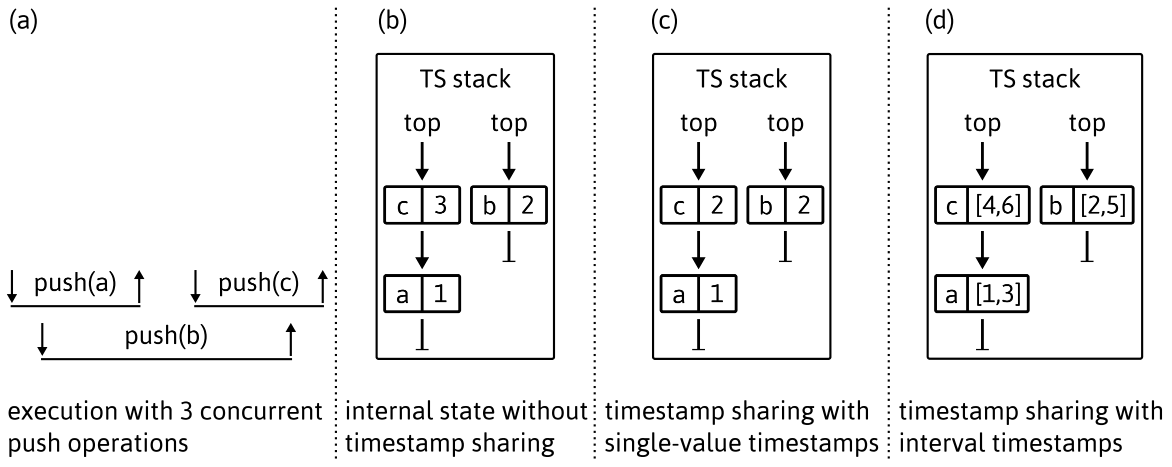

(a)
push(a)
push(c)
push(b)
execution with 3 concurrent push operations
(b)
TS stack
top
top
c 3
b 2
a 1
internal state without timestamp sharing
(c)
TS stack
top
top
c 2
b 2
a 1
timestamp sharing with single-value timestamps
(d)
TS stack
top
top
c [4,6]
b [2,5]
a [1,3]
timestamp sharing with interval timestamps

Figure 1.4: The internal state of the TS stack with and without timestamp sharing.

More timestamp sharing can be achieved with interval timestamps, a contribution of this thesis. Interval timestamps consist of two single-value timestamps, a start timestamp and an end timestamp. With interval timestamps, elements do not only share a timestamp when they have exactly the same timestamp, they also share a timestamp if their interval timestamps overlap.

Figure 1.4 illustrates the potential of timestamp sharing with interval timestamps. In the execution in Figure 1.4 (a) the operations push(a) and push(c) run sequentially, and push(b) runs concurrently to push(a) and push(c). Without timestamp sharing (see Figure 1.4 (b)), all elements are ordered by their timestamps. Element c is the single candidate. With timestamp sharing, b can share a timestamp with both a and c, but a and c cannot share a timestamp. With interval timestamps it is possible to assign an interval timestamp to element b which overlaps with both the interval timestamps of a and c while the interval timestamps of a and c do not overlap (see Figure 1.4 (d)).

As mentioned, timestamp sharing is essential for the performance of remove operations of timestamped data structures. However, when insert operations are too fast, even with interval timestamps a concurrent execution of insert operations is unlikely. As a consequence, few timestamp sharing is happening and overall remove performance may be poor. In this thesis we show how we can trade-off the performance of insert and remove operations by slowing down insert operations and thereby increase overall performance significantly. In our experiments with the TS stack in Section 3.2.2 we show that we can make pop operations 5 times faster by making the push operations

7 times slower. Thereby pop operations become as fast as push operations, which results in the best overall performance in our benchmarks.

We slow down insert operations by adding a busy wait between the generation of the start timestamp and the end timestamp of an interval timestamp. By slowing down insert operations we increase the chance that insert operations execute concurrently and thereby make timestamp sharing more likely. The increased amount of timestamp sharing then improves the performance of remove operations. Note that also other concurrent data structure algorithms use busy waits to improve performance. For example, in our experiments we use a busy wait in the constant-backoff algorithm [DHM13] of the Treiber stack to improve its performance by a factor of 4.

For good remove performance it is also important to reduce the time it takes to search and identify a candidate element. We keep the elements in each linked list sorted by their insertion times. Thereby we can identify candidate elements without iterating over all elements in the linked lists. With sorted lists the costs of identifying a candidate element are not proportional to the number of elements in the data structure but only proportional to the number of linked lists.

For the TS stack we can make the identification of candidate elements even faster by using an optimization called elimination [HSY04]. According to elimination, any element which is pushed concurrently to a pop operation is a candidate element for that pop operation, independent of the timestamp of the element. Therefore, if a pop operation encounters an element which has been inserted during its search for a candidate element, it can stop its search and immediately try to remove that element. With elimination, the search time for a candidate element is reduced significantly. A variant of elimination also works for the TS deque.

We use timestamps to implement elimination. At the beginning of a pop operation we read the current time t. When iterating over the head elements of all linked lists, any element with a timestamp later than t is a candidate element and we can immediately try to remove that element.

Correctness. We prove the correctness of timestamped data structures in terms of linearizability. Intuitively, linearizability is about relating the behavior of a concurrent data structure to a sequential specification. Linearizability allows then to reason about the concurrent data structure as if it was accessed only sequentially according to its specification, and all operations which access the concurrent data structure take effect atomically.

A linearizability proof of a concurrent data structure implementation requires to show that all possible executions of the implementation are linearizable with respect to the sequential specification of the data structure. This means that all operations in an execution satisfy their sequential specification if they are ordered by the time they took effect in the execution, and that an operation always takes effect within its execution time.

In many implementations of concurrent data structures it is obvious when operations take effect. For example, if you consider the Treiber stack, then both the push and the pop operation take effect when they modify the `top` pointer. Therefore the linearizability of the Treiber stack can be proved by showing that whenever the `top` pointer is modified, the surrounding operation changes the state of the Treiber stack according to its sequential specification.

For timestamped data structures it is not obvious when operations take effect. For example, if two elements share a timestamp, then their insert operations can take effect in arbitrary order. The order in which these insert operations actually take effect in an execution depends on the order in which their elements are removed. However, also the order in which elements get removed is unclear. The reason is that remove operations first identify a candidate element and then remove that candidate element in a separate step. Therefore the identified element may not be a candidate element anymore at the time it gets removed. A new element could have been inserted in the meantime. In this thesis we demonstrate how to prove the linearizability of timestamped data structures. We show for the TS stack and the TS queue that there always exists an order in which operations can take effect such that they satisfy their sequential specification. For the TS deque, such a linearizability proof remains future work.

In our linearizability proofs we use an insight published by Henzinger et al. [HSV13]. Henzinger et al. showed that any execution which satisfies a set of queue conditions is linearizable with respect to the sequential specification of a queue. We partition these conditions into two kinds. The first kind of conditions can be summarized by the requirement that the execution has to be linearizable with respect to the sequential specification of a set, i.e. a data structure which allows elements to be removed in arbitrary order. The second kind requires that the execution is order-correct, i.e. that elements are removed in the right order. For a queue, order-correctness means that if an enqueue(a) is executed sequentially before an enqueue(b), then the matching dequeue() $\to b$ is not executed sequentially before dequeue() $\to a$.

The TS queue is order-correct because if an enqueue(a) is executed sequentially before an enqueue(b), then the timestamp of a is older than the timestamp of b. Thus b cannot be a candidate element until a gets dequeued, which means that b cannot be dequeued before a. Therefore the TS queue is order-correctness. In Section 6.2 we prove that the TS queue is also linearizable with respect to the sequential specification of a set.

By showing that the TS queue is order-correct and linearizable with respect to the sequential specification of a set, we show that the TS queue is linearizable with respect to the sequential specification of a queue. Note that the definition of order-correctness for queues only uses information about whether operations are executed sequentially or not. We call this information precedence information.

For the linearizability proof of the TS stack we propose similar conditions for stacks in general, and we show the correctness of these stack conditions with the Isabelle HOL proving assistant [NWP02]. The proof scripts are attached at the end of this thesis. Similar to the queue conditions, these stack conditions also require that executions are linearizable with respect to the sequential specification of a set, and that they are order-correct. However, we present an example in Chapter 4 which indicates that order-correctness for stacks requires more than just precedence information. The reason is that different to order-correctness for queues, order-correctness for stacks also has to consider the order between push and pop operations. This order between push and pop operations cannot be described precise enough with just precedence information.

In our definition of order-correctness we require additional information, called insert-remove information, to describe the order between push and pop operations. We say that an execution is order-correct if whenever a push(a) happens before a push(b) which happens before pop() $\rightarrow a$, then pop() $\rightarrow a$ does not happen before pop() $\rightarrow b$. By happens before we mean a combination of precedence information and insert-remove information which we explain in more detail in Section 4.2.

For the Treiber stack and the TS stack, the insert-remove information can be acquired easily from the algorithm. For the Treiber stack, the insert-remove information comes from the order in which push and pop operations modify the `top` pointer. The Treiber stack is order-correct because if a push(a) inserts its element before a push(b) by modifying the `top` pointer, and b is pushed before a is popped, then b is closer to the head of the linked list than a. Therefore a cannot be popped before b, which satisfies the requirements of order-correctness.

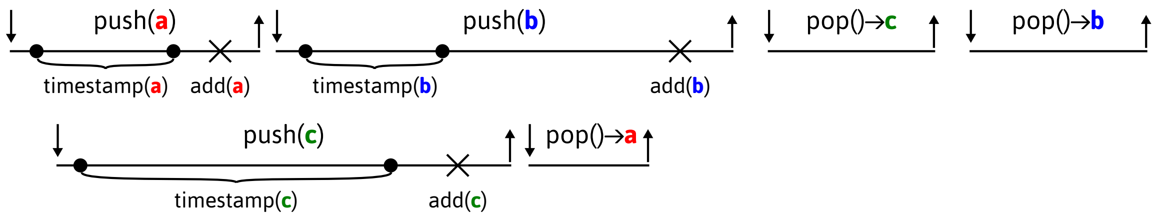


Figure 1.5: An execution of the TS stack which is possible if elements are timestamped before they are added to a linked list. This execution is not linearizable with respect to the sequential specification of a stack.

For the TS stack a push operation is ordered before a pop operation if its element is added to a linked list before the pop operation starts searching for a candidate element. The TS stack is order-correct because if push(a) happens before push(b), then the timestamp of b is younger than the timestamp of a. When b is added to a linked list before pop() $\rightarrow a$ starts searching for a candidate element, then b has to be removed before a can be a candidate element. Therefore a cannot be popped before b, which satisfies the requirements of order-correctness.

Note that in the proof sketch of the TS stack above we say that if push(a) happens before push(b), then the timestamp of b is younger than the timestamp of a. This statement holds because of a small but important detail in the push operation of the TS stack. In the push operation of the TS stack we add elements to the linked lists before we timestamp them. Indeed, if elements are timestamped before they are added, then the TS stack would be incorrect.

The execution in Figure 1.5 illustrates why it is incorrect to timestamp elements before adding them to the linked lists. In this execution, push(c) executes concurrently to push(a) and push(b), and push(b) executes sequentially after push(a). A first pop operation executes concurrently to push(b) and removes element a, two other pop operations execute later and remove the elements c and b in sequence. For each push operation we mark the time the element is added to a linked list with a $\times$, and the two $\bullet$ per push operation indicate when an interval timestamp for the element is generated.

In this execution the timestamp of c is generated concurrently to the timestamps of a and b. Therefore c shares a timestamp with a and b. The elements a and b are ordered by their timestamps, but b is added to a linked list only after the first pop operation responds. Therefore b cannot be found by the first pop operation and a appears like a valid candidate element. For the second pop operation, element c shares a timestamp with b and therefore appears to be a valid candidate element.

However, depending on whether push(b) takes effect before or after pop() $\rightarrow a$, either the first pop operation removes a incorrectly, or the second pop operation removes c incorrectly. Since push(a) is executed sequentially before push(b), push(a) definitely takes effect before push(b). Therefore the first pop operation can only remove a correctly if push(b) takes effect after the first pop operation. Now assume that push(b) takes effect after the first pop operation. Since push(c) is executed sequentially before the first pop operation and therefore takes effect before that pop operation, by transitivity push(c) also takes effect before push(b). However, if push(b) takes effect after push(c), then b is the only candidate element for the second pop operation. For correctness the second pop operation would therefore have to remove b instead of c.

Note that since the definition of order-correctness of queue is solely based on precedence information, the TS queue is order-correct no matter if elements are timestamped before or after they are added to the linked lists.

Order Deviation. Additionally to the linearizability proofs and the performance analysis we define a new metric called order deviation to analyze concurrent queue implementations. This analysis focuses on the fact that linearizability does not require that operations take effect in the order they are invoked. Therefore operations may get reordered between their invocation and the time they take effect. With order deviation we analyze how much operations of concurrent queue implementations get reordered between their invocation and the time they take effect.

To measure order deviation we record the order in which insert and remove operations are invoked in benchmarks and calculate how close the operations in this order are to sequential queue semantics. For a perfect queue implementation, a queue implementation where all operations take effect immediately at their invocation times, order deviation would always be zero. If there is a deviation from queue semantics, then this deviation shows that the implementation reorders operations between their invocation and the time they take effect.

In our experiments in Section 8 all concurrent queue implementations we considered show some order deviation. Surprisingly, however, in the case of the TS queue order deviation does not correlate with the amount of timestamp sharing. Our results suggest that the timestamp sharing caused by interval timestamps introduces fewer order deviation than slow dequeue operations caused by the lack of timestamp sharing. Even more surprisingly, the Michael Scott queue [MS96], the standard concurrent queue implementation, shows three times as much order deviation as the TS queue.

This suggests that all concurrent data structures reorder operations internally, and that with interval timestamps the potential due to reordering can be utilized more effectively.

Contributions. To summarize, the contributions of this thesis are:

- A new class of data structures based on timestamping, realized as a stack, queue, and deque.
- A new performance optimization, interval timestamping, which exploits the reordering of operations allowed by linearizability.
- An experimental evaluation of the performance of the TS stack, TS queue, and TS deque.
- A detailed application of the queue conditions proposed by Henzinger et al. [HSV13] to show that the TS queue is linearizable with respect to its sequential specification.
- New stack conditions for establishing the linearizability of concurrent stacks, and a mechanization of the core theorem in the Isabelle HOL proving assistant.
- A detailed application of these stack conditions to show that the TS stack is linearizable with respect to its sequential specification.
- The definition of order deviation, a metric which allows to quantify the number of operations which have been reordered between their invocation and the time they took effect in an execution.
- An analysis of concurrent queue implementations with the order deviation metric.

Thesis structure. In Chapter 2 we give an overview over the key ideas behind timestamped data structures. In Chapter 3 we describe the TS stack algorithm in detail. In Chapter 4 we state and prove our stack theorem, while in Chapter 5 we use it to establish that the TS stack is linearizable with respect to the sequential specification of a stack. In Chapter 6 and Chapter 7 we discuss TS queue and deque variants, respectively. In Chapter 8 we analyze concurrent queue algorithms with the order deviation metric. In Chapter 9 we survey related work and conclude. The appendix contains the Isabelle HOL proof scripts of the stack theorem.

Publications. Most of the content of this thesis has been peer-reviewed:

- The TS stack, which is described in Chapter 2 to Chapter 5, has been published at POPL 2015 [DHK15].
- The implementations of the TS stack, the producer-consumer benchmarks, and the Isabelle HOL proof scripts have been evaluated in the process of the POPL 2015 artifact evaluation.
- The idea of order deviation, described in Chapter 8, has been published at RACES 2012 [HKLP12]. In this thesis, however, we use different definitions, different experiments, and additional data structures. In the original paper, order deviation is called element fairness.
- The idea of the CTS queue and RTS queue in Section 6.3 has been presented at the DMTM 2014 workshop.

Additionally the thesis provides the following content:

- The TS queue and the TS deque descriptions, presented in Chapter 6 and Chapter 7, are based on a technical report [DHK14]. However, the descriptions in this thesis are new.
- In Section 4.3 and Section 5.4 we provide additional proof details which are not contained in TS stack paper [DHK15].

Chapter 2

Timestamped Data Structures

In this chapter we repeat shortly the basic construction of timestamped data structures, and then present the timestamping algorithms we use in our experiments in detail.

2.1 Construction of Timestamped Data Structures

For a timestamped data structure, each thread which is accessing the data structure has an associated single-producer multi-consumer pool (called *SP pools*), implemented as a linked list. These are linked from an array `spPools` which is common for all threads. Each element in the data structure is stored in the SP pool of the thread that inserted it. Figure 2.1 shows the internal structure of the TS stack. The internal structure of the other timestamped data structures is similar.

Inserting an element into a timestamped data structure involves

1. adding a node to the head of the thread's SP pool,
2. generating a new timestamp, and
3. attaching the timestamp to the node.

Thus un-timestamped nodes are visible to other threads – we write these in Figure 2.1 as the maximal value, $\top$. As nodes are only added by one thread to each SP pool, elements in a single pool are totally ordered by their timestamps, and no synchronization is needed when inserting.

Removing from timestamped data structures involves

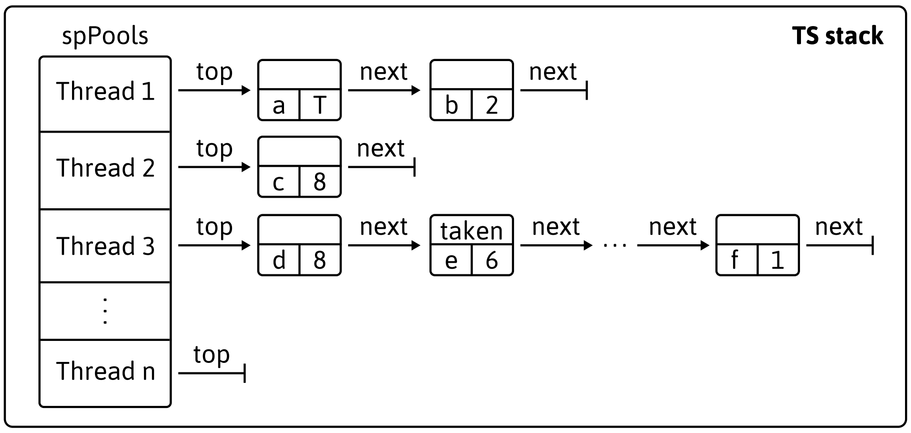


Figure 2.1: The TS stack data structure.

1. searching all the SP pools for an element with a maximal timestamp, and
2. attempting to remove it.

This process repeats until an element is removed successfully. As each SP pool is ordered, searching only requires reading one element of each pool in turn. Timestamps in different pools may be mutually unordered – for example when two timestamps are equal. Thus more than one element may be maximal, and in this case, either can be chosen. Multiple threads may try to remove the same node, so an atomic compare-and-set (CAS) instruction ensures that at most one thread succeeds.

An element can be removed from a timestamped data structure either by removing its node from the linked list of the SP pool, or, as indicated in Figure 2.1, by marking the node of the element as taken [HHL$^+$05]. Marked nodes are removed lazily from the linked list. We show in Chapter 3 how we remove marked nodes from the linked lists of the TS stack.

2.2 Timestamping Algorithms

Next we describe the timestamping algorithms which we use in our experiments. We use three algorithms which generate single-value timestamp, TS-atomic, TS-hardware, and TS-stutter, and we use two algorithms which generate interval timestamps, TS-interval and TS-CAS. All these algorithms guarantee that if two timestamps are generated in sequence, then the timestamps are ordered according to the sequence in which they were generated. If timestamps are generated concurrently, then they may

Listing 2.1: TS-atomic algorithm.

```
TS_Atomic{
 int counter=1;

 Timestamp newTimestamp(){
   return FAI(counter);
 }
}
```

Listing 2.2: TS-hardware algorithm.

```
TS_Hardware{

 Timestamp newTimestamp(){
   return RDTSCP();
 }
}
```

be unordered. Two single-value timestamps are unordered if they are the same. Two interval timestamps are unordered if they overlap. In the following we describe the timestamping algorithms in more detail.

TS-atomic: TS-atomic, shown in Listing 2.1, takes a timestamp from a global counter using an atomic fetch-and-increment (FAI) instruction. Such instructions are available on most modern processors – for example the `LOCK XADD` instruction on x86.

TS-hardware: Listing 2.2 illustrates the TS-hardware algorithm. TS-hardware uses the x86 `RDTSCP` instruction [Int13] to read the current value of the TSC register. The TSC register counts the number of processor cycles since the last reset.

TSC was not originally intended for timestamping, so an obvious concern is that it might not be synchronized across cores. In this case, relaxed-memory effects could lead to stack-order violations. We believe this is not a problem for modern CPUs. Ruan et al. [RLS13] have tested `RDTSCP` on various x86 systems as part of their transactional memory system. Our understanding of [RLS13] and the Intel x86 architecture guide [Int13] is that `RDTSCP` should provide sufficient synchronization on recent multi-core and multi-socket machines. We have also observed no violations of stack semantics in our experiments across different machines. Aside from `RDTSCP`, we use C11 sequentially consistent atomics throughout, forbidding all other relaxed behaviors.

Listing 2.3: TS-stutter algorithm.

```
14 TS_Stutter{
15
16  int[maxThreads] counters;
17
18  Timestamp newTimestamp(){
19    Timestamp timestamp = max(counters) + 1;
20    counters[threadID] = timestamp;
21    return timestamp;
22  }
23 }
```

However, we have anecdotal reports that `RDTSCP` is not synchronized on older x86 systems. Furthermore, memory order guarantees offered by multi-processors are often under-specified and inaccurately documented – see e.g. Sewell et. al.'s work on a formalized x86 model [SSO+10] (which does not cover `RDTSCP`). Implementors should test `RDTSCP` thoroughly before using it to generate timestamps on substantially different hardware.

We hope timestamped data structures will motivate further research into TSC, `RDTSCP`, and hardware timestamp generation more generally.

TS-stutter: TS-stutter, shown in Listing 2.3, uses thread-local counters which are synchronized by Lamport's algorithm [Lam78]. To generate a new timestamp a thread first reads the values of all thread-local counters, calculates the maximum value and increments it by one (line 19). It then stores this value in its thread-local counter (line 20), and returns it as the new timestamp (line 21). Note that the TS-stutter algorithm does not require strong synchronization [AGH+11], e.g. no atomic instructions are used. Therefore TS-stutter may return the same timestamp multiple times. However, this happens only if these timestamps were generated concurrently.

TS-interval: This algorithm, shown in Listing 2.4, does not return a single-value timestamp, but rather an interval consisting of a pair of timestamps generated by one of the algorithms above. Let $[a,b]$ and $[c,d]$ be two such interval timestamps. They are ordered $[a,b] <_{\mathsf{TS}} [c,d]$ if and only if $b < c$. That is, if the two intervals overlap, the timestamps are unordered. The TS-interval algorithm is correct because for any two interval timestamps $[a,b]$ and $[c,d]$, if these intervals are generated sequentially, then b is generated before c and therefore $b < c$, as discussed above.

Listing 2.4: TS-interval algorithm.

```
24 TS_Interval{
25
26  Timestamp newTimestamp(){
27    int timestamp = TS_Hardware.newTimestamp();
28    pause(); // busy wait.
29    int timestamp2 = TS_Hardware.newTimestamp();
30    return [timestamp, timestamp2];
31  }
32 }
```

In our experiments we use the TS-hardware algorithm (i.e. the x86 `RDTSCP` instruction) to generate the start and end of the interval, because it is faster than TS-atomic and TS-stutter. Adding a delay between the generation of the two timestamps (line 28) increases the size of the interval, allowing more timestamps to overlap and thereby reducing contention during element removal. Additionally we increase the potential of elimination. The effect of adding a delay on overall performance is analyzed in Section 3.2.2. For the TS stack the optimal delay is between 3 µs and 10 µs, which is less time than the time it takes to execute a fetch-and-increment instruction on a contended memory location.

TS-CAS: TS-CAS can be seen as an optimization of TS-atomic combined with interval timestamps. It exploits the insight that the shared counter needs only be incremented by *some* thread, not necessarily the current thread. In a highly concurrent situation, many threads using TS-atomic will increment the counter unnecessarily. The TS-CAS algorithm instead uses CAS failure to detect when the counter has been incremented. CAS failure without retrying is comparatively inexpensive, so this scheme is fast despite using strong synchronization.

The source code for TS-CAS is given in Listing 2.5. The algorithm begins by reading the counter value (line 37). If the CAS in line 42 succeeds, then the timestamp takes the counter's original value as its start and end (line 43). If the CAS fails, then another concurrent call must have incremented the counter, and TS-CAS does not have to. Instead it returns an interval starting at the original counter value and ending at the new value minus one (line 44). This interval will overlap with intervals generated by concurrent calls to `newTimestamp`, but will not overlap with any intervals created later.

Listing 2.5: TS-CAS algorithm. The gray highlighted code is an optimization to avoid unnecessary `CAS`.

```
33 TS_CAS{
34  int counter = 1;
35
36  Timestamp newTimestamp(){
37    int timestamp = counter;
38    pause(); // delay optimization.
39    int timestamp2 = counter;
40    if(timestamp != timestamp2)
41      return [timestamp, timestamp2 - 1];
42    if(CAS(counter, timestamp, timestamp + 1))
43      return [timestamp, timestamp];
44    return [timestamp, counter - 1];
45  }
46 }
```

Similar to TS-interval, adding a small delay between reading the counter value and attempting the CAS can improve performance. Here this not only increases the number of overlapping intervals, but also reduces contention on the global counter. Contention is reduced further by line 39-41, a standard CAS optimization. If the value of `counter` changed during the delay, then the CAS in line 42 is guaranteed to fail. Instead of executing the CAS we can immediately return an interval timestamp. Our experiments show that in high-contention scenarios the performance of TS-CAS with a delay is up to 3x faster than without a delay.

Chapter 3

TS Stack

In this chapter we describe the TS stack algorithm in detail. First we discuss its implementation, then we do a detailed performance analysis where we compare the performance of the TS stack, configured with different timestamping algorithms, with other state-of-the-art concurrent stack implementations. The correctness of the TS stack is shown in Chapter 5.

3.1 The TS Stack in Detail

Listing 3.1 and Listing 3.2 show the TS stack code. This code manages the collection of thread-specific SP stack pools linked from the array `spPools`. We factor the SP stack pool out as a separate data structure supporting the following operations:

- `insert` – inserts an element without attaching a timestamp, and return a reference to the new node.
- `getYoungest` – returns a reference to the node in the pool with the youngest timestamp, together with the `top` pointer of the pool.
- `remove` – tries to remove the given node from the pool. Return `true` and the element of the node or `false` and `NULL` depending whether it succeeds.

We describe our implementation of the SP stack pool in Section 3.1.1. Listing 3.1 also assumes the timestamping function `newTimestamp` – various implementations are discussed in Section 2.2.

Listing 3.1: Part 1 of the TS stack algorithm. Part 2 is presented in Listing 3.2. The SP pool is defined in Listing 3.3 and described in Section 3.1.1, timestamps are discussed in Section 2.2.

```
1  TSStack{
2   Node{
3     Element element,
4     Timestamp timestamp,
5     Node next,
6     Bool taken
7   };
8
9   SPPool[maxThreads] spPools;
10
11  void push(Element element){
12    SPStackPool pool=spPools[threadID];
13    Node node=pool.insert(element);
14    Timestamp timestamp=newTimestamp();
15    node.timestamp=timestamp;
16  }
```

We can now describe Listing 3.1 and Listing 3.2. To push an element, the TS stack inserts an un-timestamped element into the current thread's SP pool (line 13), generates a fresh timestamp (line 14), and sets the new element's timestamp (line 15).

A pop iteratively scans over all SP stack pools (line 34-53) and searches for the node with the youngest timestamp in all SP stack pools (line 47-52). The binary operator $<_{\mathsf{TS}}$ is timestamp comparison. This is just integer comparison for single-value timestamps – see Section 2.2. For interval timestamps the comparison operator is described in Section 2.2. If removing the identified node succeeds (line 62) then its element is returned (line 25). Otherwise the iteration restarts.

For simplicity, in Listing 3.1 and Listing 3.2 threads are associated statically with slots in the array `spPools`. To support a dynamic number of threads, this array can be replaced by a linked list for iteration in pop, and by a hashtable or thread-local storage for fast access in push.

Elimination and emptiness checking. Code in gray in Listing 3.1 handles elimination and emptiness checking.

Elimination [HSY04] is an essential optimization in making our stack efficient. It is permitted whenever a push and pop execute concurrently. To detect opportunities for elimination, a pop reads the current time when it starts (line 19). When searching

Listing 3.2: Part 2 of the TS stack algorithm. The SP stack pool is defined in Listing 3.3 and described in Section 3.1.1, timestamps are discussed in Section 2.2. The gray highlighted code deals with the emptiness check and elimination.

```
Element pop(){
  // Elimination
  Timestamp startTime=newTimestamp();
  Bool success;
  Element element;
  do{
    <success,element>=tryRem(startTime);
  } while (!success);
  return element;
}

<Bool,Element> tryRem(Timestamp startTime){
  Node youngest=NULL;
  Timestamp timestamp=-1;
  SPStackPool pool;
  Node top;
  Node[maxThreads] empty;
  for each(SPStackPool current in spPools){
    Node node;
    Node poolTop;
    <node,poolTop>=current.getYoungest();
    // Emptiness check
    if(node==NULL){
      empty[current.ID]=poolTop;
      continue;
    }
    Timestamp nodeTimestamp=node.timestamp;
    // Elimination
    if(startTime <TS nodeTimestamp)
      return current.remove(poolTop,node);
    if(timestamp <TS nodeTimestamp){
      youngest=node;
      timestamp=nodeTimestamp;
      pool=current;
      top=poolTop;
    }
  }
  // Emptiness check
  if(youngest==NULL){
    for each(SPStackPool current in spPools){
      if(current.top!=empty[current.ID])
        return <false,NULL>;
    }
    return <true,EMPTY>;
  }
  return pool.remove(top,youngest);
}
}
```

through the SP stack pools, any element with a later timestamp must have been pushed during the current pop, and can be eliminated immediately (lines 45-46).

To check whether the stack is empty, we reuse an approach from [HHK$^+$13]. When scanning the SP stack pools, if a pool indicates that it is empty, then its `top` pointer is recorded (lines 39-42). If no candidate for removal is found then the SP stack pools are scanned again to check whether their `top` pointers have changed (lines 55-59). If not, the pools must have been empty between the first and second scan. The linearizability of this emptiness check has been proved in [HHK$^+$13].

3.1.1 SP Stack Pool

In this section we describe the SP stack pool, the data structure the TS stack is based on. The SP stack pool (Listing 3.3) is a singly linked list of nodes accessed by a `top` pointer. A node consists of a `next` pointer for the linked list, the `element` it stores, the `timestamp` assigned to the element, and a `taken` flag which indicates if the node has already been removed logically from the SP stack pool. The singly linked list is closed at its end by a sentinel node pointing to itself (line 71). Initially the list contains only the sentinel node. The `taken` flag of the sentinel is set to `true` indicating that the sentinel does not contain an element. The `top` pointer is annotated with an ABA-counter to avoid the ABA-problem [HS08].

Elements are inserted into the SP stack pool by adding a new node (line 76) at the head of the linked list (line 77-78). To remove an element, the `taken` flag of its node is set atomically with a `CAS` instruction (line 99). `getYoungest` iterates over the list (line 87-95) and returns the first node which is not marked as taken (line 91). If no such node is found, `getYoungest` returns `<NULL,oldTop>` (line 93).

Unlinking taken nodes. Nodes marked as `taken` are considered removed from the SP stack pool and are therefore ignored by `getYoungest` and `remove`. However, to reclaim memory and to reduce the overhead of iterating over `taken` nodes, nodes marked as `taken` are eventually unlinked either in `insert` (line 79-82) or in `remove` (line 100-108).

To unlink a node we redirect the `next` pointer from a node a to a node b previously connected by a sequence of `taken` nodes (line 82, line 103, and line 108).

In `insert` the nodes between the new node and the next un-`taken` node are unlinked, and in `remove` the nodes between the old top node and the removed node, and between the removed node and the next un-`taken` node are unlinked. Additionally

Listing 3.3: SP stack pool algorithm. The gray highlighted code deals with the unlinking of taken nodes.

```
SPStackPool{
 Node top;
 Int ID; // The ID of the owner thread.

 init(){
   Node sentinel=createNode(element=NULL,taken=true);
   sentinel.next=sentinel;
   top=sentinel;
 }

 Node insert(Element element){
   Node newNode=createNode(element=element,taken=false);
   newNode.next=top;
   top=newNode;
   Node next=newNode.next; // Unlinking
   while(next->next!=next && next.taken)
     next=next->next;
   newNode.next=next;
   return newNode;
 }

 <Node,Node> getYoungest(){
   Node oldTop=top;
   Node result=oldTop;
   while(true){
     if(!result.taken)
       return <result,oldTop>;
     else if (result.next==result)
       return <NULL,oldTop>;
     result=result.next;
   }
 }

 <Bool,Element> remove(Node oldTop, Node node){
   if(CAS(node.taken,false,true)){
     CAS(top,oldTop,node); // Unlinking
     // Unlink nodes before node in the list.
     if(oldTop!=node)
       oldTop.next=node;
     // Unlink nodes after node in the list.
     Node next=node.next;
     while(next->next!=next && next.taken)
       next=next->next;
     node.next=next;
     return <true,node.element>;
   }
   return <false,NULL>;
 }
}
```

`remove` tries to unlink all nodes between `top` and the removed node (line 100). By using `CAS`, we guarantee that no new node has been inserted between the `top` and the removed node.

The maximum number of nodes in a SP stack pool is bound asymptotically by the number of elements stored in the SP stack pool and the maximum number of threads which can remove nodes concurrently. The key insight is that the maximum number of consecutive `taken` nodes is bounded.

First we observe that the number of consecutive `taken` nodes can only increase if either the node before or the node after the sequence of `taken` nodes is marked as `taken`. However, the sequence of `taken` nodes only increases if after marking an additional node as `taken` no nodes are unlinked.

First assume the node before the sequence is marked as `taken`. In that case in the lines 105-108 the whole sequence of `taken` nodes will be unlinked, and therefore the sequence does not grow.

Next assume that the node after the sequence is marked as `taken`. In this case all nodes visited in the iteration to find the node will be unlinked, except if the node was the top node when it was returned by `getYoungest`. However, this can only happen if the sequence of `taken` nodes was created after the node was returned by `getYoungest` and before it is marked as `taken`. This can only happen N times, where N is the maximum number of concurrent pop operations. Therefore a sequence of marked nodes cannot grow by more than N.

The `next` point of `oldTop` in line 103 can be written racefully, which means that two threads may write to the `next` pointer without synchronization. Thereby it is possible that nodes which have already been unlinked get reinserted again. However, whenever a node is reinserted by a raceful write to the `next` pointer of `oldTop` it is unlinked again later in the lines 105-108.

3.2 Performance Analysis

In our experiments we compare the performance and scalability of the TS stack with two high-performance concurrent stacks: the Treiber stack [Tre86] because it is the de-facto standard lock-free stack implementation and because it is used often as a performance baseline (e.g. in [HSY04, BNHS11]); and the elimination-backoff (EB) stack [HSY04] because it is the fastest concurrent stack we are aware of.

Of course, other high-performance stacks exist. We decided against benchmarking the DECS stack [BNHS11] because (1) no implementation is available for our platform and (2) according to their experiments, in peak performance it is no better than a flat-combining stack [HIST10]. We decided against benchmarking the flat-combining stack because the EB stack outperforms it when configured to access the elimination array before the stack itself, see the description of the EB stack below.

The Treiber stack consists of a linked list of nodes which is accessed by a `top` pointer. Push and pop operations try to modify the `top` pointer with a `CAS` instruction until they succeed.

The EB stack consists of a Treiber stack backend and an elimination array. The elimination array works as follows: Push and pop operations start by writing an operation descriptor into a random slot of the elimination array. If a push and a pop operation write their descriptor into the same slot and thereby collide, the pop operation can return the value of the push operation without accessing the backend stack because of elimination. If no collision happens, then an operation removes its descriptor from the elimination array and tries to add the element to the backend stack. To increase the chance of a collision an operation waits some time before it removes its descriptor from the elimination array. This waiting is similar to the delay in the TS-interval timestamping.

For better performance in high contention scenarios, in the EB stack the operations access the elimination array before they access the backend stack. However, this means that in scenarios where no elimination is possible (e.g. producer-only workloads) the EB stack suffers from a constant-time overhead introduced by the elimination array.

Benchmarking Environment. We ran our experiments on two x86 machines:

- an Intel-based server with four 10-core 2GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of UMA memory running Linux 3.8.0-36; and
- an AMD-based server with four 16-core 2.3GHz AMD Opteron processors (64 cores), 16MB shared L3-cache, and 512GB of cc-NUMA memory running Linux 3.5.0-49.

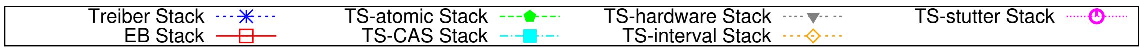
Measurements were done in the Scal Benchmarking Framework [HHK$^+$15]. To avoid measurement artifacts, the framework uses a custom memory allocator which performs cyclic allocation [NR07] in preallocated thread-local buffers for objects smaller than 4096 bytes. Larger objects are allocated with the standard allocator of glibc. All memory is allocated cache-aligned when it is beneficial to avoid cache artifacts. The framework is written in C/C++.

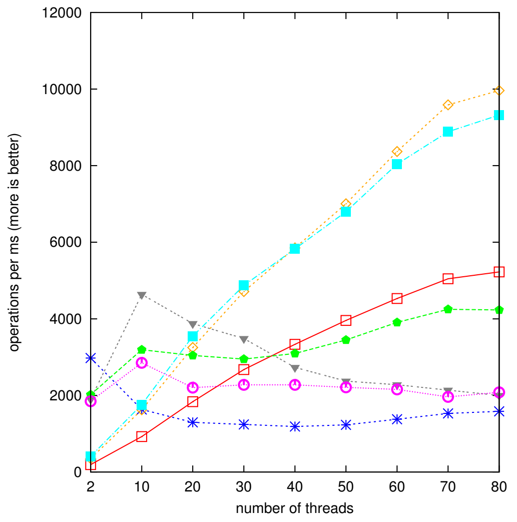
Scal provides implementations of the Treiber stack and of the EB stack. Unlike the description of the EB stack in [HSY04] we access the elimination array *before* the backend stack – this improves scalability in our experiments. We configured the EB stack such that the performance is optimal in our benchmarks when exercised with 80 threads on the 40-core machine, or with 64 threads on the 64-core machine. These configurations may be suboptimal for lower numbers of threads. Similarly, the TS stack configurations we discuss later are selected to be optimal for 80 and 64 threads on the 40-core and 64-core machine, respectively. On the 40-core machine the elimination array is of size 28 with a delay of 28 µs in the high-contention benchmark, and of size 24 with a delay of 28 µs in the low contention benchmark. On the 64-core machine the elimination array is of size 32 with a delay of 21 µs in the high-contention benchmark, and of size 16 with a delay of 18 µs in the low contention benchmark.

On the 64-core machine the Treiber stack benefits from a backoff strategy [DHM13] which delays the retry of a failed `CAS`. On this machine, we configured the Treiber stack with a constant delay of 200 µs in the high-contention experiments and a constant delay of 150 µs in the low-contention experiments, which is optimal for the benchmark when exercised with 64 threads. On the 40-core machine performance decreases when a backoff delay is added, so we disable it.

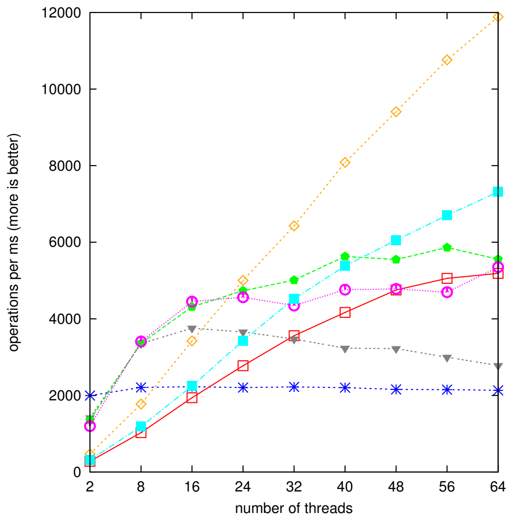
We compare the data structures in producer-consumer microbenchmarks where threads are split between dedicated producers which insert 1,000,000 elements into the data structure, and dedicated consumers which remove 1,000,000 elements from the data structure. We measure performance as total execution time of the benchmark. Figures show the total execution time in successful operations per millisecond to make scalability more visible. All numbers are averaged over 5 executions. To avoid measuring empty removal, operations that do not return an element are not counted.

The contention on the data structure is controlled by a busy wait between two operations of a thread. In the high-contention scenario the busy wait is 1 µs, in the low-contention the busy wait is 10 µs long. We do not show any results without the busy wait because in these measurements machine artifacts of the memory system dominate the results.

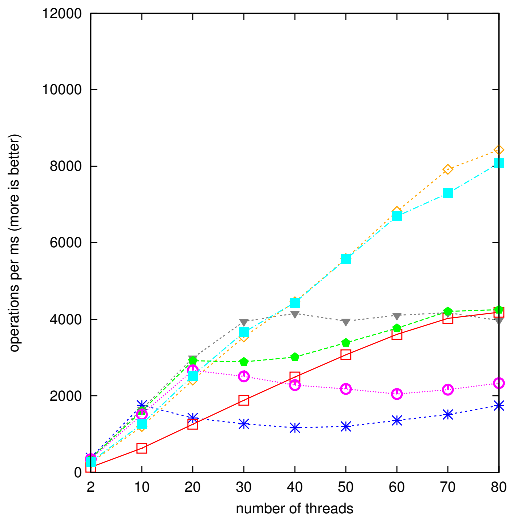




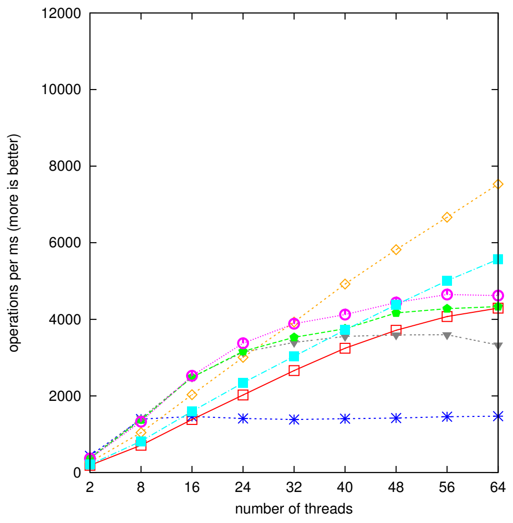
(a) High contention benchmark on the 40-core machine.



(b) High contention benchmark on the 64-core machine.



(c) Low contention benchmark on the 40-core machine.



(d) Low contention benchmark on the 64-core machine.

Figure 3.1: TS stack performance on the 40-core machine (left) and on the 64-core machine (right) in the producer-consumer benchmark.

3.2.1 Performance and Scalability Results

Figures 3.1 shows performance and scalability in the producer-consumer benchmark where half of the threads are producers and half of the threads are consumers. In

	40-core machine	64-core machine
high-contention:		
TS-interval stack	9 µs	6 µs
TS-CAS stack	7.5 µs	10.5 µs
low-contention:		
TS-interval stack	4.5 µs	4.5 µs
TS-CAS stack	3 µs	9 µs

Table 3.1: Benchmark delay times for the TS-interval stack / TS-CAS stack.

the discussion we focus on the high-contention measurements. Results in the low-contention benchmarks are similar, but less pronounced.

For TS-interval timestamping and TS-CAS timestamping we use the optimal delay when exercised with 80 threads on the 40-core machine, and with 64 threads on the 64-core machine, derived from the experiments in Section 3.2.2. The delay thus depends on the machine and benchmark. The delay times we use in the benchmarks are listed in Table 3.1. The impact of different delay times on performance is discussed in Section 3.2.2.

TS stack in the producer-consumer benchmark

We start our analysis with a comparison of the performance of the TS stack with different timestamping algorithms. As a reminder, the following timestamping algorithms were described in Section 2.2:

TS-atomic A timestamp is generated from a single counter which is incremented with an atomic fetch-and-increment instruction.

TS-CAS A variant of interval timestamping. TS-CAS generates an interval timestamp from a single counter which is incremented with an atomic compare-and-set instruction.

TS-hardware A timestamp is generated from the TSC register of the CPU using the `RDTSCP` instruction.

TS-interval A variant of interval timestamping. A timestamp interval is generated by reading the TSC register twice.

TS-stutter The generated timestamp is the maximum value of a vector of thread-local counters.

	40-core machine	64-core machine
EB stack	92%	76%
TS-atomic stack	89%	93%
TS-hardware stack	43%	64%
TS-stutter stack	96%	88%
TS-interval stack	99%	98%
TS-CAS stack	99%	99%

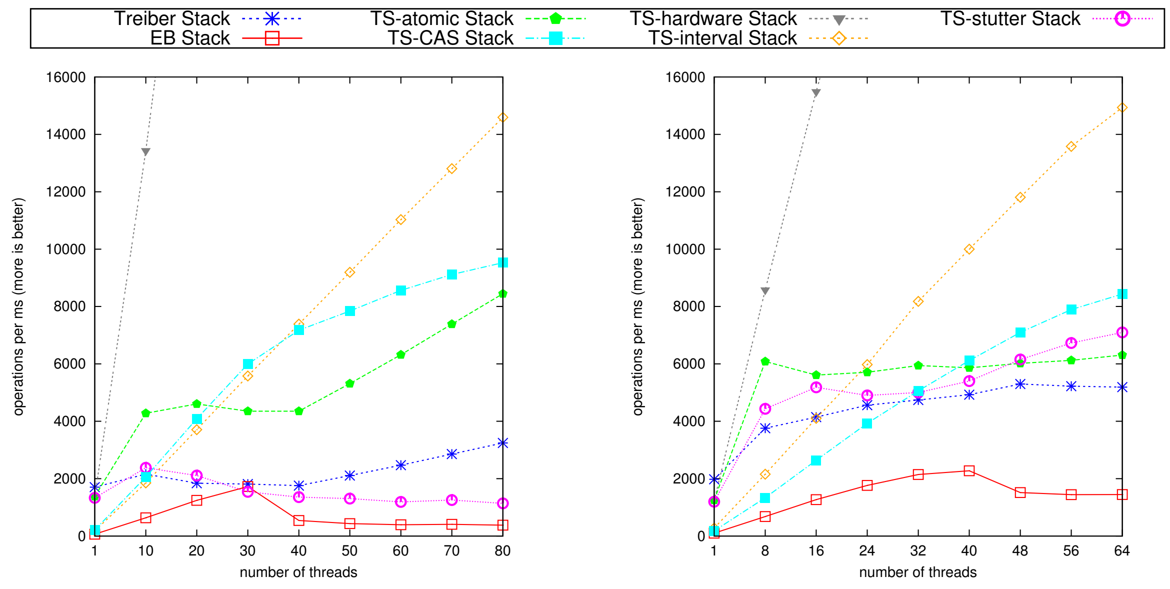
Table 3.2: Percentage of elements removed by elimination in the high-contention producer-consumer benchmark with 80 threads on the 40-core machine and with 64-threads on the 64-core machine.

With an increasing number of threads the TS-interval stack is faster than the TS stack variants in the producer-consumer benchmarks. Interestingly the TS-atomic stack is faster than the TS-hardware stack in the high-contention producer-consumer benchmark. The reason is that since the push operations of the TS-hardware stack are so much faster than the push operations of the TS-atomic stack, elimination is possible for more pop operations of the TS-atomic stack (e.g. 41% more elimination on the 64-core machine, see Table 3.2), which results in a factor of 3 less retries of `tryRem` operations than in the TS-hardware stack. On the 40-core machine the TS-stutter stack is significantly slower than the TS-atomic stack, while on the 64-core machine their performance is similar. The reason is that on the 40-core machine TS-stutter timestamping is significantly slower than TS-atomic timestamping (see Figure 3.2). On the 40-core machine the TS-CAS stack is much faster than the TS-hardware stack, TS-atomic stack, and TS-stutter stack, on the 64-core machine it is slightly faster. The reason seems to be that on the 64-core machine a `CAS` is slower in comparison to other instructions than on the 40-core machine.

EB Stack and Treiber Stack in the producer-consumer benchmark

With more than 16 threads all TS stacks are faster than the Treiber stack in the producer-consumer benchmark. On both machines the TS-interval stack and the TS-CAS stack outperform the EB stack in the high-contention producer-consumer benchmark with a maximum number of threads, on the 64-core machine also the TS-stutter stack and the TS-atomic stack are as fast as the EB stack.

We believe TS-interval's and TS-CAS's performance increase with respect to the EB stack comes from three sources: (a) more elimination; (b) faster elimination; (c)



(a) 40-core machine (b) 64-core machine

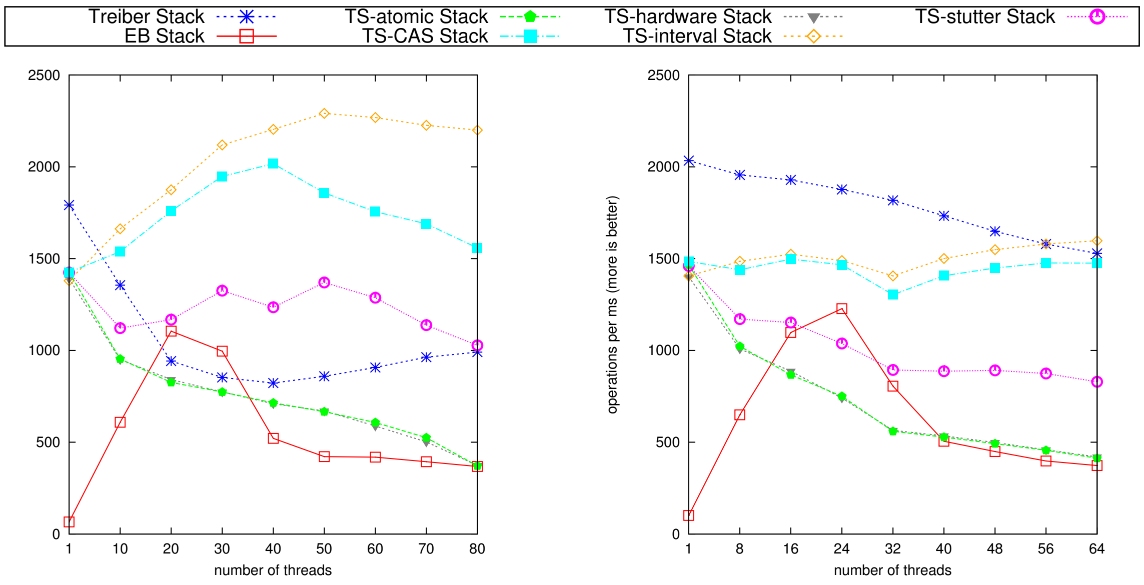
Figure 3.2: TS stack performance on the 40-core machine (left) and on the 64-core machine (right) in the high contention producer-only benchmark.

higher performance without elimination. As shown in producer-only and consumer-only experiments, the lack of push-contention and mitigation of contention in pop makes our stack fast even without elimination. Additional experiments show that for example the TS-interval stack eliminates 7% and 23% more elements than the EB stack in high-contention scenarios on the 40-core and on the 64-core machine, respectively. Thus we improve on EB in both (a) and (c). (b) is difficult to measure, but we suspect integrating elimination into the normal code path introduces less overhead than an elimination array, and is thus faster.

Push performance

We measure the performance of push operations of all data structures in a producer-only benchmark where each thread pushes 1,000,000 element into the stack. The TS-interval stack and the TS-CAS stack use the same delay as in the high-contention producer-consumer benchmark, see Table 3.1:

Figure 3.2 shows the performance and scalability of the data structures in the high-contention producer-only benchmark. The push performance of the TS-hardware stack is significantly better than the push of the other stack implementations. On the 40-core machine it scales linearly up to 80 threads and achieves up to 94,000 operations



(a) 40-core machine (b) 64-core machine

Figure 3.3: TS stack performance on the 40-core machine (left) and on the 64-core machine (right) in the high contention consumer-only benchmark.

per millisecond. On the 64-core machine the TS-hardware stack performs 52,000 operations per millisecond with 64 threads.

With an increasing number of threads the push operation of the TS-interval stack is faster than the push operations of the TS-atomic stack and the TS-stutter stack, which means that the delay in the TS-interval timestamping is actually shorter than the execution time of the TS-atomic timestamping and the TS-stutter timestamping. Perhaps surprisingly, TS-stutter, which does not require strong synchronisation, is slower than TS-atomic on the 40-core machine, although TS-atomic is based on an atomic fetch-and-increment instruction.

Pop performance

We measure the performance of pop operations of all data structures in a consumer-only benchmark where each thread pops 1,000,000 from a pre-filled stack. Note that no elimination is possible in this benchmark. The stack is pre-filled concurrently, which means in case of the TS-interval stack, TS-CAS stack, and TS-stutter stack that some elements may have unordered timestamps. Again the TS-interval stack and the TS-CAS stack use the same delay as in the high-contention producer-consumer benchmark, see Table 3.1.

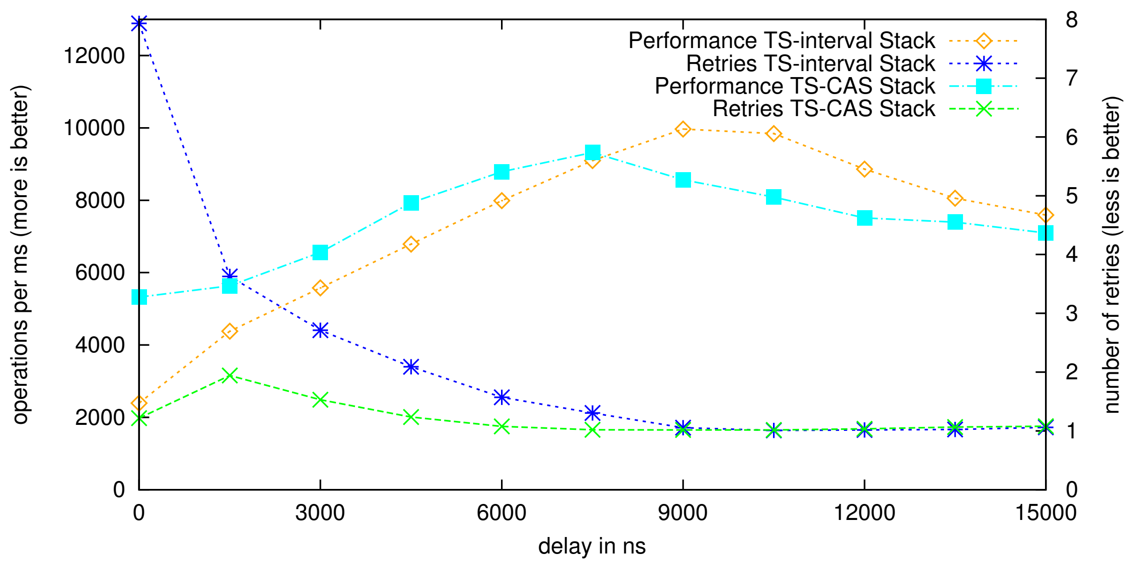


Figure 3.4: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 40-core machine, exercising 40 producers and 40 consumers.

Figure 3.3 shows the performance and scalability of the data structures in the high-contention consumer-only benchmark. The performance of the TS-interval stack is higher than the performance of the other TS stack variants. The performance of TS-CAS is close to the performance of TS-interval on the 64-core machine. The TS-stutter stack is faster than the TS-atomic and TS-hardware stack due to the fact that some elements share timestamps and therefore can be removed in parallel. The TS-atomic stack and TS-hardware stack show the same performance because all elements have unique timestamps and therefore have to be removed sequentially. Also in the Treiber stack and the EB stack elements have to be removed sequentially. Removing elements sequentially from multiple lists (TS stack) is more expensive than removing elements sequentially from a single list (Treiber stack).

3.2.2 Analysis of Interval Timestamping

Figure 3.4 and Figure 3.5 show the performance of the TS-interval stack and the TS-CAS stack along with the average number of `tryRem` calls needed in each pop (one call is optimal, but contention may cause retries). These figures were collected with an increasing interval length in the high contention producer-consumer benchmark with 40 producers and 40 consumers on the 40-core machine, and with 32 producers and

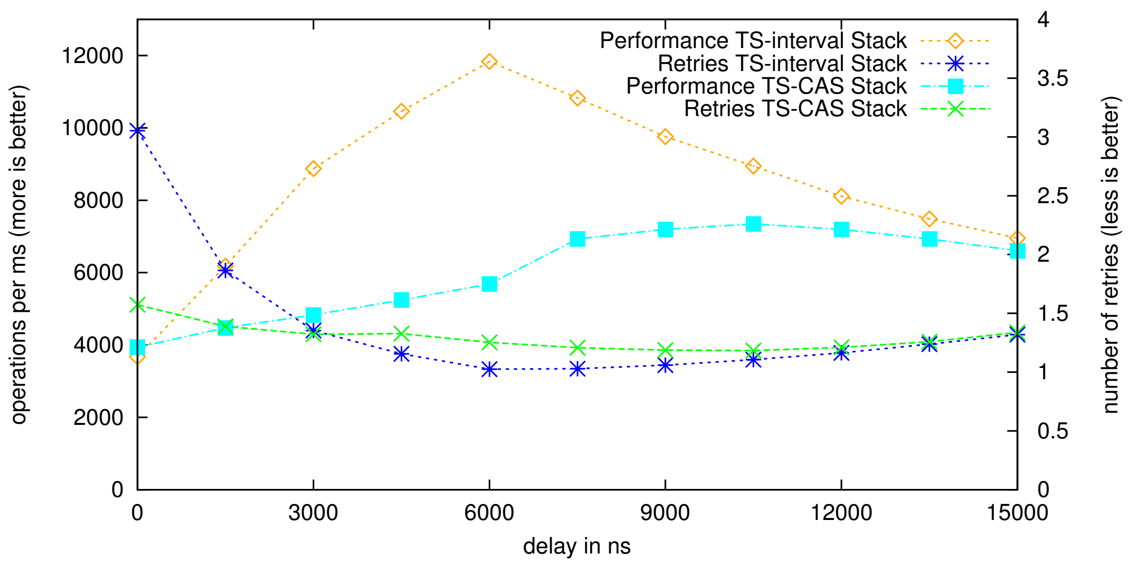


Figure 3.5: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 64-core machine, exercising 32 producers and 32 consumers.

32 consumers on the 64-core machine. We used these results to determine the delays for the benchmarks in Section 3.2.1.

Initially the performance of the TS-interval stack increases with an increasing delay time, but beyond 9 µs on the 40-core machine the performance decreases again. After that point an average push operation is slower than an average pop operation and the number of pop operations which return `EMPTY` increases.

For the TS-interval stack the high performance correlates with a drop in `tryRem` retries. We conclude from this that the impressive performance we achieve with interval timestamping arises from reduced contention in `tryRem`. For the optimal delay time we have 1.009 calls to `tryRem` per pop, i.e. less than 1% of pop calls need to scan the SP stack pools array more than once. In contrast, without a delay the average number of retries per pop call is more than 6.

The performance of the TS-CAS stack increases initially with an increasing delay time. However this does not decrease the number of `tryRem` retries significantly. The reason is that without a delay there is more contention on the global counter of the timestamping algorithm. Therefore the performance of the TS-CAS timestamping algorithm with a delay is actually better than the performance without a delay.

Beyond a delay of 10 µs the performance of teh TS-CAS stack decreases again. This is the point where an average push operation becomes slower than an average pop operations.

Chapter 4

Correctness Theorem for Stacks

In this chapter we prove a correctness theorem for concurrent stacks in general. We start with definitions and a general discussion about correctness proofs of concurrent data structures. We then show with examples why the common approach of correctness proofs is challenging for the TS stack. In Section 4.2 we state and prove a stack theorem which helps to prove the correctness of the TS stack. Additional proof details are provided in Section 4.3.

4.1 Problem Description

We are interested in correctness in terms of linearizability, [HW90], the standard correctness condition for concurrent algorithms. Linearizability ensures that every behavior observed by an algorithm's calling context could also have been produced by a sequential (i.e. atomic) version of the same algorithm. We call the ideal sequential version of the algorithm the *specification*, e.g. in Section 4.1.3 we define a sequential specification of a stack.

Interactions between the algorithm and its calling context in a given execution are expressed as a *history*.

Definition 1 (History). *A* history $\mathcal{H}$ *is a tuple* $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ *where* $\mathcal{A}$ *is a finite set of operations (for example, push(5)), and* $\mathsf{pr}, \mathsf{val} \subseteq \mathcal{A} \times \mathcal{A}$ *are the partial* precedence *order and* value *relation, respectively. A history is* sequential *if* pr *is total.*

Given a particular program execution, a history can be extracted from its *trace*, $\mathcal{T}$. The trace of an execution is the interleaved sequence of events that took place during the execution of a program and therefore of the algorithms of the program. Events are the invocations and responses of operations as well as instructions within operations. In this thesis we only consider histories which were extracted from traces.

To extract a history, we first generate the set $\mathcal{A}$ of executed operations in a trace. As is standard in linearizability, assume that all invocations have corresponding responses. A pair (x,y) of operations is in pr if the response event of x is ordered before the invocation event of y in $\mathcal{T}$. A pair (x,y) is in val if x is an insert operation, y is a remove operation, and the value inserted by x was removed by y. Note that by using val we do not have to assume that values are unique.

Linearizability requires that algorithms only interact with their calling context through invocation and response events. Therefore, a history captures all interactions between algorithm and context. We thus define a data structure specification as just a set of histories. For example, Stack is the set of histories produced by an ideal sequential stack). Linearizability is defined by relating implementation and specification histories.

Definition 2 (linearizability)**.** *A history* $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ *is* linearizable *with respect to some specification* $\mathcal{S}$ *if there exists a* linearization order pr^T *such that* $\mathsf{pr} \subseteq \mathsf{pr}^T$*, and* $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val} \rangle \in \mathcal{S}$*.*

An implementation C is linearizable with respect to $\mathcal{S}$ if any history $\mathcal{H}$ arising from the algorithm is linearizable with respect to $\mathcal{S}$.

Our formulation of linearizability differs from the classic one [HW90]. Rather than have a history record the total order on invocations and responses, we convert this information into the partial order pr. Likewise, linearizability between histories is defined by inclusion on partial orders, rather than by reordering invocation and response events. This approach, taken from [BDG13], is convenient for us because our stack theorem is defined by constraints on partial orders. However, the two formulations are equivalent.

4.1.1 The Problem with Linearization Points

Proving that a concurrent algorithm is linearizable with respect to a sequential specification amounts to showing that, for every possible execution, there exists a total linearization order. The standard strategy is to identify *linearization points* in the algorithm's syntax. Conceptually, when a linearization point is reached, the method 'takes effect' and is appended to the linearization order, pr^T. Thus the implementa-

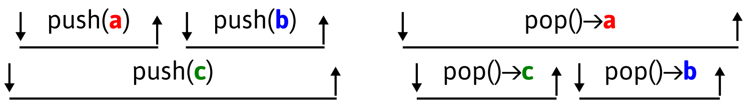


Figure 4.1: A concurrent execution.

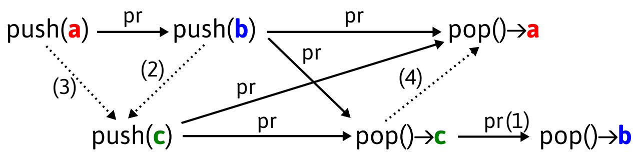


Figure 4.2: A graph showing the precedence order pr of the execution in Figure 4.1.

tion and specification histories are constructed in lock-step, allowing the algorithm designer to show that they correspond.

It has long been understood that linearization points are a limited approach. Algorithms may have linearization points inside other methods or fixed non-deterministically by future behavior. The TS stack is a particularly acute example of this problem. Two push methods that run concurrently may insert elements with unordered timestamps, giving no information to choose a linearization order. However, if the items are later popped sequentially, an order is imposed on the earlier pushes. Worse, ordering two pushes can implicitly order other methods, leading to a cascade of linearizations back in time.

We illustrate the problem in Figure 4.1. The precedence order pr of the history in Figure 4.1 is shown in Figure 4.2. First consider the history immediately before the return of pop()→c (i.e. without order (1) in the graph in Figure 4.2). As push(b) and push(c) run concurrently, elements b and c may have unordered timestamps. At this point, there are several consistent ways that the history might linearize, even given access to the TS stack's internal state.

Now consider the history after pop()→b. Dotted edges represent linearization orders forced by this operation. As c is popped before b, LIFO order requires that push(b) has to be linearized before push(c) – order (2). Transitivity then implies that push(a) has to be ordered before push(c) – order (3). Furthermore, ordering push(a) before push(c) requires that pop()→c is ordered before pop()→a – order (4). Thus a method's linearization order may be fixed long after it returns, frustrating any attempt to choose syntactic linearization points.

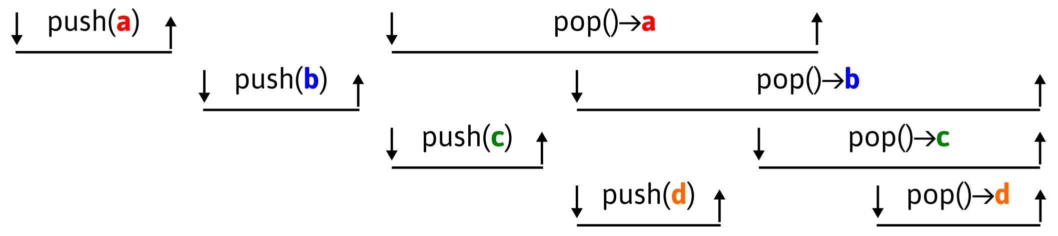


Figure 4.3: Example of an execution which is not linearizable with respect to stack specification because of non-local behavior.

4.1.2 Specification-Specific Conditions

For a given sequential specification, it may not be necessary to find the entire linearization order to show that an algorithm is linearizable. A degenerate example is the specification which contains all possible sequential histories; in this case, we need not find a linearization order, because any order consistent with pr will do. One alternative to linearization points is thus to invent special-purpose conditions for particular sequential specifications.

Henzinger et al. [HSV13] have justified such a set of conditions for queues. (They call this approach *aspect-oriented*). Informally, these condidtions require that a data structure is linearizable without order constraints, and that whenever two enqueue operations enqueue(a) and enqueue(b) are ordered enqueue(a) $\xrightarrow{\text{pr}}$ enqueue(b), then their matching dequeue operations dequeue(a) and dequeue(b) are not ordered dequeue(b) $\xrightarrow{\text{pr}}$ dequeue(a).

One attractive property of this approach is that the queue-specific conditions are expressed by using only the precedence order, pr. In other words, correct behavior in term of element order can be checked without locating linearization points at all.

Our goal is to prove a theorem for stacks which is based on similar conditions. By 'similar', we mean a local theorem defined by forbidding a finite number of finite-size bad orderings. It is this locality that makes Henzinger's theorem so appealing. It reduces data structure correctness from global ordering to ruling out a number of specific bad cases. However, we believe that any theorem for stacks must require some additional information to pr (which means, as a corollary, that checking linearizability for stacks is fundamentally harder than for queues).

Our key evidence that additional information is needed is the execution shown in Figure 4.3. This execution as a whole is not linearizable – this can be seen more clearly in the graph shown in Figure 4.4, which projects out the pr and val relations. By the definition of linearizability all pr edges have to be preserved in the linearization.

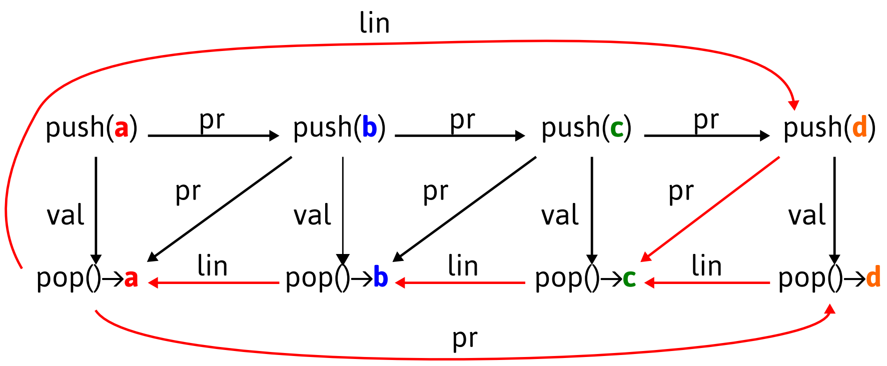


Figure 4.4: The pr and val relations of the example in Figure 4.3.

Additionally, the lin edges in Figure 4.4 are required in the linearization by stack semantics. However, the pr and lin edges form cycles, indicated in red, which contradicts the requirement that the linearization is a total order. Therefore it is not possible to find a linearization of the example in Figure 4.3 which satisfies stack semantics.

However, if for any $i \in \{a,b,c,d\}$ the corresponding push(i) – pop()→i pair is deleted, the execution becomes linearizable with respect to stack semantics. Intuitively, doing this breaks the cycle in $\mathsf{lin} \cup \mathsf{pr}$ that appears in Figure 4.4. Thus, any forbidden shape based on precedence that is smaller than this whole execution cannot forbid it – otherwise it would forbid legitimate executions. Worse, we can make arbitrarily-large bad executions of this form. Thus no theorem based on finite-size forbidden shapes can define linearizability for stacks. Instead, in our stack theorem we define what we call an insert-remove relation which introduces just enough extra structure to let us define a local stack theorem.

Note that it is not possible to construct a similar execution like the one in Figure 4.3 for queues. The reason is that for a queue a lin edge never depends on the order between an enqueue and a dequeue.

4.1.3 Stack and Set Specifications

Our theorem makes use of two sequential specifications: Stack, and a weaker specification Set that does not require LIFO order. We define the set of permitted histories by defining updates over abstract states. Assume a set of values Val. Abstract states are finite sequences in Val^*. Let $\sigma \in \mathsf{Val}^*$ be an arbitrary state. In Stack, push and pop have the following sequential behavior ('$\cdot$' means sequence concatenation, ε means an empty sequence):

- push(v) – Update the abstract state to $\sigma \cdot \mathrm{v}$.
- pop() – If $\sigma = \varepsilon$, return EMPTY. Otherwise, σ must be of the form $\sigma' \cdot v'$, with $\sigma' \in \mathsf{Val}^*$ and $v' \in \mathsf{Val}$. Update the state to σ', return v'.

In Set, push is the same, but pop behaves as follows:

- pop() – If $\sigma = \varepsilon$, return EMPTY. Otherwise, σ must be of the form $\sigma' \cdot v' \cdot \sigma''$, with $\sigma', \sigma'' \in \mathsf{Val}^*$ and $v' \in \mathsf{Val}$. Update the state to $\sigma' \cdot \sigma''$, return v'.

4.2 The Stack Theorem

We have developed stack conditions sufficient to ensure linearizability with respect to Stack. As described in Section 4.1.2, our conditions are not expressed using only pr. Rather we require an auxiliary insert-remove relation ir which relates pushes to pops and vice versa, but that does not relate pairs of pushes or pairs of pops. We call such a relation an alternating relation.

Definition 3 (alternating). *We call a relation r on $\mathcal{A}$* alternating *if every pair $+a, -b \in \mathcal{A}$ consisting of one push and one non-empty pop is related, no push $+a$ is in relation with another push $+c$, and no pop $-b$ is in relation with another pop $-d$.*

Figure 4.5 illustrates an alternating relation ir: each of the push operations $+a$, $+c$, $+c'$, $+c''$ is in relation with each of the pop operations $-b$, $-b'$, $-d$. However, no push operation is in relation with another push operation, and no pop operation is in relation with another pop operation.

We use the alternating relation ir as the core in our construction of a linearization order. In other words, our theorem shows that for stacks it is sufficient to identify just *part* of the linearization order.

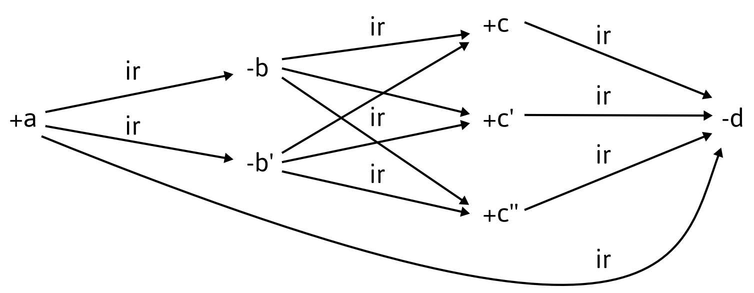


Figure 4.5: Example of an alternating relation ir.

For our formulation of the stack conditions we define the helper orders ins and rem over push operations and pop operations, respectively. Informally, ins and rem are fragments of the linearization order that are imposed by the combination of ir and the precedence order pr. In all the definitions in this section, assume that $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ is a history. Below we write $+a, +b, +c$ etc. for push operations, and $-a, -b, -c$ etc. for pop operations.

Definition 4 (derived orders ins and rem)**.** *Given an alternating insert-remove relation* ir*.*

- *For all $+a, +b \in \mathcal{A}$, $+a \xrightarrow{\mathsf{ins}} +b$ if either $+a \xrightarrow{\mathsf{pr}} +b$ or there exists an operation $-c \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{ir}} +b$.*
- *For all $-a, -b \in \mathcal{A}$, $-a \xrightarrow{\mathsf{rem}} -b$ if either $-a \xrightarrow{\mathsf{pr}} -b$ or there exists an operation $+c \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{ir}} +c \xrightarrow{\mathsf{ir}} -b$.*

Here ins is not symmetrical to rem – note the pr rather than ir in the final clause. However, our stack theorem also holds if the definitions are inverted. The version above is more convenient in verifying the TS stack.

The order ins expresses ordering between push operations imposed either by precedence, or transitively by insert-remove. Likewise rem expresses ordering between pop operations. In the TS stack, ins is approximated by the timestamps, rem only exists implicitly. Using ins and rem, we can define order-correctness, which expresses the conditions necessary to achieve LIFO ordering in a stack.

Definition 5 (order-correct)**.** *We call $\mathcal{H}$* order-correct *if there exists an alternating relation* ir *on $\mathcal{A}$ with derived orders* ins *and* rem*, such that:*

1. *$\mathsf{ir} \cup \mathsf{pr}$ is cycle-free; and*
2. *Let $+a, -a, +b \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ and $+a \xrightarrow{\mathsf{pr}} -a$. If $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{ir}} -a$, then there exists $-b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ and either $+b \not\xrightarrow{\mathsf{pr}} -b$ or $-a \not\xrightarrow{\mathsf{rem}} -b$;*

Condition (2) is at the heart of our proof approach. It forbids the non-LIFO behavior illustrated in Figure 4.6.

As ins, rem and ir are all fragments of the eventual linearization order, this shape corresponds to a non-LIFO ordering that would violate the stack specification.

Order-correctness *only* imposes LIFO ordering; it does not guarantee non-LIFO correctness properties. For a stack, these non-LIFO correctness properties are (1) elements should not be lost; (2) elements should not be duplicated; (3) popped elements

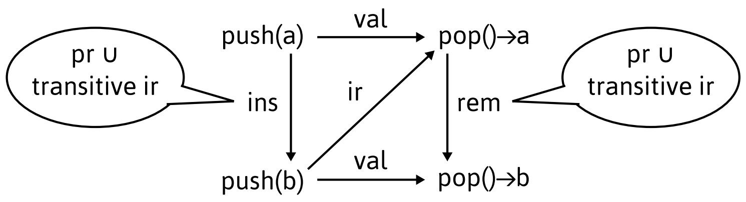


Figure 4.6: Non-LIFO behavior forbidden by the stack theorem.

should come from a corresponding push; and (4) pop should report EMPTY correctly. The last is subtle, as it is a global rather than pairwise property: pop should return EMPTY only at a point in the linearization order where the abstract stack is empty. Fortunately, these properties are also orthogonal to LIFO ordering: we just require that the algorithm is linearizable with respect to SET, which is simple to prove for the TS stack.

Theorem 1 (Stack Theorem)**.** *Let C be a concurrent algorithm. If every history arising from C is order-correct, and C is linearizable with respect to* SET*, then C is linearizable with respect to* STACK*.*

Before we start with the proof of Theorem 1 we define two types of operations which we treat specially in the proof.

Definition 6 (pop-empty)**.** *Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history, and let $-e \in \mathcal{A}$ be a pop operation such that there does not exist an operation $+e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$. Then we call $-e$ a* pop-empty *operation.*

Definition 7 (elimination pair)**.** *Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history, and let $+a, -a \in \mathcal{A}$ be a push-pop pair with $+a \xrightarrow{\mathsf{val}} -a$. If $+a \overset{\mathsf{pr}}{\not\rightarrow} -a$, then we call $(+a, -a)$ an* elimination pair*.*

With these definitions we prove now Theorem 1. In this proof we focus mainly on the key insights and construction steps and avoid technical details. All proof details are available in the mechanization of the proof which we did in Isabelle HOL. The proof scripts are provided in the appendix. The parts of the proof which deal with pop-empty operations and elimination pairs are described additionally in Section 4.3 in Lemma 3 and Lemma 4.

Proof. We prove the theorem by showing that for any history $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ which is linearizable with respect to SET and which is order-correct we can construct a linearization order pr^T such that $\mathsf{pr} \subseteq \mathsf{pr}^T$ and $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val} \rangle \in$ STACK.

$$\begin{tikzcd}
+c \arrow[r, "\mathsf{pr}_4"] & -b \arrow[r, "\mathsf{pr}_4"] & -c \arrow[r, "\mathsf{pr}_4"] & -a \\
& +a \arrow[u, "\boldsymbol{\mathsf{pr}_4}"] \arrow[ur, "\mathsf{pr}_4"'] & &
\end{tikzcd}$$

Figure 4.7: Edges added to pr_4 from a push operation $+a$ to a pop operation $-b$.

In the first construction step we construct the linearization order of a sub-history $\langle \mathcal{A}', \mathsf{pr}', \mathsf{val}' \rangle$ of $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$. This sub-history is constructed from $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ by removing all pop-empty operations and all elimination pairs. We show in Lemma 3 and Lemma 4 that we can deal with pop-empty operations and elimination pairs separately.

The core of the linearization order of the sub-history is its precedence order pr'. In a first step we extend pr' to relation pr_1 with $\mathsf{pr}' \subseteq \mathsf{pr}_1$ as follows, where $a, b \in \mathcal{A}$:

- $a \xrightarrow{\mathsf{pr}_1} b$ if $a \xrightarrow{\mathsf{pr}} b$.
- $a \xrightarrow{\mathsf{pr}_1} b$ if $a \xrightarrow{\mathsf{rem}} b$.
- $a \xrightarrow{\mathsf{pr}_1} b$ if $b \xrightarrow{\mathsf{rem}} a$ would violate order correctness, i.e. there exist $+a, +b \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} a$, $+b \xrightarrow{\mathsf{val}} b$, and $+b \xrightarrow{\mathsf{ins}} +a \xrightarrow{\mathsf{ir}} b$.

In a second step we construct a relation pr_2 with $\mathsf{pr}_1 \subseteq \mathsf{pr}_2$. Additionally to the operations related in pr_1 all pop operations are related in pr_2 such that pr_2 becomes a total order on pop operations.

In a third step we construct pr_3 as the transitive closure of pr_2. We showed in Isabelle HOL that for any order-correct history which does not contain pop-empty operations or elimination pairs the relation pr_3 as constructed above is a partial order on all operations, and a total order on pop operations.

In the next step we construct a partial order pr_4 with $\mathsf{pr}_3 \subseteq \mathsf{pr}_4$ by extending pr_3 with edges between push and pop operations. Informally we order push operations as late as possible. The relation pr_4 is defined recursively. Formally a push operation $+a$ and a pop operation $-b$ are ordered $+a \xrightarrow{\mathsf{pr}_4} -b$ if either

1. $+a \xrightarrow{\mathsf{pr}_3} -b$ or
2. there exist $-a, +c, -c \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$, $+c \xrightarrow{\mathsf{val}} -c$, $+c \xrightarrow{\mathsf{pr}_4} -b$, $-b \xrightarrow{\mathsf{pr}_4} -c \xrightarrow{\mathsf{pr}_4} -a$, and $+a \xrightarrow{\mathsf{pr}_4} -c$. This condition is illustrated in Figure 4.7. Note that if

$-b \xrightarrow{\mathsf{pr}_4} +a$ were constructed, then there would be the stack violation $+c \xrightarrow{\mathsf{lin}} +a \xrightarrow{\mathsf{lin}} -c \xrightarrow{\mathsf{lin}} -a$.

If $+a$ and $-b$ are not ordered, $+a \xrightarrow{\mathsf{pr}_4} -b$, then these operations are ordered $-b \xrightarrow{\mathsf{pr}_4} +a$.

Similar to above we define pr_5 as the transitive closure of pr_4. We showed in Isabelle HOL that pr_5 is a partial order on all operations. In pr_5 only pairs of push operation may be unrelated.

Note that although pr_5 is part of the final linearization order we are constructing, it is not guaranteed that $\mathsf{pr}_5 \cup \mathsf{ir}$ is cycle-free. The reason is that although ir is part of some linearization order, it is not necessarily part of the linearization order we construct here.

In a last step we construct a total order pr_6 with $\mathsf{pr}_5 \subseteq \mathsf{pr}_6$. In pr_6 we order all push operations which are not ordered in pr_5 such that pr_6 is a linearization order of the sub-history, i.e. $\langle \mathcal{A}', \mathsf{pr}_6, \mathsf{val} \rangle \in$ Stack. We show in Isabelle HOL that such a total order pr_6 always exists if the history is order-correct and linearizable with respect to Set. These conditions are satisfied by the assumptions of this theorem.

It remains to show that we can construct a linearization order not just for special histories which do not contain pop-empty operations and elimination pairs but for any history. We show this in two separate lemmata in Section 4.3. In Lemma 3, which deals with pop-empty operations, we show that if a history is linearizable with respect to Set, and the history without pop-empty operations is linearizable with respect to Stack, then the history is linearizable with respect to Stack. In Lemma 4, which deals with elimination, we show that if a history is linearizable with respect to Set, and the history without elimination pairs is linearizable with with respect to Stack, then the history is linearizable with respect to Stack.

As we satisfy the conditions of both Lemma 3 and Lemma 4, we can use them to show that there exists a linearization order pr^T such that $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val} \rangle \in$ Stack, which completes the proof.

To simplify the proof structure in Isabelle, we assume that in every execution, all elements which are pushed also get popped. We justify this by observing that any concurrent history where some elements do not get popped can be extended to one where all elements get popped. If the extended history is linearizable, then also the original history was linearizable.

□

The advantage of Theorem 1 is that orders on operations do not need to be established eagerly. In the example in Figure 4.1 on Page 37, push(a) and push(c) are unordered in ir, removing the need to decide their eventual linearization order; likewise pop()→a and pop()→c.

Our stack theorem is generic, not tied to the TS stack. It characterizes the internal ordering sufficient for an algorithm to achieve stack semantics. As well as sound, it is complete – for any linearizable stack, ir can be trivially projected from the linearization order. For CAS-based stacks such as Treiber's famous non-blocking stack [Tre86] it is simple to see intuitively why the theorem applies. If two pushes are ordered, then their CASes are ordered. As a result their elements will be ordered in the stack representation and removed in order.

Intuitively, our theorem seems close to the lower bound for stack ordering, as discussed in Section 4.1.2. We would expect any concurrent stack to enforce orders as strong as the ones in our theorem. Thus, Theorem 1 points towards fundamental constraints on the structure of concurrent stacks.

4.3 Proof Details of the Stack Theorem

This section gives more details about the proof of the stack theorem, especially about how we deal with pop-empty operations and elimination pairs. We start with axiomatized definitions of Stack and Set which are equivalent to the definitions in Section 4.1.3 but more convenient to use. We continue by stating a property of the precedence order of a history, the abcd property. We show that this property is always provided when a history is extracted from the trace of an execution. This abcd property is crucial for the correctness of the stack theorem. In the end of the section we prove the lemmata which allow us to deal with pop-empty operations and elimination separately.

4.3.1 Axiomatized Sequential Specifications

In the following definitions $\mathsf{isPush}(a)$ is true if a is a push operation, $\mathsf{isPop}(b)$ is true if b is a pop operation, and $\mathtt{emp}(e)$ is true if e is a pop operation which returns EMPTY.

Definition 8 (Set). *A sequential history* $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ *is in* Set *if and only if the following conditions hold:*

1. *No operation is both a push operation and a pop operation:*

$$\forall a, b \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \implies \\ \neg\exists c \in \mathcal{A}.\, c \xrightarrow{\mathsf{val}} a \wedge \neg\exists d \in \mathcal{A}.\, b \xrightarrow{\mathsf{val}} d$$

2. *An element is removed at most once:*

$$\forall a, b, c \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \wedge a \xrightarrow{\mathsf{val}} c \implies b = c$$

3. *Each pop operation removes at most one element:*

$$\forall a, b, c \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \wedge c \xrightarrow{\mathsf{val}} b \implies a = c$$

4. *Elements are inserted before they are removed:*

$$\forall a, b \in \mathcal{A}.\, a \xrightarrow{\mathsf{val}} b \implies a \xrightarrow{\mathsf{pr}} b$$

5. *A push operation only returns* EMPTY *if the set is actually empty:*

$$\forall e, a \in \mathcal{A}.\, \mathsf{emp}(e) \wedge \mathsf{isPush}(a) \wedge a \xrightarrow{\mathsf{pr}} e \implies \\ \exists b.\, a \xrightarrow{\mathsf{val}} b \wedge b \xrightarrow{\mathsf{pr}} e$$

Definition 9 (Stack)**.** *A sequential history $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ is in* Stack *if and only if the following conditions hold:*

1. $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ *is in* Set

2. *Elements are removed in a LIFO fashion:*

$$\forall +a, -a, +b \in \mathcal{A}.\, +a \xrightarrow{\mathsf{val}} -a \wedge +a \xrightarrow{\mathsf{pr}} +b \xrightarrow{\mathsf{pr}} -a \implies \\ \exists -b \in \mathcal{A}.\, +b \xrightarrow{\mathsf{val}} -b \wedge -b \xrightarrow{\mathsf{pr}} -a$$

4.3.2 The abcd Property of pr

Next we state a property of pr which exists whenever pr is derived from a trace $\mathcal{T}$ of an execution.

Lemma 2 (abcd)**.** *Let $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history derived from a trace $\mathcal{T}$. Given actions $a, b, c, d \in \mathcal{A}$, if $a \xrightarrow{\mathsf{pr}} b$ and $c \xrightarrow{\mathsf{pr}} d$, then either $a \xrightarrow{\mathsf{pr}} d$ or $c \xrightarrow{\mathsf{pr}} b$.*

Proof. Recall that a trace $\mathcal{T}$ is a total order on the events in an execution. If an event e occurs before an event f in $\mathcal{T}$, then we write $e <_{\mathcal{T}} f$. Additionally, for any $g \in \mathcal{A}$ we write the invocation event and response event of g in $\mathcal{T}$ as g_{inv} and g_{ret}, respectively.

With these notions defined we prove Lemma 2 with a case-split on whether $a_{\mathsf{ret}} <_{\mathcal{T}} d_{\mathsf{inv}}$. If $a_{\mathsf{ret}} <_{\mathcal{T}} d_{\mathsf{inv}}$, then $a \xrightarrow{\mathsf{pr}} d$. Otherwise, because $\mathcal{T}$ totally orders events, $d_{\mathsf{inv}} <_{\mathcal{T}} a_{\mathsf{ret}}$. The assumption gives us $c_{\mathsf{ret}} <_{\mathcal{T}} d_{\mathsf{inv}} <_{\mathcal{T}} a_{\mathsf{ret}} <_{\mathcal{T}} b_{\mathsf{inv}}$, which gives us $c \xrightarrow{\mathsf{pr}} b$ by transitivity.

□

Orders which provide the abcd property are sometimes called *interval order* [BEEH15].

4.3.3 Correctness of Emptiness Checks

Next we show that if a history is linearizable with respect to Stack without considering pop-empty operations, and the whole history is linearizable with respect to Set, then the whole history is linearizable with respect to Stack. This means that if pop-empty operations are correct according to set semantics, then they are also correct according to stack semantics.

Lemma 3 (correct emptiness check)**.** *Let $\mathcal{H}$ be a history, and let $\mathcal{H}'$ be the history obtained from $\mathcal{H}$ by removing all pop-empty operations. If $\mathcal{H}$ is linearizable with respect to* Set*, and $\mathcal{H}'$ is linearizable with respect to* Stack*, then $\mathcal{H}$ is linearizable with respect to* Stack.

Proof. We will only describe the key insights of the proof here and leave out technical details. A complete version of the proof is mechanized in Isabelle.

Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$, and let $\mathcal{H}' = \langle \mathcal{A}', \mathsf{pr}', \mathsf{val} \rangle$. As $\mathcal{H}$ is linearizable with respect to Set, and $\mathcal{H}'$ is linearizable with respect to Stack, there exists a linearization order pr^S with $\mathsf{pr} \subseteq \mathsf{pr}^S$ and $\langle \mathcal{A}, \mathsf{pr}^S, \mathsf{val} \rangle \in$ Set, and there exists a linearization order pr^T with $\mathsf{pr}' \subseteq \mathsf{pr}^T$ and $\langle \mathcal{A}', \mathsf{pr}^T, \mathsf{val} \rangle \in$ Stack. We show that $\mathcal{H}$ is linearizable with respect to Stack by constructing a linearization order lin with $\mathsf{pr} \subseteq \mathsf{lin}$ and $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val} \rangle \in$ Stack.

The linearization order lin is constructed as follows: the position of pop-empty operations $-e \in \mathcal{A}$ in pr^S is preserved in lin. This means for any operation $a \in \mathcal{A}$ that if $-e \xrightarrow{\mathsf{pr}^S} a$, then also $-e \xrightarrow{\mathsf{lin}} a$, and if $a \xrightarrow{\mathsf{pr}^S} -e$, then also $a \xrightarrow{\mathsf{lin}} -e$. Moreover, if two operations $a, b \in \mathcal{A}$ are ordered $a \xrightarrow{\mathsf{pr}^S} -e \xrightarrow{\mathsf{pr}^S} b$ and therefore by transitivity it holds that $a \xrightarrow{\mathsf{pr}^S} b$, then also $a \xrightarrow{\mathsf{lin}} b$.

For all other operations the order of pr^T is preserved. This means for any two operation $a, b \in \mathcal{A}'$ with $a \xrightarrow{\mathsf{pr}^T} b$, that if for all pop-empty operations $-e \in \mathcal{A}$ it holds that $-e \xrightarrow{\mathsf{pr}^S} a$ if and only if $-e \xrightarrow{\mathsf{pr}^S} b$, then $a \xrightarrow{\mathsf{lin}} b$.

First we show that $\mathsf{pr} \subseteq \mathsf{lin}$. By construction it holds that if $a \xrightarrow{\mathsf{lin}} b$ for any two operations $a, b \in \mathcal{A}$, then also either $a \xrightarrow{\mathsf{pr}^S} b$ or $a \xrightarrow{\mathsf{pr}^T} b$. As it cannot be for any $a, b \in \mathcal{A}$ with $a \xrightarrow{\mathsf{pr}} b$ that $b \xrightarrow{\mathsf{pr}^S} a$ or $b \xrightarrow{\mathsf{pr}^T} a$ (since they both contain either pr or pr'), it can also not be that $b \xrightarrow{\mathsf{lin}} a$. Since by construction lin is total on $\mathcal{A}$, this means that $\mathsf{pr} \subseteq \mathsf{lin}$.

Next we show that $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val} \rangle$ satisfies all condition of SET in Definition 8, which are:

1. No operation is both a push operation and a pop operation.
2. An element is removed at most once.
3. Each pop operation removes at most one element.
4. Elements are inserted before they are removed.
5. A pop operation only returns EMPTY if the set is actually empty.

The conditions 1, 2, and 3 of SET are trivially satisfied by $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val} \rangle$ as they only depend on val and val is the same in $\langle \mathcal{A}, \mathsf{pr}^S, \mathsf{val} \rangle$ as in $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val} \rangle$. Also condition 5 is satisfied trivially as all pop-empty operations $-e \in \mathcal{A}$ are ordered the same in lin as in pr^S and $\langle \mathcal{A}, \mathsf{pr}^S, \mathsf{val} \rangle$ satisfies condition 5.

We continue with condition 4. As $\langle \mathcal{A}, \mathsf{pr}^S, \mathsf{val} \rangle$ satisfies condition 5 of SET, there cannot exist a pop-empty $-e \in \mathcal{A}$ and operations $+a, -a, \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ such that $+a \xrightarrow{\mathsf{pr}^S} -e \xrightarrow{\mathsf{pr}^S} -a$. Therefore $+a$ and $-a$ are ordered in lin as they are ordered in pr^T, and as $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val} \rangle$ satisfies condition 4 of SET, also $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val} \rangle$ satisfies condition 4.

It only remains to be proved that all elements are removed in LIFO fashion. We have to show the following:

$$\begin{aligned} \forall +a, -a, +b \in \mathcal{A}. +a \xrightarrow{\mathsf{val}} -a \wedge +a \xrightarrow{\mathsf{pr}} +b \xrightarrow{\mathsf{pr}} -a \implies \\ \exists -b \in \mathcal{A}. +b \xrightarrow{\mathsf{val}} -b \wedge -b \xrightarrow{\mathsf{pr}} -a \end{aligned}$$

Assume $+a, -a, +b \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ and $+a \xrightarrow{\mathsf{lin}} +b \xrightarrow{\mathsf{lin}} -a$. As $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val} \rangle$ satisfies condition 5 of SET, there cannot be a pop-empty $-e \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{lin}} -e \xrightarrow{\mathsf{lin}} -a$. It can also not be that $+a \xrightarrow{\mathsf{lin}} -e \xrightarrow{\mathsf{lin}} +b$, as lin is an order and $+a \xrightarrow{\mathsf{lin}} -e \xrightarrow{\mathsf{lin}} +b$ would imply by transitivity that $+a \xrightarrow{\mathsf{lin}} -e \xrightarrow{\mathsf{lin}} -a$, a contradiction to condition 5 shown above. For the same reason $+b \xrightarrow{\mathsf{lin}} -e \xrightarrow{\mathsf{lin}} -a$ is impossible. Thus $+a, +b$, and $-a$ are ordered $+a \xrightarrow{\mathsf{lin}} +b \xrightarrow{\mathsf{lin}} -a$ because of $+a \xrightarrow{\mathsf{pr}^T} +b \xrightarrow{\mathsf{pr}^T} -a$. As $\langle \mathcal{A}, \mathsf{pr}^T, \mathsf{val}\rangle \in$ STACK, there has to exist a $-b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{pr}^T} -a$. For the same reason as above it cannot be that $-a \xrightarrow{\mathsf{pr}^S} -e \xrightarrow{\mathsf{pr}^S} -b$. Therefore $-b$ and $-a$ are ordered in lin the same as in pr^T, and $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val}\rangle$ satisfies condition 2 of STACK. Therefore $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val}\rangle$ satisfies all conditions of STACK.

□

4.3.4 Correctness of Elimination

Next we show that if a history is linearizable according to STACK without considering elimination pairs, then the history is also linearizable according to STACK with the elimination pairs. Note that in the next lemma we only require that the history is linearizable with respect to SET to make sure that the elimination pairs are well-formed, i.e. they consist of one push and one pop operation.

Lemma 4. *Let $\mathcal{H}$ be a history, and let $\mathcal{H}'$ be the history obtained from $\mathcal{H}$ by removing all elimination pairs. If $\mathcal{H}$ is linearizable with respect to* SET*, and $\mathcal{H}'$ is linearizable with respect to* STACK*, then also $\mathcal{H}$ is linearizable with respect to* STACK*.*

Proof. We will only describe the key insights of the proof here and leave out technical details. A complete version of the proof is mechanized in Isabelle.

We do an induction on the number of elimination pairs in $\mathcal{H}$.

Induction base ($n = 0$): If $\mathcal{H}$ is a history which contains 0 elimination pairs, then $\mathcal{H}'$ is $\mathcal{H}$. As we assume that $\mathcal{H}'$ is linearizable with respect to STACK, also $\mathcal{H}$ is linearizable with respect to STACK.

Induction hypothesis: Let $\mathcal{H}_n$ be a history with n elimination pairs, and let $\mathcal{H}_0$ be the sub-history of $\mathcal{H}_n$ without elimination pairs. If $\mathcal{H}_0$ is linearizable with respect to STACK, and $\mathcal{H}_n$ is linearizable with respect to SET, then also $\mathcal{H}_n$ is linearizable with respect to STACK.

Induction step: Let $\mathcal{H}_{n+1} = \langle \mathcal{A}_{n+1}, \mathsf{pr}, \mathsf{val}\rangle$ be a history with $n+1$ elimination pairs, let $\mathcal{H}_0$ be the sub-history of $\mathcal{H}_{n+1}$ without elimination pairs, and let $(+a, -a)$ be an elimination pair in $\mathcal{H}_{n+1}$. Let $\mathcal{H}_n$ be the sub-history of $\mathcal{H}_{n+1}$ where $+a$ and $-a$ are removed, i.e. for the set of operations $\mathcal{A}_n$ of $\mathcal{H}_n$ it holds that $\mathcal{A}_n = \mathcal{A}_{n+1} \setminus \{+a, -a\}$. Therefore $\mathcal{H}_n$ is a history with n elimination-pairs. Assume $\mathcal{H}_0$ is linearizable with respect to Stack. Then according to the induction hypothesis $\mathcal{H}_n$ is linearizable with respect to Stack. Therefore there exists a linearization order pr_n of $\mathcal{H}_n$ with $\langle \mathcal{A}, \mathsf{pr}_n, \mathsf{val}\rangle \in$ Stack.

We show now that $+a$ and $-a$ can be inserted into pr_n such that the resulting order lin is a linearization order of $\mathcal{H}_{n+1}$. We start by showing that there exists a position in pr_n where both $+a$ and $-a$ can be inserted. If for all operations $b \in \mathcal{A}_n$ it holds that $d \overset{\mathsf{pr}}{\not\to} +a$ and $d \overset{\mathsf{pr}}{\not\to} -a$, then $+a$ and $-a$ can be inserted as the first operations in lin. Otherwise there exists a first operation c in pr_n such that for all $d \in \mathcal{A}_n$ with $c \xrightarrow{\mathsf{pr}_n} d$ it holds that $d \overset{\mathsf{pr}}{\not\to} +a$ and $d \overset{\mathsf{pr}}{\not\to} -a$. In that case $+a$ and $-a$ are inserted right after c.

We have to show now that the resulting order still contains pr. This means that we have to show that for any operation $d \in \mathcal{A} \setminus \{+a, -a, c\}$ and $a \in \{+a, -a\}$ that if $a \xrightarrow{\mathsf{pr}} d$, then $c \xrightarrow{\mathsf{pr}_n} d$ and therefore $a \xrightarrow{\mathsf{lin}} d$, and if $d \xrightarrow{\mathsf{pr}} a$, then $d \xrightarrow{\mathsf{pr}_n} c$ and therefore $d \xrightarrow{\mathsf{lin}} a$.

We assumed above that for all $d \in \mathcal{A}_n$ with $c \xrightarrow{\mathsf{pr}_n} d$ it holds that $d \overset{\mathsf{pr}}{\not\to} +a$ and $d \overset{\mathsf{pr}}{\not\to} -a$. Therefore, since pr_n is a total order, if $d \xrightarrow{\mathsf{pr}} +a$ or $d \xrightarrow{\mathsf{pr}} -a$, then $d \xrightarrow{\mathsf{pr}_n} c$ and thus $d \xrightarrow{\mathsf{lin}} +a$ and $d \xrightarrow{\mathsf{lin}} -a$.

Next we show that if $+a \xrightarrow{\mathsf{pr}} d$ or $-a \xrightarrow{\mathsf{pr}} d$ for some $d \in \mathcal{A}$, then $+a \xrightarrow{\mathsf{lin}} d$ and $-a \xrightarrow{\mathsf{lin}} d$. We do this by showing that there cannot exist a $d \in \mathcal{A}_n$ with $d \xrightarrow{\mathsf{pr}_n} c$ and $a \xrightarrow{\mathsf{pr}} d$ for some $a \in \{+a, -a\}$. Assume such a $d \in \mathcal{A}_n$ existed with $d \xrightarrow{\mathsf{pr}_n} c$ and $a \xrightarrow{\mathsf{pr}} d$. We observe that since c is the first operation in pr_n such that for all $b \in \mathcal{A}_n$ with $c \xrightarrow{\mathsf{pr}_n} b$ it holds that $b \overset{\mathsf{pr}}{\not\to} +a$ and $b \overset{\mathsf{pr}}{\not\to} -a$, it has to be true that $c \xrightarrow{\mathsf{pr}} a'$ for some $a' \in \{+a, -a\}$.

As $a \xrightarrow{\mathsf{pr}} d$ and $c \xrightarrow{\mathsf{pr}} a'$, according to the abcd property of pr (see Lemma 2) either $c \xrightarrow{\mathsf{pr}} d$ or $a \xrightarrow{\mathsf{pr}} a'$. $c \xrightarrow{\mathsf{pr}} d$ is impossible as we assumed that $\mathsf{pr} \subseteq \mathsf{pr}_n$ and $d \xrightarrow{\mathsf{pr}_n} c$. If $a = a'$, then $a \xrightarrow{\mathsf{pr}} a'$ is also impossible because pr is irreflexive. If $+a \xrightarrow{\mathsf{pr}} -a$, then there would be a violation of the assumption that $(+a, -a)$ is an elimination pair, and if $-a \xrightarrow{\mathsf{pr}} +a$, than that would be a violation of the assumption that $\mathcal{H}_{n+1}$ is linearizable with respect to Set, as of condition 4 says that elements are inserted before they are removed.

Therefore there always exists a position in pr_n where both $+a$ and $-a$ can be inserted validly. It is trivial to show that the resulting sequential history $\langle \mathcal{A}, \mathsf{lin}, \mathsf{val}\rangle$ is within Stack.

□

Chapter 5

Correctness of the TS Stack

In this chapter we show that the TS stack is linearizable with respect to Stack. We use a two-level argument to separate concerns in the proof. By verifying a lower-level structure first, we hide the complexities of the data structure from the higher-level proof. The two proof steps are:

1. Prove the linearizability of an intermediate structure called the *TS buffer*. This shows that the SP stack pools combine to form a single consistent pool, but this consistent pool does not enforce LIFO ordering.

2. Use our stack theorem (Theorem 1) to prove that the TS stack is linearizable with respect to stack semantics. Linearizability lets us use the lower-level TS buffer in terms of its sequential specification.

5.1 TS Buffer

The TS buffer is a 'virtual' intermediate data structure, i.e. a proof convenience that does not exist in the algorithm syntax. (It would be easy to add, but would make our code more complex). The TS buffer methods are the lines in push and pop which modify the `spPools` array and thread-specific pools. Proving the TS buffer linearizable means these lines can be treated as atomic. We name the TS buffer operations as follows – line numbers refer to Listing 3.1 and Listing 3.2. Note that where possible these names coincide with names in these code listing.

- `ins` – inserts an element into an SP stack pool (line 13).

- `newTimestamp` – generates a new timestamp (line 14).

- `setTimestamp` – assigns a timestamp to a SP stack pool element (line 15).
- `getStart` – records the current time at the beginning of a pop (line 19).
- `tryRem` – searches through the SP stack pools and try to remove the element with the youngest timestamp (line 23).

Note that `newTimestamp` and `getStart` have the same underlying implementation, but different abstract specifications. This is because they play different roles in the TS stack: respectively, generating timestamps for elements, and controlling elimination.

The abstract state of the TS buffer hides individual SP stack pools by merging all the elements into a single pool. As elements may be eliminated depending on when the method started, the abstract state also records snapshots representing particular points in the buffer's history.

As with Stack and Set, we define the sequential specification TSbuf by tracking updates to abstract states. Formally, we assume a set of *buffer identifiers*, ID, representing individual buffer elements; and a set of *timestamps*, TS, with strict partial order $<_{\mathsf{TS}}$ and top element $\top$.

A TSbuf abstract state is a pair (B, S). $B \in \mathsf{Buf}$ is a partial map from identifiers to value-timestamp pairs, representing the current values stored in the buffer. $S \in \mathsf{Snapshots}$ is a partial map from timestamps to Buf, representing snapshots of the buffer at particular timestamps.

$$\mathsf{Buf} = \{B | B \colon \mathsf{ID} \rightharpoonup (\mathsf{Val} \times \mathsf{TS})\} \qquad \mathsf{Snapshots} = \{S | S \colon \mathsf{TS} \rightharpoonup \mathsf{Buf}\}$$

We implicitly assume that all timestamps in the buffer were previously generated by `newTimestamp`.

Snapshots are used to support globally consistent removal. Let (B, S) be the current abstract state of the TS buffer. To remove from the buffer, pop first calls `getStart` to generate a timestamp `t` – abstractly, $[\mathtt{t} \mapsto B]$ is added to the library of snapshots. When pop calls `tryRem(t)`, elements that were present when `t` was generated may be removed normally, while elements added or timestamped more recently than `t` may be eliminated. The stored snapshot $S(\mathtt{t})$ determines whether an element is removed normally or by elimination.

The TS buffer operations have the following specifications, assuming (B, S) is the abstract state before the operation:

- `newTimestamp()` – pick a timestamp $t \neq \top$ such that for all $t' \neq \top$ already in B, $t' <_{\mathsf{TS}} t$. Return t.

  Note that this means many elements can be issued the same timestamp if the thread is preempted before the new timestamp is attached to the element.

- `ins(v)` – Pick an ID $i \notin \mathrm{dom}(B)$. Update the state to $(B[i \mapsto (v,\top)], S)$ and return i.

- `setTimestamp(i,t)` – assume that `t` was generated by `newTimestamp()`. If $B(\mathtt{i}) = (v,\top)$, then update the abstract state to $(B[\mathtt{i} \mapsto (v,\mathtt{t})], S)$. If $B(\mathtt{i}) = \bot$, do nothing.

- `getStart()` – pick a timestamp $\mathtt{t} \neq \top$ such that $\mathtt{t} \notin \mathrm{dom}(S)$ or $\mathtt{t} \in \mathrm{dom}(S)$ and $S(\mathtt{t}) = B$. If $\mathtt{t} \notin \mathrm{dom}(S)$, update the state to $(B, S[\mathtt{t} \mapsto B])$. Return `t`.

- `tryRem(t)` – Assume $\mathtt{t} \in \mathrm{dom}(S)$. There are four possible behaviors:

  1. *failure.* Non-deterministically fail and return $\langle \mathsf{false}, \mathsf{null} \rangle$. This corresponds to a failed SP stack pool `remove` preempted by another thread.
  2. *emptiness check.* If $\mathrm{dom}(B) = \emptyset$, then return $\langle \mathsf{true}, \mathsf{EMPTY} \rangle$.
  3. *normal removal.* Pick an ID i with $i \in \mathrm{dom}(S(\mathtt{t})) \cap \mathrm{dom}(B)$ and $B(i) = (v_i, t_i)$ such that t_i is maximal with respect to other unremoved elements from the snapshot, i.e.

     $$\nexists i', t'.\, i' \in (\mathrm{dom}(S(\mathtt{t})) \cap \mathrm{dom}(B)) \wedge B(i') = (\_, t') \wedge t_i <_{\mathsf{TS}} t'$$

     Update the abstract state to $(B[i \mapsto \bot], S)$ and return $\langle \mathsf{true}, v_i \rangle$. Note that there may be many maximal elements that could be returned.
  4. *elimination.* Pick an ID i such that $i \in \mathrm{dom}(B)$, and either $i \notin \mathrm{dom}(S(\mathtt{t}))$ and $B(i) = (v, \_)$; or $S(\mathtt{t})(i) = (v, \top)$. Update the abstract state to $(B[i \mapsto \bot], S)$ and return $\langle \mathsf{true}, v \rangle$.

     This corresponds to the case where v was inserted or timestamped after pop called `getStart`, and v can therefore be removed using elimination.

Theorem 5. *The TS buffer is linearizable with respect to the specification* TSBUF.

Proof. The concrete state of the TS buffer consists of the array `spPools`, where each slot points to a SP stack pool, i.e. a linked list of nodes. For the abstract state, the mapping Buf is easily built by erasing taken nodes. We build Snapshots by examining the preceding trace. Snapshots are generated from the state of the buffer at any point `getStart` is called.

`ins` and `setTimestamp` are SP stack pool operations which take effect atomically because they build on atomic operations, i.e. the assignment to `top` in line 78, and to `timestamp` in line 15, respectively.

`newTimestamp` and `getStart` both build on the same timestamping operation. Only concurrent timestamp requests can generate unordered timestamps. As timestamps have to be generated and then added to the buffer separately, at the invocation of `newTimestamp` and `getStart` a timestamp which is not ordered with the newly created one cannot be in the buffer. For `getStart`, the snapshot is correctly constructed automatically as a consequence of the mapping from concrete to abstract state.

The most complex proof is for `tryRem`.The cases of `tryRem` returning `<false,NULL>` (line 58 in the emptiness check and line 111 when setting the `taken` flag fails) are trivially correct. The linearizability of `tryRem` returning `<true,EMPTY>` has been shown in [HHK$^+$13]: if the recorded `top` pointers have not changed, then there must have existed a point in time where the abstract state of the TS buffer contained no elements.

In normal removal (case 3) and elimination (case 4), the linearization point is the successful `CAS` to set the `taken` flag of an item in line 99. With the `taken` flag an element is removed atomically from the concrete state, so we only need to show that the correct element is removed.

We start with some properties of the elements in the buffer. Let $S(\mathtt{t})$ be the snapshot of the TS buffer created by `getStart`. Any element in $S(\mathtt{t})$ must have been in a SP stack pool before `tryRem` begins, and elements that are not in $S(\mathtt{t})$ are guaranteed to have timestamps younger than `t`.

Next we identify invariants of the SP stack pool.

1. If a node is reachable from the `top` pointer of a SP stack pool once, then it remains reachable at least as long as its `taken` flag is not set.

2. If a node b is reachable from a node a once by a sequence of `next` pointers, then b remains reachable from a as long as its `taken` flag is not set.

3. The nodes in the list are sorted by their timestamps. Thereby it is guaranteed that the node with the youngest timestamp in the SP stack pool is contained in the first node in the list which is not marked as `taken`.

With these invariants we show that the node returned by `getYoungest` is either the element in the snapshot with the youngest timestamp, or the element of the node is not in $S(\mathtt{t})$. Invariant 1 and invariant 2 guarantee that all nodes in $S(\mathtt{t})$ are reachable from `top` (line 87) by following `next` pointers (line 94). Therefore the node with the youngest timestamp will be found eventually. With invariant 3 it is guaranteed that the node with the youngest timestamp in $S(\mathtt{t})$ will be the first node encountered in the iteration which is not marked as `taken`. Therefore at the time the node is found it is indeed the node with the youngest timestamp in the SP stack pool which is also in $S(\mathtt{t})$. Alternatively the first not-`taken` node in the list may not be in $S(\mathtt{t})$, in which case returning that node is also correct according to TSBUF.

Now, suppose that `tryRem` calls `remove` of a SP stack pool in line 62 in listing 3.2. Comparison with `startTime` on line 45 ensures that elements not in the snapshot are not considered for normal removal. Suppose `tryRem` removes an element a although there exists an element b in the snapshot and in the TS buffer with $a <_{\mathsf{TS}} b$. Therefore b is contained in a SP stack pool and `getYoungest` must have returned b or some even younger element c. In this case, b or c would be considered younger than a by the timestamp comparison (line 47), and either removed or eliminated, contradicting our assumption. Note that b would also have been considered younger if b was timestamped after it got returned by `getYoungest`.

Suppose alternatively that `tryRem` removes an element a on line 46. If a is not in $\mathrm{dom}(S(\mathtt{t}))$, then the removal of a is correct because of elimination. If a is in $\mathrm{dom}(S(\mathtt{t}))$, then the timestamp of a was generated after the snapshot or the timestamp was assigned to a after the snapshot. Therefore the timestamp of a in the snapshot is $\top$ and the removal is correct because of elimination.

□

5.2 The TS Stack Theorem

With the TS buffer, the TS stack provides a natural insert-remove relation: a push operation is ordered before a pop operation if the element inserted into to TS buffer by the push is visible for the pop when it calls `getStart`. We call this insert-remove relation vis for visibility. However, vis does not satisfy all conditions of order-correctness.

In the following we define a property equivalent to order-correctness which is based on vis, and which matches the TS stack perfectly. We call this property TS-order-correctness. The definition of TS-order-correctness is based on the already mentioned insert-remove relation vis, and on a remove-remove order rr. Both relations are derived from the TS stack using the linearization point method. As linearization points we use TS buffer operations – as the TS buffer is linearizable, we can treat these operations as atomic.

- vis ('visibility' – the element inserted by a push was visible to a pop). A push and non-empty pop are ordered in vis if SP stack pool insertion in the push (line 13) is ordered before recording the current time in the pop (line 19). Similar to ir, this relation is alternating (see Definition 3). However, it may, it may not be a witness for order-correctness.
- rr ('remove-remove' – two pops removed elements in order). Two non-empty pop operations are ordered in rr if their final successful `tryRem` operations (line 23) are ordered in the execution.

Note that we use different linearization points for vis and rr. Therefore it can happen that the two relations conflict, i.e. that $\mathsf{vis} \cup \mathsf{rr}$ is not cycle-free. This is, however, no problem as vis and rr are not necessarily part of the linearization order of the TS stack.

As with ins in the definition of order-correctness, it is useful to define a helper order ts ('timestamp') in the definition of TS-order-correctness. This order on push operations is imposed by precedence and vis transitivity. Informally, if two push operations are ordered in ts, then their elements are ordered by their timestamps.

Definition 10 (Derived order ts)**.** *Assume a history $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ and order vis on $\mathcal{A}$. Two push operations $+a, +b \in \mathcal{A}$ are related $+a \xrightarrow{\mathsf{ts}} +b$ if: $+a \xrightarrow{\mathsf{pr}} +b$; or $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c \in \mathcal{A}$; or $+a \xrightarrow{\mathsf{pr}} +d \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c, +d \in \mathcal{A}$.*

Definition 11 (TS-order-correct)**.** *We call $\mathcal{H}$ TS-order-correct if there exists an alternating relation vis and a total order rr on pop operations such that:*

1. *$\mathsf{pr} \cup \mathsf{vis}$ and $\mathsf{pr} \cup \mathsf{rr}$ are cycle-free; and*
2. *for all $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}} +b \xrightarrow{\mathsf{vis}} -a$, there exists $-b \in \mathcal{A}$ such that $+b \xrightarrow{\mathsf{val}} -b$ and either $+b \not\xrightarrow{\mathsf{pr}} -b$ or $-b \xrightarrow{\mathsf{rr}} -a$;*

Lemma 6. *Let $\mathcal{H}$ be a history which is linearizable with respect to* Set*. $\mathcal{H}$ is TS-order-correct if and only if $\mathcal{H}$ is order-correct.*

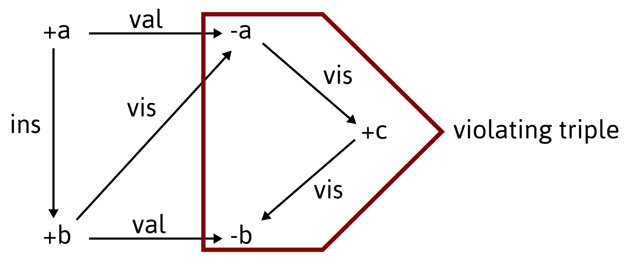


Figure 5.1: A violating triple.

Proof. The proof that order-correctness implies TS-order-correctness is trivial: from Theorem 1 we know that every history which is order-correct and linearizable with respect to Set is also linearizable with respect to Stack. Therefore there exists a linearization order of the history in Stack, and we can derive both vis and rr from that linearization order. It is trivial to show that these relations satisfy the conditions of TS-order-correctness.

The proof that TS-order-correctness implies order-correctness is more involved. We only sketch the structure here. For more details see Section 5.4. The proof is based on the insight that either vis is a witness that $\mathcal{H}$ is order-correct, or vis can be adjusted locally such that it becomes a witness.

By assumption $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free and therefore satisfies the first condition of order-correctness in Definition 5. Thus, if vis is not a witness it must violate condition 2, meaning

- there exist $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{vis}} -a$; and
- either no $-b \in \mathcal{A}$ exists such that $+b \xrightarrow{\mathsf{val}} -b$, or it exists and $-a \xrightarrow{\mathsf{rem}} -b$.

We show in Lemma 12.4 that whenever $+a \xrightarrow{\mathsf{ins}} +b$, also $+a \xrightarrow{\mathsf{ts}} +b$ holds. Therefore by the definition of TS-order-correctness there must exist a $-b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{rr}} -a$. As $\mathsf{pr} \cup \mathsf{rr}$ is cycle-free, $-a \xrightarrow{\mathsf{pr}} -b$ cannot hold. Thus $-a \xrightarrow{\mathsf{rem}} -b$ implies there exists a $+c \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{vis}} +c \xrightarrow{\mathsf{vis}} -b$. We call $(-a, +c, -b)$ a *violating triple.* Figure 5.1 illustrates a violating triple.

We show in Section 5.4 that all violating triples can be resolved iteratively, eventually resulting in a vis' which is a witness that $\mathcal{H}$ is order-correct.

□

We use the definition of TS-order-correctness to state the TS stack theorem similar to Theorem 1.

Theorem 7 (TS Stack Theorem). *Let C be a concurrent algorithm. If every history arising from C is TS-order-correct, and C is linearizable with respect to* SET*, then C is linearizable with respect to* STACK.

Proof. As every history $\mathcal{H}$ arising from C is TS-order-correct, according to Lemma 6 $\mathcal{H}$ is also order-correct. Therefore Theorem 1 applies and thus $\mathcal{H}$ is linearizable with respect to STACK.

□

5.3 TS Stack Linearizability

With the TS Stack Theorem we can now complete the linearizability proof of the TS stack. It only remains to show that the TS stack is linearizable with respect to SET, and that the TS stack is TS-order-correct.

Lemma 8. *The TS stack is linearizable with respect to* SET.

Proof. Straightforward from the fact that the TS buffer is linearizable with respect to TSBUF. We take the linearization point for push as the call to `ins` and the linearization point for pop as the call to `tryRem`. Correctness follows from the specification of TSBUF.

□

Lemma 9. *Every history $\mathcal{H}$ arising from the TS stack is TS-order-correct.*

Proof. To show that every history $\mathcal{H}$ arising from the TS stack is TS-order-correct we have to show that for $\mathcal{H}$ there exists an alternating relation vis and a total order of pop operations rr such that

1. $\mathsf{pr} \cup \mathsf{vis}$ and $\mathsf{pr} \cup \mathsf{rr}$ are cycle-free; and

2. for all $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}} +b \xrightarrow{\mathsf{vis}} -a$, there exists $-b \in \mathcal{A}$ such that $+b \xrightarrow{\mathsf{val}} -b$ and either $+b \overset{\mathsf{pr}}{\not\to} -b$ or $-b \xrightarrow{\mathsf{rr}} -a$;

Assume the history $\mathcal{H}$ is extracted from a trace $\mathcal{T}$ consisting of calls and returns to stack methods and atomic calls to the TS buffer methods. We write $a <_{\mathcal{T}} b$ if operations a and b are ordered in $\mathcal{T}$. As mentioned in Section 5.2 we define vis and rr as follows:

- Let $+a$ be a push operation, and let $\mathtt{ins}_{+a}$ be the `ins` operation of $+a$. Let $-b$ be a pop operation, and let $\mathtt{getStart}_{-b}$ be the `getStart` operation of $-b$. Then $+a \xrightarrow{\mathsf{vis}} -b$ if and only if $\mathtt{ins}_{+a} <_{\mathcal{T}} \mathtt{getStart}_{-b}$.

- Let $-a$, $-b$ be two pop operations, and let $\mathtt{tryRem}_{-a}$ and $\mathtt{tryRem}_{-b}$ be their final successful `tryRem` operation, respectively, then $-a \xrightarrow{\mathsf{rr}} -b$ if and only if $\mathtt{tryRem}_{-a} <_{\mathcal{T}} \mathtt{tryRem}_{-b}$.

The order rr is a subset of $<_{\mathcal{T}}$, and is therefore cycle-free. Moreover, pr is included in rr, which means that if two pop operations are ordered in pr, then they are also ordered in rr. Therefore $\mathsf{pr} \cup \mathsf{rr}$ is cycle-free. A similar argument shows that $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free.

Next we show that if two push operations are ordered in ts, then either (1) the inserted elements are ordered by $<_{\mathsf{TS}}$, or (2) the second element gets timestamped after the first element was removed.

Assume two push operations $+a$ and $+b$ are ordered in ts. Therefore either $+a \xrightarrow{\mathsf{pr}} +b$, or $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c \in \mathcal{A}$, or $+a \xrightarrow{\mathsf{pr}} +d \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} +b$ for some $-c, +d \in \mathcal{A}$. Let $\mathtt{setTimestamp}_{+a}$ be the `setTimestamp` operation of $+a$, and let $\mathtt{ins}_{+b}$ and $\mathtt{newTimestamp}_{+b}$ be the `ins` and `newTimestamp` operation of $+b$, respectively. Whenever $+a \xrightarrow{\mathsf{ts}} +b$, then it holds that $\mathtt{setTimestamp}_{+a} <_{\mathcal{T}} \mathtt{ins}_{+b}$. Therefore when $+b$ acquires a new timestamp either element a is in the TS buffer with a timestamp assigned, or it has already been removed.

If a is removed by a pop $-a$ before b is inserted, then condition 3 cannot be violated because it cannot be that that $+b \xrightarrow{\mathsf{vis}} -a$. Next we assume that a is removed after b gets inserted and $a <_{\mathsf{TS}} b$. According to condition 3 assume that $+b \xrightarrow{\mathsf{vis}} -a$, with $+a \xrightarrow{\mathsf{val}} -a$ and $+a \xrightarrow{\mathsf{pr}} -a$. This means that b is inserted into the TS buffer before $-a$ calls its `getStart` operation $\mathtt{getStart}_{-a}$. Therefore either b is removed before $-a$ calls `getStart` or b is within the snapshot generated by `getStart`.

If b is removed before $\mathtt{getStart}_{-a}$ then there must exist a pop operation $-b$ such that its `tryRem` operation $\mathtt{tryRem}_{-b}$ precedes the `tryRem` operation $\mathtt{tryRem}_{-a}$ of $-a$, satisfying condition 3.

Now assume that b is within the snapshot generated by $\mathtt{getStart}_{-a}$. $+a \xrightarrow{\mathsf{pr}} -a$ implies that a got timestamped before the snapshot is generated by $-a$, therefore a is removed by a normal remove, not by elimination. According to the TSBuf specification, `tryRem` removes a maximal element in the TS buffer which is also in the snapshot. As both a and b are in the snapshot generated by $\mathtt{getStart}_{-a}$ and $a <_{\mathsf{TS}} b$, this means that b must have been removed before `tryRem`$_{-a}$. Therefore there has to exist a pop operation $-b$ which removes b and its `tryRem` operation precedes `tryRem`$_{-a}$ in $\mathcal{T}$. This completes the proof.

□

Theorem 10. *TS stack is linearizable with respect to* Stack.

Proof. Lemma 8 deals with one clause of the TS stack theorem, Theorem 7, Lemma 9 deals with the other clause. This means that all conditions of the TS stack theorem are satisfied, and therefore the TS stack is linearizable with respect to Stack.

□

Theorem 11. *The TS stack is lock-free.*

Proof. Straightforward from the fact that the TS buffer is lock-free. This follows from the structure of `tryRem`: an attempt to remove an element can only fail when another thread preempts it. □

5.4 Formal Proof of the TS Stack Theorem

We only sketched the proof of Lemma 6, which says that order-correctness and TS-order correctness are equivalent. In this section we present a formal proof of Lemma 6. We recall Lemma 6:

Lemma 6. *Let $\mathcal{H}$ be a history which is linearizable with respect to* Set. *$\mathcal{H}$ is TS-order-correct if and only if $\mathcal{H}$ is order-correct.*

Proof. We showed already in Section 5.2 that order-correctness implies TS-order-correctness. Therefore we only need to show that TS-order-correctness implies order-correctness to complete the proof.

For the proof that TS-order-correctness implies order-correctness, it is sufficient to show that for histories without pop-empty operations and elimination pairs TS-order-correctness implies order-correctness. We have shown in Section 4.3 that if you can prove that all order-correct histories without pop-empty operations and elimination pairs are linearizable with respect to Stack, then also all other order-correct histories are linearizable with respect to Stack. Therefore, if we can show that all TS-order-correct histories without pop-empty operations and elimination pairs are order-correct and therefore linearizable with respect to Stack, then also all other TS-order-correct histories are linearizable with respect to STACK. ir can be derived directly from the linearization, and it is trivial to show that this ir relation is a witness that the history is order-correct.

Therefore we can state and prove a specialized version of Lemma 6 which deals only with histories without pop-empty operations and elimination pairs.

Lemma 12. *Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history without pop-empty operations and elimination pairs which is linearizable with respect to* SET*. If $\mathcal{H}$ is TS-order-correct, then it is also order-correct.*

Proof. Since $\mathcal{H}$ is TS-order-correct, there has to exist vis, an alternating relation on $\mathcal{A}$, rr, a total order on pop operations in $\mathcal{A}$, and the derived order ts (Definition 10) on push such that:

1. $\mathsf{pr} \cup \mathsf{vis}$ and $\mathsf{pr} \cup \mathsf{rr}$ are cycle-free; and
2. for all $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}} +b \xrightarrow{\mathsf{vis}} -a$, there exists $-b \in \mathcal{A}$ such that $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{rr}} -a$;

To show that $\mathcal{H}$ is order-correct, we have to show that there exists an alternating relation ir and derived orders ins and rem (Definition 4) such that:

1. $\mathsf{ir} \cup \mathsf{pr}$ is cycle-free; and
2. Let $+a, -a, +b \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ and $+a \xrightarrow{\mathsf{pr}} -a$. If $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{ir}} -a$, then there exists $-b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ and either $+b \not\xrightarrow{\mathsf{pr}} -b$ or $-a \not\xrightarrow{\mathsf{rem}} -b$;

The proof is based on the insight that either vis is already such an ir relation, or vis can be adjusted locally such that it becomes an ir relation.

By assumption $\mathsf{pr} \cup \mathsf{vis}$ is cycle-free and therefore satisfies condition 1 of Definition 5. Thus, if vis is not a witness it must violate condition 2, meaning

- there exist $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{vis}} -a$; and
- either no $-b \in \mathcal{A}$ exists such that $+b \xrightarrow{\mathsf{val}} -b$, or it exists and $-a \xrightarrow{\mathsf{rem}} -b$.

Remember that it cannot be that a $-b \in \mathcal{A}$ exists such that $+b \xrightarrow{\mathsf{val}} -b$ and $+b \not\xrightarrow{\mathsf{pr}} -b$. In that case $(+b, -b)$ would be an elimination pair, and we assumed that $\mathcal{H}$ does not contain any elimination pairs.

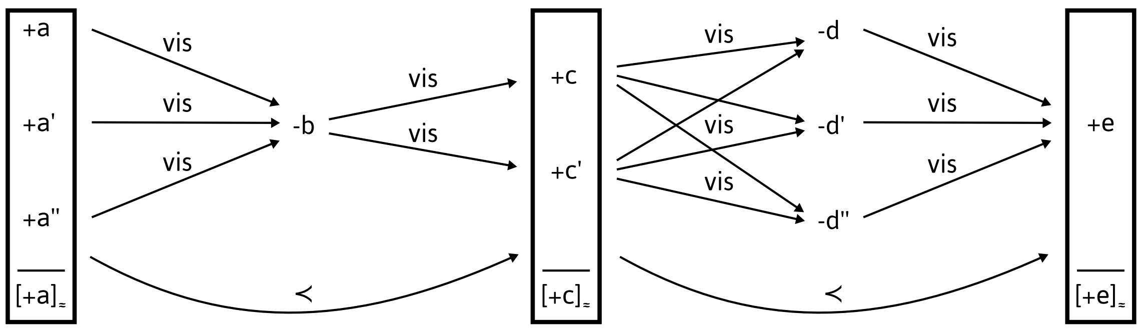


Figure 5.2: Illustration of a history with vis order. Push operations ordered the same in vis form an equivalence class, e.g. $+a$, $+a'$, and $+a''$ form the equivalence class $[+a]_{\approx}$. Equivalence classes are ordered by the total order $\prec$.

Now assume that $+a \xrightarrow{\mathsf{ins}} +b \xrightarrow{\mathsf{vis}} -a$. We show later in Lemma 12.4 that whenever $+a \xrightarrow{\mathsf{ins}} +b$, also $+a \xrightarrow{\mathsf{ts}} +b$. Therefore it holds that $+a \xrightarrow{\mathsf{ts}} +b \xrightarrow{\mathsf{vis}} -a$, and according to the definition of TS-order-correctness there exists a $-b$ with $-b \xrightarrow{\mathsf{rr}} -a$. As $\mathsf{pr} \cup \mathsf{rr}$ is cycle-free, $-a \xrightarrow{\mathsf{pr}} -b$ cannot hold. Thus $-a \xrightarrow{\mathsf{rem}} -b$ implies that there exists a $+c \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{vis}} +c \xrightarrow{\mathsf{vis}} -b$. We call $(-a, +c, -b)$ a *violating triple.*

We distinguish between two kinds of violating triples:

Forward triple. A violating triple $(-a, +c, -b)$ is called a *forward triple* if for all $+c' \in \mathcal{A}$ with $-a \xrightarrow{\mathsf{vis}} +c' \xrightarrow{\mathsf{vis}} -b$ there exists a $-c' \in \mathcal{A}$ with $+c' \xrightarrow{\mathsf{val}} -c'$ and $-c' \xrightarrow{\mathsf{rr}} -a$. This kind of violating triple is called forward triple because we will resolve it by moving $+c$ forwards in vis.

Backward triple. All violating triples which are not forward triples are called *backward triples.* Backward triples are resolved by moving $+b$ backwards. For every backward triple $(-a, +c, -b)$ there exists a violating triple $(-a, +c, -b)$ such that either there does not exist a $-c' \in \mathcal{A}$ with $+c' \xrightarrow{\mathsf{val}} -c'$, or such a $-c'$ exists and $-a \xrightarrow{\mathsf{rr}} -c'$.

For the resolution of violating triples we define an equivalence relation $\approx$ and an order relation $\prec$ derived from vis.

Definition 12 (equivalence relation $\approx$)**.** *Let $+a, +b \in \mathcal{A}$ be two push operations. We say that $+a \approx +b$ if and only if for all pop operations $-c \in \mathcal{A}$ it holds that $+a \xrightarrow{\mathsf{vis}} -c$ if and only if $+b \xrightarrow{\mathsf{vis}} -c$.*

We show in Lemma 12.2 that $\approx$ is an equivalence relation. We define $[+a]_{\approx}$ as the equivalence class of $+a$ in $\approx$. Note that since vis is alternating, it also holds that

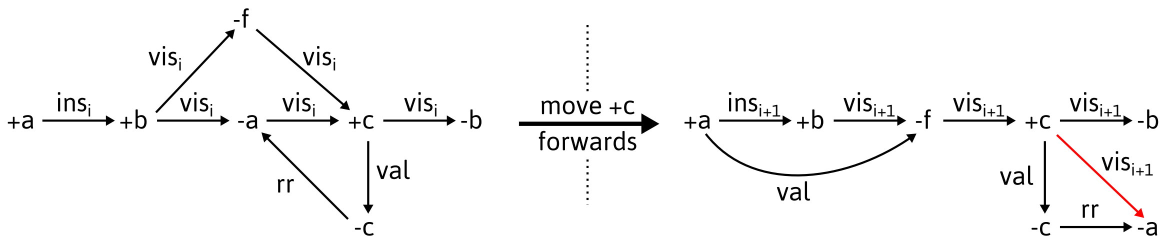


Figure 5.3: The forward triple $(-a,+c,-b)$ is resolved by moving $+c$ before $-a$. Red arrows indicate edges where vis_i and vis_{i+1} differ.

$+a \approx +b$ if and only if for all pop operations $-c \in \mathcal{A}$ it holds that whenever $-c \xrightarrow{\mathsf{vis}} +a$, also $-c \xrightarrow{\mathsf{vis}} +b$, and vice versa. Next we define a total order on the equivalence classes of $\approx$.

Definition 13 (total order $\prec$)**.** *Let $[+a]_\approx, [+b]_\approx$ be two equivalence classes of $\approx$, and let $+a \in [+a]_\approx$ and $+b \in [+b]_\approx$. We say that $[+a]_\approx \prec [+b]_\approx$ iff there exists a pop operation $-c \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} +b$.*

We show in Lemma 12.3 that $\prec$ is a total order on equivalence classes of $\approx$. Figure 5.2 illustrates $\approx$ and $\prec$. In the figure there exist three equivalence classes, $[+a]_\approx$, $[+c]_\approx$, and $[+e]_\approx$. As an example, $[+a]_\approx$ and $[+c]_\approx$ are ordered $[+a]_\approx \prec [+c]_\approx$ because $+a \xrightarrow{\mathsf{vis}} -b \xrightarrow{\mathsf{vis}} +c$ and $+a \in [+a]_\approx$ and $+c \in [+c]_\approx$.

Violating triples are resolved iteratively. vis_i is the modified vis order in the i-th iteration. The following invariants are preserved during the whole iteration. Let ts_i be the ts relation derived from vis_i:

1. vis_i is alternating; and
2. $\mathsf{pr} \cup \mathsf{vis}_i$ is cycle-free;
3. for all $+a, -a, +b \in \mathcal{A}$ such that $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, and $+a \xrightarrow{\mathsf{ts}_i} +b \xrightarrow{\mathsf{vis}_i} -a$, there exists $-b \in \mathcal{A}$ such that $+b \xrightarrow{\mathsf{val}} -b$ and $-b \xrightarrow{\mathsf{rr}} -a$;

The modification to resolve a forward triple $(-a,+c,-b)$ is illustrated in Figure 5.3. In vis_{i+1} $+c$ is ordered before $-a$, all other edges remain unchanged. In Figure 5.3 this means that $-f$, which is not part of the violating triple, is ordered $-f \xrightarrow{\mathsf{vis}_{i+1}} +c$, $-a$ is ordered $+c \xrightarrow{\mathsf{vis}_{i+1}} -a$. If there were other push operations $+c' \in [+c]^i_\approx$, then also $+c'$ would have been moved forwards. Formally vis_{i+1} is defined as

$$\begin{aligned}\mathsf{vis}_{i+1} := \{(x,y) \mid \; & ((x \neq -a \vee y \notin [+c]^i_\approx) \wedge (x,y) \in \mathsf{vis}_i) \\ \vee \; & (x \in [+c]^i_\approx \wedge y = -a)\}.\end{aligned}$$

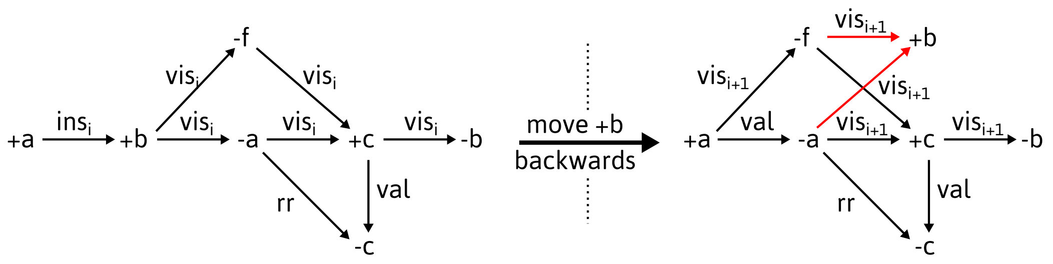


Figure 5.4: The backward triple $(-a,+c,-b)$ is resolved by moving $+b$ backwards to the same position as $+c$. Red arrows indicate edges where vis_i and vis_{i+1} differ.

Since only existing edges in vis_i are turned around and no new edges are created, vis_{i+1} is still alternating. In Lemma 12.5 and Lemma 12.6 we show that in vis_{i+1} also the other invariants are preserved.

The modification to resolve a backward triple $(-a,+c,-b)$ is illustrated in Figure 5.4, where $+b\in\mathcal{A}$ with $+b\xrightarrow{\mathsf{val}}-b$. After the modification $+b$ is ordered the same in vis_{i+1} as $+c$, all other edges remain unchanged. This means in Figure 5.4 that $-f$ is ordered $-f\xrightarrow{\mathsf{vis}_{i+1}}+b$ although $-f$ is not part of the violating triple and $+b\xrightarrow{\mathsf{vis}_i}-f$. If there were other push operations $+b'\in[+b]^i_{\approx}$, then these operations would remain unchanged. Formally vis_{i+1} is defined as follows: let $+c'\in[+c]_{\approx}$ such that either no $-c'\in\mathcal{A}$ exists with $+c'\xrightarrow{\mathsf{val}}-c'$, or there exists such a $-c'$ and $-a\xrightarrow{\mathsf{rr}}-c'$. Then

$$\begin{aligned}\mathsf{vis}_{i+1}:=\{(x,y)\mid\ &(x\neq+b\wedge y\neq+b\wedge(x,y)\in\mathsf{vis}_i)\\ \vee\ &(x=+b\wedge(+c',y)\in\mathsf{vis}_i)\\ \vee\ &(y=+b\wedge(x,+c')\in\mathsf{vis}_i)\}.\end{aligned}$$

Similar to above, since only existing edges in vis_i are turned around and no new edges are created, vis_{i+1} is still alternating. In Lemma 12.7 and Lemma 12.9 we show that also the other invariants still hold. Next we constrain the order in which violating triples are resolved:

1. Forward triples are resolved before backward triples.
2. If there exist two forward triples $(-a,+c,-b)$ and $(-a,+d,-b)$ and $[+c]_{\approx}\prec[+d]_{\approx}$, then $(-a,+c,-b)$ is resolved first.

As a consequence of these constraints we can assume that whenever we resolve a forward triple $(-a,+c,-b)$, then there does not exist a forward triple $(-a,+d,-b)$ with $+d\xrightarrow{\mathsf{vis}_i}-e\xrightarrow{\mathsf{vis}_i}+c$ for some $-e\in\mathcal{A}$. Additionally we can assume that whenever we resolve a backward triple $(-a,+c,-b)$, then there do not exist any forward triple.

Note that with each modification new violating triples may be introduced. We can show, however, that the iteration will eventually terminate. The proof is done by induction on the number of equivalence classes in $\approx$. We show that after finitely many iterations no operation of the n $\prec$-greatest equivalence classes will be part of any violating triples anymore. The proof is based on the insights that eventually all operations in the n $\prec$-greatest equivalence classes will remain in the n $\prec$-greatest equivalence class no matter how vis is modified to resolve further violating triples, and that operations in the n $\prec$-greatest equivalence classes will eventually only be moved forwards. As operations can only be moved forwards finitely many times without being moved backwards, eventually the iteration will terminate.

We use these insights to prove the following lemma:

Sub-Lemma 12.1. *For any $n \geq 1$ there exists a $N(n) \geq 0$ such that for any $k \leq n$ and $i \geq N(n)$ it holds that $[+d_k]^i_\approx = [+d_k]^{i+1}_\approx$, where $[+d_k]^i_\approx$ is the k $\prec$-greatest equivalence class in iteration i.*

Proof. We do a proof by induction on n.

Induction base ($n = 1$): We have to show that after some iteration $N(1)$ no push operation can join or leave the last equivalence class $[+d_1]_\approx$.

First we observe that operations $+b \in [+d_1]^g_\approx$ can never be moved backwards, for any iteration g, i.e. there cannot exist a backward triple $(-a, +c, -b)$ with $+b \xrightarrow{\mathsf{val}} -b$ in vis_g. The reason is that for any such violating triple it holds that $+b \xrightarrow{\mathsf{vis}_g} -a \xrightarrow{\mathsf{vis}_g} +c$ and therefore $+c$ is in a later equivalence class than $+b$.

Next we observe that if an operation $+d$ is in the last equivalence class $[+d_1]^g_\approx$ for some iteration g, then it will also be in the last equivalence class in all later iterations. The reason is that operations in the last equivalence class can never be moved backwards to split the last equivalence class, and if operations of the last equivalence class are moved forwards, they cannot be moved before operations of other equivalence classes. We show this in Lemma 12.11. This means that $[+d_1]_\approx$ only grows over time, it never shrinks.

As we only deal with finite histories, the last equivalence class can grow only finitely many times in the iteration. Therefore there exists an iteration $N(1)$ such that for any $i \geq N(1)$, $[+d_1]^i_\approx = [+d_1]^{i+1}_\approx$.

Induction hypothesis: We assume there exists a $N(n) \geq 0$ such that for any $k \leq n$ and $i \geq N(n)$ it holds that $[+d_k]^i_\approx = [+d_k]^{i+1}_\approx$, where $[+d_k]^i_\approx$ is the k-last equivalence class in iteration i.

Induction step: We begin with the observation that after the $N(n)$-th iteration no operation in the $(n+1)$-last equivalence class $[d_{n+1}]_{\approx}$ can be moved backwards because by that it would join one of the last n equivalence classes and thereby violate the induction hypothesis. Therefore operations in $[d_{n+1}]_{\approx}$ cannot be moved backwards after iteration $N(n)$.

Next we observe that if an operation $+d$ is in $[+d_{n+1}]^{g}_{\approx}$ for some iteration $g \geq N(n)$, then it will also be in $[+d_{n+1}]^{h}_{\approx}$ for any $h > g$. The reason is that operations in $[+d_{n+1}]^{g}_{\approx}$ can never be moved backwards to split the last equivalence class, and if operations are moved forwards, they cannot be moved before operations of other equivalence classes. We show this in Lemma 12.12. This means that after iteration $N(n)$ $[+d_{n+1}]_{\approx}$ can only grow over time, it never shrinks.

As we only deal with finite histories, $[+d_{n+1}]_{\approx}$ can grow only finitely many times in the iteration. Therefore there exists an iteration $N(n+1)$ such that for any $k \leq n+1$ and $i \geq N(n+1)$ it holds that $[+d_k]^{i}_{\approx} = [+d_k]^{i+1}_{\approx}$, where $[+d_k]^{i}_{\approx}$ is the k-last equivalence class in iteration i.

□

With Lemma 12.1 we showed that eventually all push operations will converge to a position where they will not be moved anymore and thus cannot be part of any violating triple anymore. Therefore the iteration will converge to a vis_{ir} which does not contain any violating triples anymore and therefore satisfies all conditions of a witness of $\mathcal{H}$ being order-correct.

□

□

5.4.1 Additional Proof Details

In the proof of Lemma 12 we left out some technical details to improve readability. In the following sections we will present these details. We start by showing that $\approx$ is an equivalence relation, and that $\prec$ is a total order. Afterwards we show that for any alternating relation vis and precedence order pr the derived order ins (Definition 4) is contained in the derived order ts (Definition 10). Next we show that the invariants of the vis relation used in the proof of Lemma 12 are preserved in each iteration of the proof. We conclude the section with the proof details needed to show that the iteration will eventually terminate.

Properties of $\approx$ and $\prec$

Sub-Lemma 12.2. $\approx$ *is an equivalence relation.*

Proof. Comes from the fact that the logical operator "if and only if" is reflexive, symmetric, and transitive.

□

Sub-Lemma 12.3. $\prec$ *is a total order on the equivalence classes of* $\approx$*.*

Proof. As we assume that we deal with a TS-order-correct history, we also assume that vis is alternating and cycle-free.

total. Assume two equivalence classes $[+a]_\approx$ and $[+b]_\approx$ with $+a \in [+a]_\approx$ and $+b \in [+b]_\approx$. Let $-c \in \mathcal{A}$ be a pop operation. Since vis is alternating, it holds that either $+a \xrightarrow{\mathsf{vis}} -c$ or $-c \xrightarrow{\mathsf{vis}} +a$. Similarly, either $+b \xrightarrow{\mathsf{vis}} -c$ or $-c \xrightarrow{\mathsf{vis}} +b$. If $+a \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} +b$, then $[+a]_\approx \prec [+b]_\approx$, and if $+b \xrightarrow{\mathsf{vis}} -c \xrightarrow{\mathsf{vis}} +a$, then $[+b]_\approx \prec [+a]_\approx$. If for all pop operations $-c \in \mathcal{A}$ $+a$ and $+b$ are ordered the same, then according to the definition of $\approx$ $[+a]_\approx = [+b]_\approx$.

transitive. Assume $[+a]_\approx \prec [+b]_\approx \prec [+c]_\approx$ with $+a \in [+a]_\approx$, $+b \in [+b]_\approx$, $+c \in [+c]_\approx$, and $+a \xrightarrow{\mathsf{vis}} -d \xrightarrow{\mathsf{vis}} +b \xrightarrow{\mathsf{vis}} -e \xrightarrow{\mathsf{vis}} +c$. As vis is alternating and cycle-free, this means that also $+a \xrightarrow{\mathsf{vis}} -e \xrightarrow{\mathsf{vis}} +c$, and therefore $[+a]_\approx \prec [+c]_\approx$.

antisymmetric. $\prec$ is a sub-relation of the transitive closure of vis. Since vis is cycle-free, also $\prec$ is cycle-free and as a consequence antisymmetric.

□

Properties of ts

Next we show that ins is contained in ts.

Sub-Lemma 12.4 (ins then ts)**.** *Let $+a, +b \in \mathcal{A}$ be two push operations. If $+a \xrightarrow{\mathsf{ins}} +b$, then also $+a \xrightarrow{\mathsf{ts}} +b$.*

Proof. Assume $+a, +b \in \mathcal{A}$ and $+a \xrightarrow{\mathsf{ins}} +b$. Therefore either $+a \xrightarrow{\mathsf{pr}} +b$ or there exists a $-c$ with $+a \xrightarrow{\mathsf{pr}} -c \xrightarrow{\mathsf{vis}} +b$. Both these conditions are also preconditions of the definition of ts, which completes the proof. □

Invariant Proofs for Forward Triples

In the following lemmata we show that the invariants of the iteration in the proof of Lemma 12 are preserved when resolving forward triples $(-a,+c,-b)$.

Sub-Lemma 12.5. *Let $(-a,+c,-b)$ be a forward triple in iteration i, and let vis_{i+1} be constructed by resolving this violating triple. If the invariants of the iteration hold for vis_i, then $\mathsf{pr} \cup \mathsf{vis}_{i+1}$ is cycle-free.*

Proof. As we described in the proof of Lemma 12, we define constraints on the order in which violating triples are resolved. Therefore we can assume here that there does not exist a violating triple $(-a,+f,-b)$ with $+f \xrightarrow{\mathsf{vis}_i} -e \xrightarrow{\mathsf{vis}_i} +c$, for any $-e \in \mathcal{A}$, because that violating triple would be resolved before $(-a,+c,-b)$.

Now, if there exists a cycle in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$, then it has to contain the edge $+c' \xrightarrow{\mathsf{vis}_{i+1}} -a$ with $+c' \in [+c]^i_\approx$ since all other edges existed already in vis_i and we assumed that $\mathsf{pr} \cup \mathsf{vis}_i$ is cycle-free. Therefore there has to exist a transitive edge from $-a$ to $+c'$ in $\mathsf{pr} \cup \mathsf{vis}_i$ which is preserved in the modification. The transitivity and abcd property of pr, vis_i being alternating, and $\mathsf{pr} \cup \mathsf{vis}_i$ being cycle-free implies that whenever such a transitive edge exists, then also one of the following edges exists:

1. $-a \xrightarrow{\mathsf{pr}} +c'$,
2. $-a \xrightarrow{\mathsf{vis}_i} +f \xrightarrow{\mathsf{pr}} +c'$ for some $+f \in \mathcal{A}$,
3. $-a \xrightarrow{\mathsf{vis}_i} +f \xrightarrow{\mathsf{vis}_i} -e \xrightarrow{\mathsf{vis}_i} +c'$ for some $+f, -e \in \mathcal{A}$, or
4. $-a \xrightarrow{\mathsf{pr}} -e \xrightarrow{\mathsf{vis}_i} +c'$ for some $-e \in \mathcal{A}$.

We check all cases. Let $-c' \in \mathcal{A}$ with $+c' \xrightarrow{\mathsf{val}} -c'$ and $-c' \xrightarrow{\mathsf{rr}} -a$ (which comes from the assumption that $(-a,+c,-b)$ is a forward triple).

Case 1: assume $-a \xrightarrow{\mathsf{pr}} +c'$, then there would exist the cycle $-a \xrightarrow{\mathsf{pr}} +c' \xrightarrow{\mathsf{pr}} -c' \xrightarrow{\mathsf{rr}} -a$, which would violate the assumption that $\mathsf{pr} \cup \mathsf{rr}$ is cycle-free. Note that we know that $+c' \xrightarrow{\mathsf{pr}} -c'$ because we assumed that the history $\mathcal{H}$ contains no elimination pairs.

Case 2: assume $-a \xrightarrow{\mathsf{vis}_i} +f \xrightarrow{\mathsf{pr}} +c'$ for some $+f \in \mathcal{A}$. Since $\mathsf{pr} \cup \mathsf{vis}_i$ is cycle-free it holds that $+f \xrightarrow{\mathsf{vis}_i} -b$ and also $(-a,+f,-b)$ is a violating triple. According to our assumption it holds that there exists a $-f \in \mathcal{A}$ with $-f \xrightarrow{\mathsf{rr}} -a$, and $(-a,+f,-b)$ is also a forward triple. If $+f \in [+c]^i_\approx$, then also $+f \xrightarrow{\mathsf{vis}_{i+1}} -a$ and no cycle is created. If $+f \notin [+c]^i_\approx$, then since $\mathsf{pr} \cup \mathsf{vis}_i$ is cycle-free there exists a $-e \in \mathcal{A}$ with $+f \xrightarrow{\mathsf{vis}_i} -e \xrightarrow{\mathsf{vis}_i}$

$+c'$. This, however violates our assumptions on the order in which we resolve violating triples, as described above.

Case 3: assume $-a \xrightarrow{\mathsf{vis}_i} +f \xrightarrow{\mathsf{vis}_i} -e \xrightarrow{\mathsf{vis}_i} +c'$ for some $+f, -e \in \mathcal{A}$. Because of $+c' \xrightarrow{\mathsf{vis}_i} -b$ this would imply that there exists a violating triple $(-a, +f, -b)$, with $+f \xrightarrow{\mathsf{vis}_i} -e \xrightarrow{\mathsf{vis}_i} +c'$, and therefore would violate our constraints on the order in which we resolve violating triples, as described above.

Case 4: assume $-a \xrightarrow{\mathsf{pr}} -e \xrightarrow{\mathsf{vis}_i} +c'$ for some $-e \in \mathcal{A}$. We assume that there exists a $-c'$ with $+c' \xrightarrow{\mathsf{val}} -c'$ and therefore $+c' \xrightarrow{\mathsf{pr}} -c'$. Applying the abcd property of Lemma 2 means that either $-a \xrightarrow{\mathsf{pr}} -c'$ or $+c' \xrightarrow{\mathsf{pr}} -e$. With the former there exists the cycle $-a \xrightarrow{\mathsf{pr}} -c' \xrightarrow{\mathsf{rr}} -a$, and with the later there exists the cycle $+c' \xrightarrow{\mathsf{pr}} -e \xrightarrow{\mathsf{vis}_i} +c$, both violating our assumptions that $\mathsf{pr} \cup \mathsf{rr}$ and $\mathsf{pr} \cup \mathsf{vis}_i$ are cycle-free. Therefore vis_{i+1} is cycle-free. □

Sub-Lemma 12.6. *Let ts_i and ts_{i+1} be the ts relation derived from vis_i and vis_{i+1}, respectively. Let vis_{i+1} be the result of resolving a forward triple $(-a, +c, -b)$ in iteration i. Let $+d, -d, +e \in \mathcal{A}$ with $+d \xrightarrow{\mathsf{val}} -d$, $+d \xrightarrow{\mathsf{pr}} -d$, and $+d \xrightarrow{\mathsf{ts}_{i+1}} +e \xrightarrow{\mathsf{vis}_{i+1}} -d$, then there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -d$.*

Proof. The current lemma holds for vis_i, and the only edges which are different in vis_i and in vis_{i+1} are those of $+c' \in [+c]^i_\approx$. Therefore we only have to show that $+c' \xrightarrow{\mathsf{vis}_{i+1}} -a$ is not part of a violation of this lemma. There exist two ways how $+c \xrightarrow{\mathsf{vis}_{i+1}} -a$ could be part of a violation:

1. $+a \xrightarrow{\mathsf{val}} -a$, $+a \xrightarrow{\mathsf{pr}} -a$, $+a \xrightarrow{\mathsf{ts}_{i+1}} +c' \xrightarrow{\mathsf{vis}_{i+1}} -a$, and either no $-c' \in \mathcal{A}$ exists with $+c' \xrightarrow{\mathsf{val}} -c'$ or such a $-c'$ exists and $-a \xrightarrow{\mathsf{rr}} -c'$. This cannot be true because $(-a, +c', -b)$ is a forward triple and therefore there exists a $-c' \in \mathcal{A}$ with $+c' \xrightarrow{\mathsf{val}} -c'$ and $-c' \xrightarrow{\mathsf{rr}} -a$.

2. There exist $+e, +f, -e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$, $+e \xrightarrow{\mathsf{pr}} -e$, $+e \xrightarrow{\mathsf{pr}} +c \xrightarrow{\mathsf{vis}_{i+1}} -a \xrightarrow{\mathsf{vis}_{i+1}} +f$ and therefore $+e \xrightarrow{\mathsf{ts}_{i+1}} +f$, $+f \xrightarrow{\mathsf{vis}_{i+1}} -e$, and either there does not exist a $-f \in \mathcal{A}$ with $+f \xrightarrow{\mathsf{val}} -f$ or such a $-f$ exists and $-e \xrightarrow{\mathsf{rr}} -f$.

   It cannot be the case that $+e \xrightarrow{\mathsf{ts}_i} +f$ because then $+e \xrightarrow{\mathsf{ts}_i} +f \xrightarrow{\mathsf{vis}_i} -e$ and the violation would have already existed in vis_i. As $+e \xrightarrow{\mathsf{ts}_i} +f$ would be constructed with any $-g$ with $+c' \xrightarrow{\mathsf{vis}_i} -g \xrightarrow{\mathsf{vis}_i} +f$, it has to be true that for all $-g \in \mathcal{A}$, if $+c' \xrightarrow{\mathsf{vis}_i} -g$, then also $+f \xrightarrow{\mathsf{vis}_i} -g$. Therefore $+f \in [+c]^i_\approx$, or $[+f]^i_\approx \prec [+c]^i_\approx$. In both cases it holds that $+f \xrightarrow{\mathsf{vis}_i} -b$, and with $-a \xrightarrow{\mathsf{vis}_i} +f$ it would hold that $(-a, +f, -b)$ is also a forward triple. This violating triple would be resolved before or at the same time as $(-a, +c, -b)$ and therefore $-a \xrightarrow{\mathsf{vis}_{i+1}} +f$ is not possible.

Therefore with both cases no violation is possible, which completes the proof. □

Invariant Proofs for Backward Triples

In the last section we showed that resolving forward triples preserves all the invariants stated in the proof of Lemma 12. We continue showing that also resolving backward triples preserves the invariants. Note that since forward triples are always resolved before backward triples, we can assume in this section that there do not exist any forward triples when we are resolving backward triples.

Sub-Lemma 12.7. *Let $(-a, +c, -b)$ be a backward triple in iteration i, and let vis_{i+1} be constructed by resolving this violating triple.*

If the invariants of the iteration hold for vis_i, then $\mathsf{pr} \cup \mathsf{vis}_{i+1}$ is cycle-free.

Proof. We assume without restriction of generality that we picked the violating triple $(-a, +c, -b)$ such that either there does not exist a $-c \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{val}} -c$, or such a $-c$ exists and $-a \xrightarrow{\mathsf{rr}} -c$.

Now, assume a cycle exists in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$, then the cycle has to contain $+b$, as all other operations are ordered the same as in vis_i and $\mathsf{pr} \cup \mathsf{vis}_i$ is cycle-free. By construction vis_{i+1} itself without pr is cycle-free, and therefore a potential cycle in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$ has to contain at least one pr edge.

As $+b$ is moved backwards, whenever for some operation $d \in \mathcal{A}$ there exists a transitive edge from $+b$ to d in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$, there also exists a transitive edge in $\mathsf{pr} \cup \mathsf{vis}_i$. Therefore a cycle has to contain a transitive edge from some $d \in \mathcal{A}$ to $+b$ in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$ which did not exist in $\mathsf{pr} \cup \mathsf{vis}_i$.

As $+b$ is moved to $+c$ in vis_{i+1} and a transitive edge from d to $+b$ is created in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$, a transitive edge from d to $+c$ has to have existed in $\mathsf{pr} \cup \mathsf{vis}_i$.

To close the cycle a transitive edge from $+b$ to d has to be part of the cycle which already existed in $\mathsf{pr} \cup \mathsf{vis}_i$ and is preserved in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$.

Any transitive edge from $+b$ to d in $\mathsf{pr} \cup \mathsf{vis}_i$ which starts with a vis_i edge (i.e. there exists some $-e \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{vis}_i} -e$ and a transitive edge from $-e$ to d in $\mathsf{pr} \cup \mathsf{vis}_i$) is not preserved in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$ because by transitivity $-e \xrightarrow{\mathsf{vis}_i} +c$ would hold and therefore $-e \xrightarrow{\mathsf{vis}_{i+1}} +b$. Thus a transitive edge from $+b$ to d in $\mathsf{pr} \cup \mathsf{vis}_i$ which is preserved has to start with a pr edge.

With the observation above and with the transitivity and abcd property of pr, vis_i being alternating, and $\mathsf{pr} \cup \mathsf{vis}_i$ being cycle-free, one of the following two kinds of transitive edges has to exist:

1. $+b \xrightarrow{\mathsf{pr}} -f \xrightarrow{\mathsf{vis}_i} +c$ for some $-f \in \mathcal{A}$; and

2. $+b \xrightarrow{\mathsf{pr}} +g \xrightarrow{\mathsf{vis}_i} -f \xrightarrow{\mathsf{vis}_i} +c$ for some $-f, +g \in \mathcal{A}$.

In both cases it holds that $+b \xrightarrow{\mathsf{ts}_i} +c$ and therefore $+b \xrightarrow{\mathsf{ts}_i} +c \xrightarrow{\mathsf{vis}_i} -b$ and therefore according to our assumptions there exists a $-c \in \mathcal{A}$ with $-c \xrightarrow{\mathsf{rr}} -b$. As we know from invariant 3 that $-b \xrightarrow{\mathsf{rr}} -a$, it holds by the transitivity of rr that $-c \xrightarrow{\mathsf{rr}} -a$, which contradicts our assumptions that $-a \xrightarrow{\mathsf{rr}} -c$. Therefore no cycle in $\mathsf{pr} \cup \mathsf{vis}_{i+1}$ is possible. □

Before proving the last invariant we first show a helping lemma:

Sub-Lemma 12.8. *Let $(-a, +c, -b)$ be a backward triple in iteration i, and let vis_{i+1} be constructed by resolving this violating triple. Let the invariants of the iteration hold for vis_i.*

Let $+s, -s, +e \in \mathcal{A}$ with $+s \neq +b$, $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}_{i+1}} +e \xrightarrow{\mathsf{vis}_{i+1}} -s$, then either also $+s \xrightarrow{\mathsf{ts}_i} +e \xrightarrow{\mathsf{vis}_i} -s$, or $+e = +b$ and $+s \xrightarrow{\mathsf{ts}_i} +c \xrightarrow{\mathsf{vis}_i} -s$.

Proof. First we observe that the only difference between vis_i and vis_{i+1} is that $+b$ has been moved backwards. Therefore it holds for any $+g, -h \in \mathcal{A}$ that if $+g \xrightarrow{\mathsf{vis}_{i+1}} -h$, than also $+g \xrightarrow{\mathsf{vis}_i} -h$ (otherwise $+g$ would have been moved forwards before $-h$). Therefore, if $+e \xrightarrow{\mathsf{vis}_{i+1}} -s$, it holds that $+e \xrightarrow{\mathsf{vis}_i} -s$, and in the case of $+e = +b$ also $+c \xrightarrow{\mathsf{vis}_i} -s$. It only remains to show that if $+s \xrightarrow{\mathsf{ts}_{i+1}} +e$, then either also $+s \xrightarrow{\mathsf{ts}_i} +e$, or $+e = +b$ and $+s \xrightarrow{\mathsf{ts}_i} +c$. We do a case distinction on the definition of ts_{i+1}.

- if $+s \xrightarrow{\mathsf{pr}} +e$, then also $+s \xrightarrow{\mathsf{ts}_i} +e$.

- if $+s \xrightarrow{\mathsf{pr}} -k \xrightarrow{\mathsf{vis}_{i+1}} +e$ for some $-k \in \mathcal{A}$, and $+e \neq +b$, then also $-k \xrightarrow{\mathsf{vis}_i} +e$ because vis_i and vis_{i+1} only differ in the order of $+b$. Therefore $+s \xrightarrow{\mathsf{ts}_i} +e$. If $+e = +b$, then $+c$ is ordered the same in vis_i as $+b$ in vis_{i+1} and therefore $+s \xrightarrow{\mathsf{pr}} -k \xrightarrow{\mathsf{vis}_i} +c$ and $+s \xrightarrow{\mathsf{ts}_i} +c$.

- if $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}_{i+1}} -k \xrightarrow{\mathsf{vis}_{i+1}} +e$ for some $+l, -k \in \mathcal{A}$, then, as we observed already before, it holds that $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}_i} -k \xrightarrow{\mathsf{vis}_{i+1}} +e$. If $+e \neq +b$, then also $-k \xrightarrow{\mathsf{vis}_i} +e$ because vis_i and vis_{i+1} only differ in the order of $+b$, and therefore $+s \xrightarrow{\mathsf{ts}_i} +e$. If $+e = +b$, then $+c$ is ordered the same in vis_i as $+b$ in vis_{i+1} and therefore $+s \xrightarrow{\mathsf{pr}} +l \xrightarrow{\mathsf{vis}_i} -k \xrightarrow{\mathsf{vis}_i} +c$ and $+s \xrightarrow{\mathsf{ts}_i} +c$.

□

We continue proving the last invariant:

Sub-Lemma 12.9. *Let $(-a, +c, -b)$ be a backward triple in iteration i, and let vis_{i+1} be constructed by resolving this violating triple. Let the invariants of the iteration hold for vis_i.*

Let $+s, -s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}_{i+1}} +e \xrightarrow{\mathsf{vis}_{i+1}} -s$, then there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -s$.

Proof. Let $+s, -s, +e \in \mathcal{A}$ with $+s \xrightarrow{\mathsf{val}} -s$, $+s \xrightarrow{\mathsf{pr}} -s$, and $+s \xrightarrow{\mathsf{ts}_{i+1}} +e \xrightarrow{\mathsf{vis}_{i+1}} -s$. Assume $+e \neq +b$, then according to Lemma 12.8 also $+s \xrightarrow{\mathsf{ts}_i} +e \xrightarrow{\mathsf{vis}_i} -s$, and according to our assumptions there exists a $-e \in \mathcal{A}$ with $+e \xrightarrow{\mathsf{val}} -e$ and $-e \xrightarrow{\mathsf{rr}} -s$.

Next assume that $+e = +b$. From the assumption that $(-a, +c, -b)$ is a violating triple we know that $-b$ exists and $-b \xrightarrow{\mathsf{rr}} -a$. Moreover we assumed that $-a \xrightarrow{\mathsf{rr}} -c$ and therefore by transitivity also $-b \xrightarrow{\mathsf{rr}} -c$ holds. We showed in Lemma 12.8 that $+s \xrightarrow{\mathsf{ts}_i} +c \xrightarrow{\mathsf{vis}_i} -s$ and therefore by condition 3 it holds that $-c \xrightarrow{\mathsf{rr}} -s$. Again by the transitivity of rr it therefore holds that $-b \xrightarrow{\mathsf{rr}} -s$. As $-e = -b$ this means that $-e \xrightarrow{\mathsf{rr}} -s$, which completes the proof.

□

Proof Details for the Convergence of the vis Iteration

Sub-Lemma 12.10. *For all configurations vis_i there does not exist a violating triple $(-a, +c, -b)$ such that $+b$ with $+b \xrightarrow{\mathsf{val}} -b$ is within the last equivalence class of $\approx$ in $\prec$.*

Proof. As a precondition of a violating triple $(-a, +c, -b)$ it holds that $+b \xrightarrow{\mathsf{vis}_i} -a \xrightarrow{\mathsf{vis}_i} +c$. Therefore $[+b]^i_\approx \prec [+c]^i_\approx$, which concludes the proof.

□

Sub-Lemma 12.11. *Let $[+d]^i_\approx$ and $[+e]^{i+1}_\approx$ be the $\prec$-greatest equivalence classes of $\approx$ of vis_i and vis_{i+1}, respectively. Then $[+d]^i_\approx \subseteq [+e]_\approx i+1$.*

Proof. Proof by contradiction. Assume there exists an i such that a $+d'$ is in the $\prec$-greatest equivalence class $[+d]^i_\approx$ of vis_i but $+d' \notin [+d]^{i+1}_\approx$. Therefore there has to exist a $-e, +c \in \mathcal{A}$ with $+d' \xrightarrow{\mathsf{vis}_{i+1}} -e \xrightarrow{\mathsf{vis}_{i+1}} +c$. As either $+d' \xrightarrow{\mathsf{vis}_i} -e$ or $-e \xrightarrow{\mathsf{vis}_i} +c$ do not hold, either $+d'$ is moved forwards before $-e$, or $+c$ is moved backwards behind $-e$.

If $+d'$ is moved forwards, there has to exist a forward triple $(-e, +d', -f)$ for some $-f \in \mathcal{A}$, then all elements in $[+d]^i_{\approx}$ are moved forwards. As $-e \xrightarrow{\mathsf{vis}_{i+1}} +c$, $+c \notin [+d]^i_{\approx}$ and therefore there exists some $-g \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{vis}_i} -g \xrightarrow{\mathsf{vis}_i} +d'$. As vis_i is cycle-free and alternating, this implies that $(-e, +c, -f)$ is also a forward triple which is resolved before $(-e, +d', -f)$ is resolved, as we defined in our ordering constraints. Therefore $+d' \xrightarrow{\mathsf{vis}_{i+1}} -e \xrightarrow{\mathsf{vis}_{i+1}} +c$ cannot be constructed by moving $+d'$ forwards.

If $+c$ is moved backwards, there has to exist a backward triple $(-g, +h, -c)$ with $+c \xrightarrow{\mathsf{val}} -c$ and $-e \xrightarrow{\mathsf{vis}_i} +h$. As $+d' \xrightarrow{\mathsf{vis}_i} -e$, this means that $[+d]^i_{\approx} \prec [+h]^i_{\approx}$, which violates our assumption that $[+d]^i_{\approx}$ is the $\prec$-greatest equivalence class of vis_i. Therefore $+d' \xrightarrow{\mathsf{vis}_{i+1}} -e \xrightarrow{\mathsf{vis}_{i+1}} +c$ cannot be constructed by moving $+c$ backwards, which concludes the proof.

□

Sub-Lemma 12.12. *Let $N(n) \geq 0$ be such that for any $k \leq n$ and $i \geq N(n)$ it holds that $[+d_k]^i_{\approx} = [+d_k]^{i+1}_{\approx}$, where $[+d_k]^i_{\approx}$ is the k $\prec$-greatest equivalence class in iteration i.*

Let $i > N(n)$, and let $[+d_{n+1}]^i_{\approx}$ and $[+d_{n+1}]^{i+1}_{\approx}$ be the $n+1$ $\prec$-greatest equivalence classes in $\prec$ of vis_i and vis_{i+1}, respectively. If $+d' \in [+d_{n+1}]^i_{\approx}$, then also $+d' \in [+d_{n+1}]^{i+1}_{\approx}$.

Proof. Proof by contradiction. Assume there exists an $i > N(n)$ such that a $+d'$ is in the $n+1$ $\prec$-greatest equivalence class $[+d_{n+1}]^i_{\approx}$ of vis_i but $+d' \notin [+d_{n+1}]^{i+1}_{\approx}$. Therefore there has to exist a $-e, +c \in \mathcal{A}$ with $+d' \xrightarrow{\mathsf{vis}_{i+1}} -e \xrightarrow{\mathsf{vis}_{i+1}} +c$, and $+c \in [+d_{n+1}]^{i+1}_{\approx}$. As either $+d' \xrightarrow{\mathsf{vis}_i} -e$ or $-e \xrightarrow{\mathsf{vis}_i} +c$ do not hold, either $+d'$ is moved forwards before $-e$, or $+c$ is moved backwards behind $-e$.

If $+d'$ is moved forwards, there has to exist a forward triple $(-e, +d', -f)$ for some $-f \in \mathcal{A}$, then all elements in $[+d]^i_{\approx}$ are moved forwards. As $-e \xrightarrow{\mathsf{vis}_{i+1}} +c$, $+c \notin [+d]^i_{\approx}$ and therefore there exists some $-g \in \mathcal{A}$ with $+c \xrightarrow{\mathsf{vis}_i} -g \xrightarrow{\mathsf{vis}_i} +d'$. As vis_i is cycle-free and alternating, this implies that $(-e, +c, -f)$ is also a forward triple which is resolved before $(-e, +d', -f)$ is resolved, as we defined in our ordering constraints. Therefore $+d' \xrightarrow{\mathsf{vis}_{i+1}} -e \xrightarrow{\mathsf{vis}_{i+1}} +c$ cannot be constructed by moving $+d'$ forwards.

If $+c$ is moved backwards, there has to exist a backward triple $(-g, +h, -c)$ with $+c \xrightarrow{\mathsf{val}} -c$ and $-e \xrightarrow{\mathsf{vis}_i} +h$. As $+d' \xrightarrow{\mathsf{vis}_i} -e$, this means that $[+d_{n+1}]^i_{\approx} \prec [+h]^i_{\approx}$. This means that $[+h]^i_{\approx}$ is one of the n $\prec$-greatest equivalence classes, and according to our assumptions $[+h]^i_{\approx} = [+h]^{i+1}_{\approx}$. As $+c \notin [+h]^i_{\approx}$, also $+c \notin [+h]^{i+1}_{\approx}$. As a backward triple $(-g, +h, -c)$ would cause $+c \in [+h]^{i+1}_{\approx}$, the existence of a backward triple $(-g, +h, -c)$

would violate our assumption. Therefore $+c$ cannot be moved backwards behind $-e$, which finishes the proof.

$\square$

Chapter 6

TS Queue

In the previous chapters we discussed only the TS stack. Crucial for the correctness of the TS stack is that the element with the youngest timestamp is removed first. However, in principle we can also remove the oldest element first, and thereby turn the TS stack into a TS queue. Alternatively we could allow to remove either the youngest or the oldest element, and thereby construct a TS deque. Changing the TS stack into a TS queue or a TS deque requires three main changes:

1. Change the timestamp comparison operator in `tryRem`.
2. Change the `getYoungest` method of the SP stack pool into a `getOldest` method for the TS queue, or into a `getRightMost` and a `getLeftMost` method for the TS deque.
3. For the TS queue, remove elimination in `tryRem`. For the TS deque, enable it only for stack-like removal.

In the following we will discuss these changes for the TS queue and the TS deque.

6.1 TS Queue Implementation

The high-level structure of the TS queue is similar to the high-level structure of the TS stack. The TS queue marks elements with timestamps recording the order they were enqueued. Elements are dequeued according to this timestamp order. Also in the TS queue each thread $T_1 - T_n$ which is accessing the queue has an associated single-producer multi-consumer pool called *SP queue pool*, implemented as a linked

Listing 6.1: The TS queue algorithm. Part 2 is presented in Listing 6.2. The SP queue pool is defined in Listing 6.3 and described in Section 6.1.1, timestamps are discussed in Section 2.2.

```
114 TSQueue{
115  Node{
116    Element element,
117    Timestamp timestamp,
118    Node next
119  };
121  SPQueuePool[maxThreads] spPools;
122
123  void enqueue(Element element){
124    SPQueuePool pool=spPools[threadID];
125    Node node=pool.insert(element);
126    Timestamp timestamp=newTimestamp();
127    node.timestamp=timestamp;
128  }
```

list. Every element in the TS queue is stored in the SP queue pool of the thread that enqueued it.

Listing 6.1 and Listing 6.2 show the TS queue code in detail. The gray highlighted code is the code which is different to the TS stack. The SP pools which are used internally by the TS queue support the following operations:

- `insert` – inserts an element without attaching a timestamp, and return a reference to the new node.
- `getOldest` – returns a reference to the node in the pool with the oldest timestamp, together with the top pointer of the pool.
- `remove` – tries to remove the given node from the pool. Returns `true` and the element of the node if it succeeds, or `false` and `NULL` otherwise.

The implementation of the SP queue pool is discussed in Section 6.1.1. The timestamping algorithms which are used in the TS queue (the call to `newTimestamp`) are the same as for the TS stack, see Section 2.2.

There is no difference between the enqueue operation of the TS queue (Listing 6.1) and the push operation of the TS stack (Listing 3.1): First the un-timestamped element is inserted into the current thread's SP pool (line 125), then a fresh timestamp is generated (line 126), and finally the timestamp is assigned to the element (line 127).

All the differences between the TS stack and the TS queue are in the dequeue operation (Listing 6.2) and in the SP queue pool implementation. Similar to the pop

Listing 6.2: Part 2 of the TS queue algorithm. The SP queue pool is defined in Listing 6.3 and described in Section 6.1.1, timestamps are discussed in Section 2.2. The code lines which are different to the TS stack are highlighted in gray.

```
Element dequeue(){
  Bool success;
  Element element;
  do{
    Timestamp startTime=newTimestamp();
    <success,element>=tryRem(startTime);
  } while (!success);
  return element;
}

<Bool,Element> tryRem(Timestamp startTime){
  Node oldest=NULL;
  Timestamp timestamp=MAX;
  SPQueuePool pool;
  Node[maxThreads] empty;
  for each(SPQueuePool current in spPools){
    Node node;
    Node poolInsert;
    <node,poolInsert>=current.getOldest();
    // Emptiness check
    if(node==NULL){
      empty[current.ID]=poolInsert;
      continue;
    }
    Timestamp nodeTimestamp=node.timestamp;
    if(timestamp >_TS nodeTimestamp){
      oldest=node;
      timestamp=nodeTimestamp;
      pool=current;
    }
  }
  // Emptiness check
  if(oldest==NULL){
    for each(SPQueuePool current in spPools){
      if(current.insert!=empty[current.ID])
        return <false,NULL>;
    }
    return <true,EMPTY>
  }
  if(timestamp >_TS startTime)
    return <false,NULL>;
  return pool.remove(oldest);
}
}
```

of the TS stack the dequeue iteratively scans over all SP pools (line 144-159) and searches for a valid node to remove, where for the TS queue a valid node is one with an oldest timestamp. Therefore we use the $>_{\mathtt{TS}}$ operator for timestamp comparison instead of $<_{\mathtt{TS}}$. If removing the identified node succeeds (line 170), then its element is returned (line 136). Otherwise the iteration restarts.

A key difference between the dequeue of the TS queue and the pop of the TS stack is that in general we cannot use elimination to remove elements. Quite contrary, we do not even remove an element which was inserted after the dequeue started to scan over the pools. The reason is that there could be a second, older element inserted after the scan started which was, however, missed. To determine that an element was inserted after the start of the scan, we read the current time when `tryRem` is called (line 133). Any element which was inserted after the start of the scan will have a later timestamp and therefore we do not remove an element with a later timestamp than the start time of the scan (line 168).

The emptiness check of the TS queue is based on the same principle as the emptiness check of the TS stack: when scanning the SP queue pools, if a pool indicates that it is empty, then its `insert` pointer is recorded (lines 149-152). If no candidate for removal is found, then the SP queue pools are scanned again to check whether their `insert` pointers have changed (lines 162-165). If not, then no element has been inserted between the first and the second scan, and therefore at some time between the scans all SP queue pools were empty. The linearizability of this emptiness check has been proved in [HHK$^+$13].

6.1.1 SP Queue Pool

In this section we describe the internal data structure of the TS queue, the SP queue pool. The design of the SP queue pool is similar to the design of the Michael Scott queue [MS96], as illustrated in Figure 6.1.

The SP queue pool is a singly-linked list of nodes accessed by an `insert` and an `remove` pointer. Each node contains a `next` pointer for the linked list, the `element` it stores, and the `timestamp` of the element. Different to the SP stack pool no `taken` flag is required. The `insert` pointer points to the node of the element which has been inserted last, the `remove` pointer points to the node of the element which has been removed last. In Figure 6.1 the last inserted element is c, the last removed element is a. If the `insert` pointer and the `remove` pointer point to the same node, then all inserted elements have already been removed, and the SP queue pool is empty.

Listing 6.3: SP queue pool algorithm.

```
SPQueuePool{
 Node insert;
 Node remove;
 Int ID; // The ID of the owner thread.

 init(){
   Node sentinel=createNode(element=NULL,next=NULL);
   insert=sentinel;
   remove=sentinel;
 }

 Node insert(Element element){
   Node newNode=createNode(element=element,next=NULL);
   insert.next=newNode;
   insert=newNode;
   return newNode;
 }

 <Node,Node> getOldest(){
   Node oldRemove=remove;
   Node oldInsert=insert;
   if(oldRemove==oldInsert)
     return <NULL,oldInsert>;
   return <oldRemove.next,oldInsert>;
 }

 <Bool,Element> remove(Node node){
   if(CAS(remove,node,node.next)){
     return <true,node.element>;
   }
   return <false,NULL>;
 }
}
```

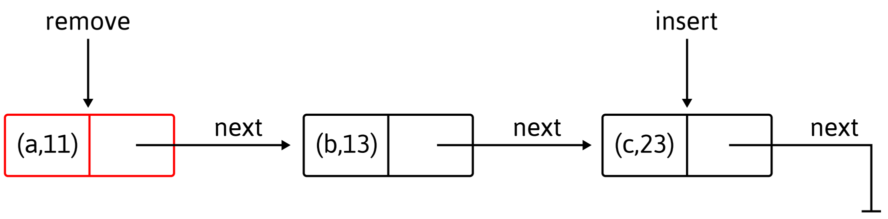


Figure 6.1: A SP queue pool containing two element, b and c. Element a has already been removed.

Initially the SP queue pool consists of a single sentinel node which contains no element. Both the `insert` and the `remove` pointer point to the sentinel node indicating that the SP queue pool is empty. The `remove` pointer is annotated with an ABA-counter to avoid the ABA-problem [HS08].

An element is inserted into the SP queue pool by storing it in a new node linked after the last inserted node. In Figure 6.1 the new node would be added after the node containing element *c*. An element is not considered in the SP queue pool as long as the insert pointer does not point to its node.

Since only a single thread inserts elements into the SP queue pool, and elements are always inserted at the end of the linked list, it is guaranteed that the elements in the SP queue pool are sorted by their age. Therefore the oldest element in the SP queue pool is stored in the first node in the linked list which has not been removed yet. This node is always the successor of the node the `remove` pointer points to. Therefore `getOldest` always returns `remove` $\rightarrow$ `next` after the emptiness check in line 194-195. In Figure 6.1 the oldest element in the SP pool is *b*.

An element is removed from the SP queue pool by moving the `remove` pointer to the next node in the list with a `CAS` instruction (line 200). This is sufficient because the TS queue always removes the oldest element in the SP queue pool first.

6.2 Correctness of the TS Queue

We use the queue theorem of [HSV13] to show the correctness of the TS queue. Similar to the stack theorem described in Section 4 the queue theorem defines a set of conditions, and any implementation which satisfies these conditions is linearizable with respect to the sequential specification of a queue.

We start by stating the sequential specification of a queue. Assume a set of values Val. Abstract states of a queue are finite sequences in Val^*. Let $\sigma \in \mathsf{Val}^*$ be an arbitrary state. In Queue, enqueue and dequeue have the following sequential behavior ('$\cdot$' means sequence concatenation):

- `enqueue(v)` – Update the abstract state to $\sigma \cdot [\mathtt{v}]$.
- `dequeue()` – If $\sigma = []$, return EMPTY. Otherwise, σ must be of the form $[v'] \cdot \sigma'$. Update the state to σ', return v'.

Next we define *order-correctness* for queues:

Definition 14 (order-correctness for queues)**.** *We call a history $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val}\rangle$ order-correct for queues if the following condition is satisfied:*

- *Let $+a, +b, -b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$. If $+a \xrightarrow{\mathsf{pr}} +b$, then there exists a $-a \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ and $-b \not\xrightarrow{\mathsf{pr}} -a$.*

With the definition of order-correctness for queues we can formulate the queue theorem of Henzinger et al. [HSV13] similar to our stack theorem:

Theorem 13 (Queue Theorem [HSV13])**.** *Let C be a concurrent algorithm. If every history arising from C is queue order-correct, and C is linearizable with respect to* Set*, then C is linearizable with respect to* Queue*.*

Proof. The theorem is based on the insight that if two enqueue operations $+a$ and $+b$ are ordered in pr (i.e. $+a \xrightarrow{\mathsf{pr}} +b$), then also their matching dequeue operations $-a$ and $-b$ with $+a \xrightarrow{\mathsf{val}} -a$ and $+b \xrightarrow{\mathsf{val}} -b$ can be ordered $-a \xrightarrow{\mathsf{pr}^T} -b$ in the linearization order pr^T. Contradicting order dependencies are only possible if also order-correctness is violated because of the abcd property of pr shown in Lemma 2. For more proof details see the original proof of the theorem in [HSV13].

□

Note that the queue theorem does not require the existence of an insert-remove relation as the stack theorem does. Therefore we can apply the queue theorem directly on the TS queue, we do not have to introduce a TS buffer first.

Theorem 14. *The TS queue is linearizable with respect to* Queue*.*

Proof. We use the queue theorem formulated in Theorem 13 to prove linearizability of the TS queue with respect to Queue. Thus it is sufficient to show that

1. the TS queue is linearizable with respect to Set; and

2. every history arising from the TS queue is queue order correct.

To prove the linearizability of the TS queue with respect to Set it is sufficient to show that elements are inserted and removed atomically, and that the emptiness check is correct. Elements are inserted into the TS queue when they are inserted in to the SP queue pool of the inserting thread. This insertion is done atomically when the `insert` pointer of the SP queue pool is changed in line 187. An element is removed atomically with the atomic `CAS` instruction in line 200. The pointer comparison in line 194 guarantees that elements are removed after they are inserted.

We reuse the emptiness check of [HHK+13] for the TS queue: if in one scan over all SP queue pools no element is found (line 144-159), and in a second scan over all SP queue pools the `insert` pointer of all SP queue pools are still the same (line 162-165), then all SP queue pools are empty between the first and the second scan. For more proof details see [HHK+13].

Next we show that every history arising from the TS queue is queue order correct according to Definition 14. Let $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be a history arising from an execution of the TS queue, and let $\mathcal{T}$ be the trace of atomic instructions of this execution. We write $x <_{\mathcal{T}} y$ if instructions x and y are ordered in $\mathcal{T}$. Assume $+a, +b, -b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ and $+a \xrightarrow{\mathsf{pr}} +b$. We have to show that there has to exist a $-a$ with $+a \xrightarrow{\mathsf{val}} -a$ and $-b \not\xrightarrow{\mathsf{pr}} -a$. Let a be the element inserted by $+a$, and let b be the element inserted by $+b$.

If $+a \xrightarrow{\mathsf{pr}} +b$, then also the `newTimestamp` operation of $+a$ is executed sequentially before the `newTimestamp` operation of $+b$. By the semantics of the timestamping algorithms it therefore always holds that $a <_{\mathsf{TS}} b$.

Now assume, towards a contradiction, that either there does not exist an operation $-a$ with $+a \xrightarrow{\mathsf{val}} -a$, or there exists such a $-a$ and $-b \xrightarrow{\mathsf{pr}} -a$. This means that at the time element b is found in the `tryRem` operation of $-b$ also a is contained in one of the SP queue pools. Note that because of the comparison with the `startTime` in line 168 the element b has already been in the SP queue pool when $-b$ started its `tryRem`. As a has been inserted strictly before b, also a has been in an SP queue pool when the successful `tryRem` was invoked within $-b$.

This means that when `getOldest` of a's SP queue pool is called in the successful `tryRem`, then either a or some element c with $c <_{\mathsf{TS}} a$ is returned. In both cases either $a <_{\mathsf{TS}} b$ or $c <_{\mathsf{TS}} b$ would hold. Therefore in the scan over the SP queue pools not b would be considered the oldest element in the TS queue but either a or c. This, however, contradicts the assumption that b is removed by $-b$.

□

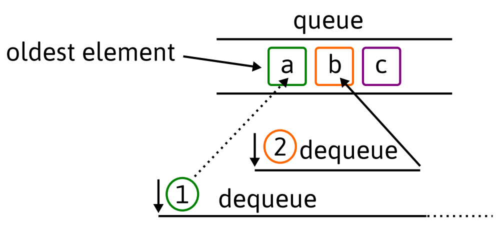


Figure 6.2: Illustration of the cooperation optimization. A dequeue operation can remove the second oldest element in a concurrent queue if it knows that there exists a concurrent dequeue operation which will remove the oldest element.

6.3 Variants of the TS Queue

We will show in Section 6.4 that the performance of the TS queue is significantly worse than the performance of the TS stack. The reason is that in general elimination, as it is implemented in the TS stack, is not possible in the TS queue[1].

Without elimination it is always necessary for a dequeue operation to scan all SP queue pools to search for the oldest element in the queue. This increases the time it takes to find the oldest element in the queue, and thereby it increases the potential for collisions between dequeue operation.

In this section we propose two variants of the TS queue. The first TS queue variant uses cooperation [HPS13] to minimize the potential for collisions between dequeue operations. However, this variant does not provide lock-freedom, and dequeue operations block if the TS queue is empty. The second TS queue variant uses an optimization which can be seen as an unsound version of elimination because dequeue operations try to remove elements before all SP queue pool have been scanned. This TS queue variant provides lock-freedom but sacrifices linearizability.

6.3.1 Initial Work: Cooperative TS Queue (CTS Queue)

Cooperation as an optimization for concurrent queue implementations has been proposed in [HPS13]. We describe this optimization with the example in Figure 6.2. The figure shows a queue with an oldest element a and a second oldest element b. Two dequeue operations try to access the queue concurrently. Without cooperation, these two dequeue operations compete for the oldest element in the queue, element a. One

[1]In special cases elimination is possible for queues [MNSS05]. Elimination is restricted to dequeue operations which could return EMPTY instead of removing an element with elimination.

of the dequeues will eventually succeeds, the other dequeue operation wasted time trying to get *a* and will eventually remove *b*.

With cooperation the first dequeue tells the second dequeue that it will eventually remove *a*. Thereby the second dequeue can consider *a* as removed and can immediately go for *b* without wasting time in the competition for *a*.

Implementation

To implement the cooperative TS queue (CTS queue) we do the following changes to the original TS queue algorithm, highlighted in gray in Listing 6.4: first we require the use of the TS-atomic timestamping algorithm which generates timestamps by incrementing a single global counter with an atomic fetch-and-increment instruction (line 215). Thereby we get deterministic timestamps, which means that a second instance of TS-atomic generates the exact same timestamps as the first instance.

Then we use a second instance of TS-atomic (line 216) to generate timestamps for dequeue operations (line 226). As long as there are more enqueue operations than dequeue operations, a deterministic timestamping algorithm guarantees that for each dequeue operation there exists exactly one element with the same timestamp. In other words, for a dequeue operation with timestamp *t* there exists exactly one element with timestamp *t*. In the CTS queue a dequeue operation with timestamp *t* searches through all SP queue pools (line 234-238) to find and remove the element with timestamp *t* (line 235).

For the CTS queue we cannot use the SP queue pool we used for the TS queue because the CTS queue requires to find the element with the same timestamp as the dequeue operation, not the element with the oldest timestamp. We show the adjusted SP queue pool in Listing 6.5, and call it the SP CTS pool. The main differences are:

- Elements are timestamped before they are inserted into the SP CTS pool. The reason is that elements without timestamps cannot be removed from the CTS queue.

- The `getOldest` operation and the `remove` operation of the SP queue pool are merged into a `findAndRemove` operation. The reason is that since there exists only a single element with the right timestamp for a dequeue operation, this element can be removed immediately when its matching dequeue operation finds it.

Listing 6.4: The CTS queue algorithm. The SP CTS pool is defined in Listing 6.5, TS-atomic is discussed in Section 2.2.

```
CTSQueue{
 Node{
   Element element,
   Timestamp timestamp,
   Node next,
   Bool taken
 };
 SPCTSPool[maxThreads] spPools;
 TSAtomic elementTimestamping;
 TSAtomic dequeueTimestamping;

 void enqueue(Element element){
   SPQueuePool pool=spPools[threadID];
   Timestamp timestamp=elementTimestamping.newTimestamp();
   pool.insert(element,timestamp);
 }

 Element dequeue(){
   Element element;
   Timestamp dequeueTS=dequeueTimestamping.newTimestamp();
   do{
     element=tryRem(dequeueTS);
   } while (element==NULL);
   return element;
 }

 Element tryRem(Timestamp dequeueTS){
   for each(SPQueuePool current in spPools){
     Node node=current.findAndRemove(dequeueTS);
     if (node != NULL)
       return node.element;
   }
   return NULL;
 }
}
```

Listing 6.5: SP CTS pool algorithm.

```
SPCTSPool{
 Node insert;
 Node remove;

 init(){
   Node sentinel=createNode(element=NULL,next=NULL,taken=true);
   insert=sentinel;
   remove=sentinel;
 }

 insert(Element element, Timestamp ts){
   Node newNode=createNode
     (element=element,next=NULL,taken=false,timestamp=ts);
   insert.next=newNode;
   insert=newNode;
 }

 Node findAndRemove(Timestamp dequeueTS){
   Node oldRemove=remove;
   Node oldInsert=insert;
   if(oldRemove==oldInsert)
     return NULL;

   Node prev=oldRemove;
   Node prevNext=prev.next;
   Node currentNode=prevNext;
   while(true) {
     if(currentNode==NULL)
       return NULL;
     if(dequeueTS==currentNode.timestamp) {
       Node next=currentNode.next;
       if (next!=NULL)
         CAS(prev.next,prevNext,next);
       currentNode.taken=true;
       return currentNode;
     } else if(dequeueTS<=currentNode.timestamp) {
       return NULL;
     } else if(!currentNode.taken) {
       prev=currentNode;
       prevNext=prev.next;
       currentNode=prevNext;
     } else {
       currentNode=currentNode.next;
     }
   }
 }
}
```

- In `findAndRemove` we do not return the first node in the list but we iterate through the linked list (line 268-286) and search for the element with the same timestamp as the current dequeue operation (line 271). Since the nodes in the linked list are sorted by their timestamps, we can stop the iteration when we encounter a node with a younger timestamp than the timestamp we are looking for (line 277). The list is sorted because timestamps increase monotonically over time, and nodes are always inserted at the end of the list.

- Elements can be removed from the middle of the linked list, not just at the end. Therefore we use a `taken` flag, similar to the SP stack pool where we also used the `taken` flag. Nodes marked as `taken` are unlinked in line 274. Note that the `taken` flag does not have to be set atomically since an element is only removed by a single thread which executes the dequeue operation with the matching timestamp.

Correctness

The CTS queue is linearizable with respect to Queue. The proof is based on the following invariants: it is guaranteed by the use of two instances of TS-atomic that for each dequeue operation there exists exactly one element with the same timestamp, as long as there are more elements than dequeue operations. Since TS-atomic is deterministic and monotonic it is guaranteed that for a dequeue operation with timestamp t and an element with timestamp $t' < t$ there exists a dequeue operation with timestamp t' which will remove the element with timestamp t'. Therefore, according to cooperation, the dequeue operation with timestamp t can remove the element with timestamp t correctly, and there will never be a second dequeue operations trying to remove the same element.

In the following we prove the linearizability of that the CTS queue.

Theorem 15. *The CTS queue is linearizable with respect to* Queue.

Proof. We use the queue theorem (Theorem 13) to sketch the linearizability proof of the CTS queue. We have to show that any history $\mathcal{H} = \langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ arising from the CTS queue is queue order-correct and linearizable with respect to Set.

As the optimization only affects the order in which elements are removed from the TS queue, and the TS queue is linearizable with respect to Set, also the CTS queue is linearizable with respect to Set.

It remains to show that the CTS queue is order-correct. Assume $+a, +b, -b \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{pr}} +b$ and $+b \xrightarrow{\mathsf{val}} -b$. We have to show that there exists a $-a \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ and $-b \not\xrightarrow{\mathsf{pr}} -a$.

Assume $+a$ enqueued element a with timestamp t_a, and $+b$ enqueued element b with timestamp t_b. Because of $+a \xrightarrow{\mathsf{pr}} +b$ it is guaranteed by the timestamping algorithm that $t_a < t_b$. As $+b \xrightarrow{\mathsf{val}} -b$, the timestamp of $-b$ is t_b. We know that TS-atomic is deterministic, and we know that $t_a < t_b$. Therefore there has to exist some dequeue operation with timestamp t_a which acquired its timestamp before $-b$ was timestamped with t_b. Let the dequeue operation with timestamp t_a be $-a$. With timestamp t_a the dequeue operation $-a$ will remove element a and therefore $+a \xrightarrow{\mathsf{val}} -a$. Since $-a$ acquired its timestamp before $-b$ it also holds that $-b \not\xrightarrow{\mathsf{pr}} -a$, which completes the proof.

□

Drawbacks

The CTS queue has the following drawbacks:

- No lock-freedom: if no element with timestamp t is inserted into the CTS queue, then the dequeue operation with timestamp t cannot finish.
- The emptiness check becomes more difficult: if a dequeue operation with timestamp t returns `EMPTY` instead of the element with timestamp t, it has to make sure that no element with timestamp t is enqueued. Otherwise there would not exist a dequeue operation to remove it. In our implementation of the CTS queue a dequeue operation never returns `EMPTY` but searches through the SP queue pools until it finds its matching element.

6.3.2 Initial Work: Relaxed TS Queue (RTS Queue)

The second TS queue variant is similar to the CTS queue in the way it is implemented, but quite different in its effect. It uses an optimization which, in general, does not guarantee queue semantics, although its behavior remains close to queue semantics. Therefore we call this variant the Relaxed TS queue (RTS queue).

In the RTS queue, concurrent dequeue operations do compete for elements, different to the CTS queue which avoids competition by using cooperation to the full extend. However, given k concurrent dequeues, these dequeues do not compete just

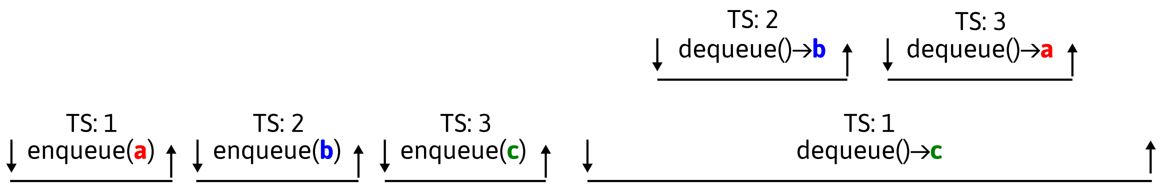


Figure 6.3: A history possible with the RTS queue which is not linearizable with respect to Queue.

for the oldest element but for the k oldest elements in the queue at the same time. The intuition is that since the k dequeue operations run concurrently, the order in which they remove the k oldest elements does not matter.

Correctness

The RTS queue optimization has the following problem. Correctness would require that if k dequeue operations compete for the k oldest elements, then each of these k oldest elements is removed by one of the k dequeue operations. However, in general it is possible that before these k dequeue operations remove their elements, new dequeue operations are invoked. These new dequeue operations then also compete for the elements which have not been removed yet. If one of the new dequeues removes one of the original k elements, then the correctness requirement stated above is violated because one of the k oldest elements has not been removed by one of the original k dequeue operations. As we show in the example in Figure 6.3, queue semantics may be violated if this correctness requirement is violated.

In Figure 6.3, first the elements a, b, and c are enqueued in sequence. Then two dequeue operations, labeled with `TS:1` and `TS:2` are executed concurrently and compete for the elements a and b. One of the dequeue operations removes b, the other dequeue operation is delayed. After b has been removed, a third dequeue operation is invoked and thereby there are again two concurrent dequeue operations, the delayed dequeue operation and the third dequeue operation. According to the optimization, these dequeue operations again compete for the two oldest elements, now a and c. It is therefore possible that the third dequeue operation removes a. Thereby a is enqueued before but dequeued after b, a clear violation of queue semantics.

Note that in an execution where dequeue operations can be partitioned such that within a partition all dequeue operations run concurrently, and dequeues across partitions are ordered by precedence, the scenario described above is not possible. With such a partition it is guaranteed that all k concurrent dequeue operations (the dequeue operations in one partition) finish before new dequeue operations (dequeue operations in the next partition) are invoked, and these k dequeue operations remove the k oldest elements. In executions where this is guaranteed, the RTS queue is linearizable with respect to QUEUE.

Additionally we believe that the RTS queue is quiescently consistent with respect to QUEUE. Quiescent consistency [AHS94, HS08] is a consistency condition which requires that a data structure converges to a consistent state whenever the execution reaches a quiescent state, i.e. a state where no operation accessing the data structure is executing.

The counter example described above requires that new dequeue operations are invoked before old dequeues finish. Therefore there cannot be a quiescent state between the dequeue operations. Whenever a quiescent state is reached, all dequeue operations have finished and these dequeue operations have removed the oldest elements, which establishes a consistent state. Therefore we believe that the RTS queue is quiescently consistent.

Implementation

Listing 6.6 shows the pseudo code of the RTS queue. Most of the code is the same as for the TS queue, the differences are highlighted in gray.

We implement the RTS queue by using again one instance of TS-atomic timestamping to timestamp elements (line 297), and a second instance of TS-atomic timestamping to timestamp dequeue operations (line 298). Different to the CTS queue, however, a dequeue operation with timestamp t does not necessarily remove the element with timestamp t but any element with a timestamp $t' \leq t$ (line 328). If a dequeue operation does not find any such element, then the dequeue operation tries to remove the element with the lowest timestamp it found (line 338). Elements are timestamped before they are inserted into the SP queue pool to avoid dealing with un-timestamped elements.

Note that the RTS queue in Listing 6.6 does not contain an emptiness check, although in principle we could use the emptiness check of the TS queue. The problem of the emptiness check is not the detection of an empty state but the violation of invariants caused by dequeue-empty operations, which we will describe next.

Removing elements from the RTS queue can be seen as a timestamping algorithm. The current time is the lowest timestamp of any element in the RTS queue, and time

Listing 6.6: The RTS queue algorithm. The SP queue pool is defined in Listing 6.3 and described in Section 6.1.1, TS-atomic is discussed in Section 2.2.

```
RTSQueue{
 Node{
   Element element,
   Timestamp timestamp,
   Node next
 };

 SPQueuePool[maxThreads] spPools;
 TSAtomic elementTimestamping;
 TSAtomic dequeueTimestamping;

 void enqueue(Element element){
   SPQueuePool pool=spPools[threadID];
   Node node=pool.insert(element);
   Timestamp timestamp=elementTimestamping.newTimestamp();
   node.timestamp=timestamp;
 }
 Element dequeue(){
   Bool success;
   Element element;
   Timestamp dequeueTS=dequeueTimestamping.newTimestamp();
   do{
     <success,element>=tryRem(dequeueTS);
   } while (!success);
   return element;
 }

 <Bool,Element> tryRem(Timestamp dequeueTS){
   Node oldest=NULL;
   Timestamp timestamp=MAX;
   SPQueuePool pool;
   for each(SPQueuePool current in spPools){
     Node node;
     Node poolInsert;
     <node,poolInsert>=current.getOldest();
     if (node == NULL)
       continue;
     Timestamp nodeTimestamp=node.timestamp;
     // RTS optimization.
     if(dequeueTS >=_TS nodeTimestamp)
       return current.remove(poolInsert,node);
     if(timestamp >_TS nodeTimestamp){
       oldest=node;
       timestamp=nodeTimestamp;
       pool=current;
     }
   }
   if(youngest==NULL)
     return <false,NULL>;
   return pool.remove(oldest);
 }
}
```

advances when the element with the lowest timestamp is removed. In the following we call this timestamping algorithm *TS-queue-state*.

In normal dequeue operations both the TS-atomic instance for dequeue operations and TS-queue-state make progress, TS-atomic in line 309, and TS-queue-state in line 329 or in line 338. Therefore these timestamping algorithms are synchronized, and the current time in both timestamping algorithms is the same whenever an execution reaches a quiescent state.

This synchronization is used to satisfy the requirements of the RTS queue optimization. The RTS queue optimization requires that k concurrent dequeue operations compete for the k oldest elements in the queue. With the synchronization of TS-atomic and TS-queue-state this is guaranteed by the invariant that if t is the latest timestamp of the k concurrent dequeue operations, then there exist k elements in the queue with a timestamp $t' \leq t$.

In a naive implementation, dequeue-empty operations would destroy the synchronization between TS-atomic and TS-queue-state because dequeue-empty operations do not remove elements and therefore TS-queue-state does not make progress. Therefore a proper implementation of dequeue-empty operations would not only require the detection of an empty state but also a re-synchronization of TS-atomic and TS-queue-state. This, however, remains future work.

6.4 Performance of the TS Queue

In this section we analyze the performance and scalability of the TS queue. We run the TS queue in the producer-consumer benchmark described in Section 3.2 with an increasing number of threads and compare its performance with the LCRQ [AM13], the fastest concurrent queue algorithm we are aware of, and the Michael-Scott (MS) queue [MS96], a standard baseline used in most publications of concurrent queue algorithm papers.

The MS queue is based on a linked list of nodes which is access by a head and a tail pointer (similar to the SP queue pool). To enqueue and dequeue elements, these access pointers are modified atomically using the `CAS` instruction.

The LCRQ stores elements in a linked list of array segments. The linked list is accessed by a head and a tail pointer, similar to the Michael-Scott queue. Elements are inserted into the tail segment, and removed from the head segment. If the tail segment gets full, a new array segment is attached at the tail of the linked list. Similarly, if the

	40-core machine	64-core machine
high-contention:		
TS-interval queue	28.5 µs	30 µs
TS-CAS queue	30 µs	30 µs
MS queue	3000 µs	3000 µs
low-contention:		
TS-interval queue	18 µs	24 µs
TS-CAS queue	30 µs	30 µs
MS queue	1000 µs	1500 µs

Table 6.1: Benchmark delay times for TS-interval queue / TS-CAS queue / MS queue.

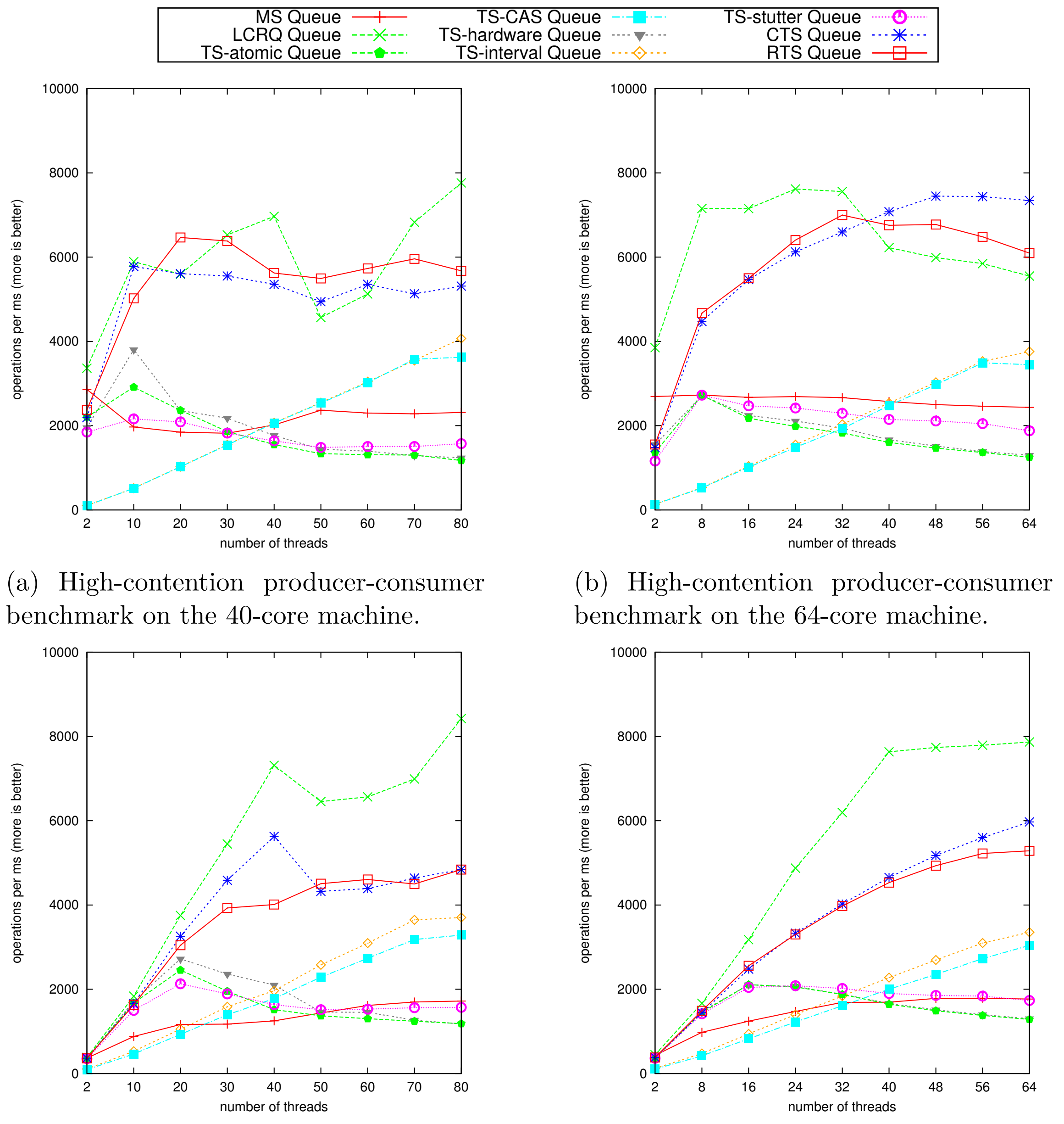
head segment gets empty, it is removed from the head of the linked list. Consecutive, monotonically increasing indices are attached to elements in each array segment. These indices are created with an atomic fetch-and-increment instruction. The index of an element determines the slot in the array segment where the element is stored. Dequeue operations acquire a similar indices and try to remove the element with the same index from the tail segment of the linked list.

Similar to the Treiber stack also the MS queue benefits from a backoff which delays the retry of a failed `CAS` [DHM13]. Therefore we configured the MS queue with a constant backoff. We use the backoffs shown in Table 6.1. These backoffs turned out to be optimal in the producer-consumer benchmarks with 80 threads and 64 threads on the 40-core machine and the 64-core machine, respectively. The LCRQ is configured with the default ring size of 2^{17}, as suggested in [AM13].

The configurations of the TS-interval queue and the TS-CAS queue are shown in Table 6.1. Similar to the backoff of the MS queue these configurations are optimal in the producer-consumer benchmark when run with 80 threads on the 40-core machine and with 64 threads on the 64-core machine. More details about the delay parameter of the TS queue are shown in Section 6.4.1.

Figure 6.4 shows the benchmarks results of the queue measurements. In the low contention experiments the LCRQ is faster than the TS queues and the MS queue, in the high contention experiments the CTS queue and the RTS queue can be faster. The other TS queue configurations are always slower than the LCRQ.

With an increasing number of threads both the TS-interval queue and the TS-CAS queue are faster than the MS queue. With lower numbers of threads these TS queues perform poorly. The reason is that TS queue configurations use a delay in their



(a) High-contention producer-consumer benchmark on the 40-core machine.

(b) High-contention producer-consumer benchmark on the 64-core machine.

(c) Low contention producer-consumer benchmark on the 40-core machine.

(d) Low contention producer-consumer benchmark on the 64-core machine.

Figure 6.4: TS queue performance on the 40-core machine (left) and on the 64-core machine (right).

timestamping algorithms which is optimal for high numbers of threads, but too long for low numbers of threads. For example, the performance of the TS-interval queue without any delay is comparable to the performance of the TS-hardware queue.

The TS-atomic queue, the TS-hardware queue, and the TS-stutter queue scale negatively with an increasing number of thread. The reason is that with these time-

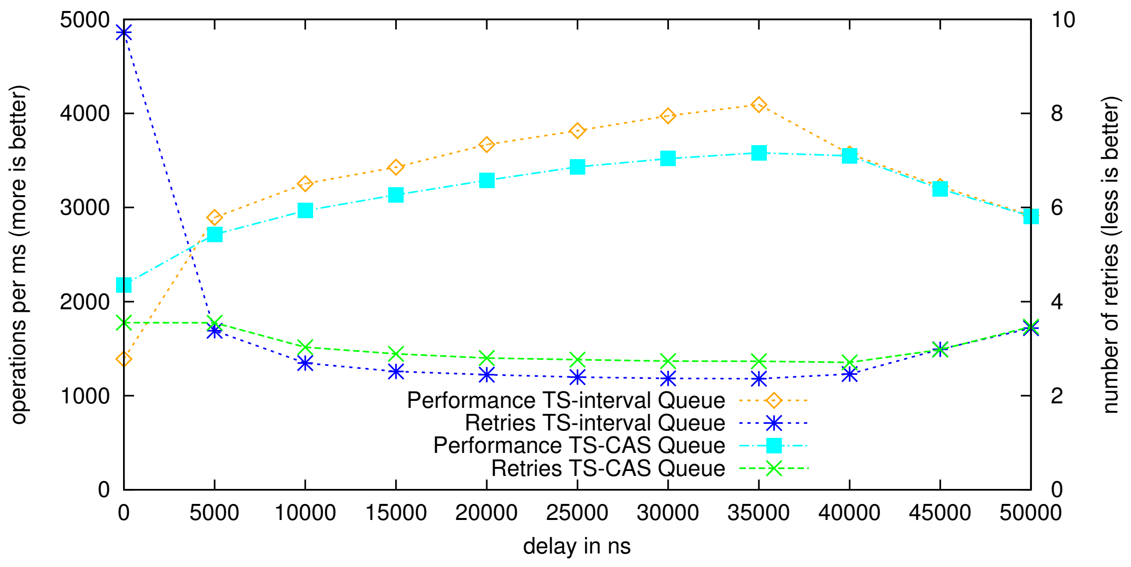


Figure 6.5: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 40-core machine, exercising 40 producers and 40 consumers.

stamping algorithms nearly all elements are ordered by their timestamps, and therefore there is contention on the element with the oldest timestamp. With high numbers of threads the TS queues can are then slower than the MS queue.

The CTS queue and the RTS queue perform better than the regular TS queues. Interestingly, depending on the machine, either the RTS queue is faster than the CTS queue, or vice versa. The reason may lie in the costs on each machine for searching and removing an element. Conceptually, searching for an element can be cheaper for the RTS queue than for the CTS queue because for a dequeue operation of the RTS queue there can exist multiple candidate elements. However, removing an element is more expensive for the RTS queue than for the CTS queue because in the RTS queue multiple dequeue operations compete for the same elements. Therefore, depending on whether searching or removing elements is more expensive, either the CTS or the RTS queue is faster.

Note however, that the CTS queue and the RTS queue do not have the same interface as the other queue implementations since they do not offer an emptiness check. An emptiness check may or may not effect the performance of these queues.

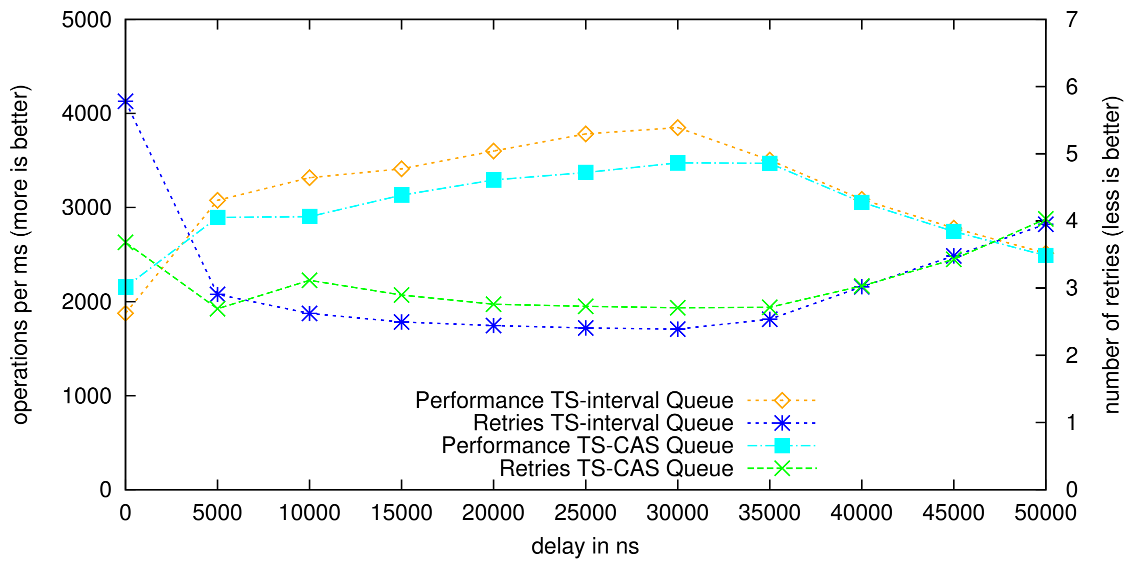


Figure 6.6: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 64-core machine, exercising 32 producers and 32 consumers.

6.4.1 Analysis of the Interval Timestamping for the TS Queue

Figure 6.5 and Figure 6.6 show the performance of the TS-interval queue and the TS-CAS queue with an increasing interval length in the high contention producer-consumer benchmark with 80 threads on the 40-core machine and with 64 threads on the 64-core machine. Similar to the TS stack the performance initially increases with an increasing interval length. However, eventually enqueue operations become slower than dequeue operations and performance decreases again. The performance correlates with the number of retries of `remove` operations in the dequeue.

Chapter 7

Initial Work: TS Deque

In this section we present a timestamped data structure which is a potential candidate for a linearizable deque. A deque theorem to prove linearizability of the so-called TS deque with respect to deque semantics is still work in progress. It is easy to show, however, that the TS deque presented in this section satisfies the stack theorem when used as a stack, and the queue theorem when used as a queue. In the following we define the sequential specification of a deque, then we will describe the TS deque algorithm with its internal data structure, the SP deque pool.

A deque allows to insert and remove elements on both sides of its state. Its sequential specification DEQUE is defined as follows: assume a set of values Val. Abstract states are finite sequences in Val^*. Let $\sigma \in \mathsf{Val}^*$ be an arbitrary state. In DEQUE, insertL, insertR, removeL, and removeR have the following sequential behavior ('$\cdot$' means sequence concatenation):

- `insertR(v)` – Update the abstract state to $\sigma \cdot [\mathtt{v}]$.
- `insertL(v)` – Update the abstract state to $[\mathtt{v}] \cdot \sigma$.
- `removeR()` – If $\sigma = []$, return EMPTY. Otherwise, σ must be of the form $\sigma' \cdot [v']$. Update the state to σ', return v'.
- `removeL()` – If $\sigma = []$, return EMPTY. Otherwise, σ must be of the form $[v'] \cdot \sigma'$. Update the state to σ', return v'.

Informally, these specifications mean that removeR returns the youngest element inserted at the right (with insertR) if there exists one, and otherwise it removes the oldest element inserted at the left (with insertL). The same holds analogously for removeL.

If only insertR and removeR are used, then a deque behaves like a stack, and if only the insertL and removeR methods are used, then the deque behaves like a queue. Thus in the TS deque algorithm we will reuse concepts from the TS stack and the TS queue algorithms. In this section we will concentrate mostly on the differences between the TS deque and the TS stack and TS queue algorithm.

Listing 7.1 and Listing 7.2 illustrate the TS deque algorithm. Note the similarity to the TS stack and TS queue algorithm. In the insertR method elements are first inserted into a SP deque pool and then timestamped. The removeR method searches for a right-most element and then tries to remove the identified element. Searching is done by iterating over all SP deque pools and comparing the timestamps of the right-most element of each pool.

Since the side where an element got inserted into a deque matters for the order in which elements get removed, we store this information in the `side` field of the node in the SP deque pool. Additionally a node has an `index` field which is used to identify the order in which elements were inserted into a SP dequeue pool.

In the `tryRemR` method elements inserted at the right can be eliminated (line 410) using the elimination optimization [HSY04]. Elements inserted at the left, however, cannot be eliminated. Similar to the TS queue (Section 6.1) we even have to check that an element inserted at the left side has been inserted before `tryRemR` started searching for the right-most element.

For all the operations shown in Listing 7.1 and Listing 7.2 which access the right side of the deque there exist analogous operations which access the left side of the deque. This means that there do not only exist insertR, removeR, isMoreRight, and tryRemR, but also insertL, removeL, isMoreLeft, and tryRemL.

7.1 SP Deque Pool

The implementation of the SP deque pool is similar to the implementation of the concurrent linked deque in the OpenJDK [ope]. A SP deque pool consists of a doubly-linked list of nodes which is accessed by a `left` pointer and a `right` pointer. Both pointers are annotated with ABA-counters to avoid the ABA-problem [HS08]. Each node contains a `left` pointer, a `right` pointer, a `side` flag, an `element` field, a `taken` flag, and an `index` field. The `left` and `right` pointers point to the left neighbor and

Listing 7.1: The TS deque algorithm. Part 2 is presented in Listing 6.2. The SP deque pool is defined in Listing 7.3 and described in Section 7.1, timestamps are discussed in Section 2.2.

```
enum Side {Left, Right};

TSDeque{
 Node{
   Element element,
   Timestamp timestamp,
   Side side,
   Node left,
   Node right,
   Int index,
   Bool taken
 };

 SPDequePool[maxThreads] spPools;

 void insertR(Element element){
   SPDequePool pool=spPools[threadID];
   Node node=pool.insertR(element);
   Timestamp timestamp=newTimestamp();
   node.timestamp=timestamp;
 }

 void insertL(Element element); // Analogous to insertR.

 Element removeR(){
   Bool success;
   Element element;
   do{
     Timestamp startTime=newTimestamp();
     <success,element>=tryRemR(startTime);
   } while (!success);
   return element;
 }

 void removeL(Element element); // Analogous to removeR.

 Bool isMoreRight(Timestamp timestamp1, Side side1,
                  Timestamp timestamp2, Side side2) {
   if (side1 == side2) {
     if (side1 == Right) // && side2 == Right
       return (timestamp1 >TS timestamp2);
     else // if side1 == Left && side2 == Left
       return (item1.timestamp <TS item2.timestamp);
   } else // if side1 != side2
     return (side1 == Right);
 }

 Bool isMoreLeft(Timestamp timestamp1, Side side1,
     Timestamp timestamp2, Side side2); // Analogous to isMoreRight.
```

Listing 7.2: Part 2 of the TS deque algorithm. The SP deque pool is defined in Listing 7.3 and described in Section 7.1, timestamps are discussed in Section 2.2.

```
<Bool,Element> tryRemR(Timestamp startTime){
  Node rightMost=NULL;
  Timestamp timestamp=MAX;
  Bood side=Left;
  SPDequePool pool;
  Node[maxThreads] emptyR;
  Node[maxThreads] emptyL;
  for each(SPDequePool current in spPools){
    Node node;
    Node poolLeft;
    Node poolRight;
    poolLeft=current.left;
    <node,poolInsert>=current.getRightmost();
    // Emptiness check
    if(node==NULL){
      emptyR[current.ID]=poolRight;
      emptyL[current.ID]=poolLeft;
      continue;
    }
    Timestamp nodeTimestamp=node.timestamp;
    if(node.side==Right && nodeTimestamp >TS startTime)
      return current.removeR(poolRight,node);
    if(isMoreRight(nodeTimestamp, node.side, timestamp, side)){
      rightMost=node;
      timestamp=nodeTimestamp;
      side=node.side;
      pool=current;
    }
  }
  // Emptiness check
  if(rightMost==NULL){
    for each(SPDequePool current in spPools){
      if(current.left!=emptyL[current.ID])
        return <false,NULL>;
      if(current.right!=emptyR[current.ID])
        return <false,NULL>;
    }
    return <true,EMPTY>;
  }
  if(side==Left && timestamp >TS startTime){
    return <false,NULL>;
  return pool.removeR(oldest);
}

<Bool,Element> tryRemL(Timestamp startTime); // Analogous to tryRemR.
}
```

Listing 7.3: SP deque pool algorithm (part 1).

```
SPDequePool {
 Node left;
 Node right;
 Int ID; // The ID of the owner thread.
 Int nextIndex = 1;

 void init() {
  Node sentinel = createNode(element=0, index=0, taken=true);
  sentinel.left = sentinel.right = sentinel;
  left = right = sentinel;
 }

 void insertR(Element element) {
  Node newNode = createNode
    (element=item, index=nextIndex, taken=false, side=Right);
  newNode->right = newNode;
  nextIndex = nextIndex + 1;
  Node rightMost = right;
  while (rightMost->left != rightMost && rightMost->taken) {
   rightMost = rightMost->left;
  }
  // The SP deque pool is empty, we have to guarantee that both the
  // left and right pointer point to the same list.
  if (rightMost = rightMost->left) {
    Node oldLeft = left;
    left = rightMost;
  }
  newNode->left = rightMost;
  rightMost->right = newNode;
  right = newNode;
 }

 void insertL(TimestampedItem item) {
  // Analogous to insertR, but the index of a new
  // node is initialized with 'index=-nextIndex'.
}
```

the right neighbor of the node, respectively, the `element` field stores the element, the `taken` flag indicates if the element of the node has been removed logically from the SP deque pool, and the `index` field is used to order and identify nodes. The `side` flag stores the insertion side. The `side` flag is not important for the internals of the SP deque pool but for the implementation of elimination in the TS deque.

The doubly-linked list is closed at its ends by nodes which have themselves as their left or right neighbor. The doubly-linked list is initialized with a sentinel node. Initially both `left` and `right` pointer of the SP deque pool as well as `left` and `right` pointer of the sentinel node itself point to the sentinel node. The `taken` flag of the sentinel is set to indicate that the node does not contain an element.

An element is contained in the SP deque pool if (1) there exists a node in the doubly-linked list that contains the element in its `element` field, if (2) the `taken` flag of that node is not set, if (3) the node is reachable from the `left` pointer of the SP deque pool following only `right` pointers, and if (4) the node is reachable from the `right` pointer of the SP deque pool following only `left` pointers. If one of the four conditions does not hold, the element is not considered as contained in the SP deque pool.

Similar to the SP stack pool we preserve the following invariants:

1. If a node is reachable from the `left` or `right` pointer once, then it remains reachable at least as long as its `taken` flag is not set.

2. If a node a is reachable from a node b in the linked list once, then a will always be reachable from b as long as both nodes have their `taken` flag not set.

3. The node with the highest index contains the right-most element in the SP deque pool, the node with the lowest index contains the left-most element.

4. The nodes in the list are sorted by their index. The left-successor of any node in the list has a lower index than the node itself, and the right-successor has a higher index. By using this order on the list it is guaranteed that the left-most element in the SP deque pool is contained in the left-most node which is not marked as `taken`, and likewise the right-most element is contained in the right-most node.

Listing 7.3 and 7.4 shows the pseudocode of the SP deque pool. To insert an element at the right side of the SP deque pool with an `insertR` operation, first a new

Listing 7.4: SP deque pool algorithm (part 2).

```
 <Node, Node> getRightmost() {
   Node oldLeft = left;
   Node oldRight = right;
   Node result = oldRight;
   while (true) {
     if(result->index < oldLeft->index)
       return <NULL, oldRight>;
     if(!result->taken)
       return <result, oldRight>;
     if(result->left == result)
       return <NULL, oldRight>;
     result = result->left;
   }
 }

 <Node, Node> getLeftmost() {
  // Analogous to getRightmost.
 }
 <Bool,Element> removeR (Node oldRight, Node node) {
  if (CAS(node->taken, false, true)) {
   CAS (right, <oldRight, aba>, <node, aba>);
   return <true,node.element>;
  }
  return <false,NULL>;
 }
 <Bool,Element> removeL (Node oldRight, Node node) {
  // Analogous to removeR.
 }
}
```

node is created with the next free index assigned to it and the element stored in its `element` field. Initially the `taken` flag is not set, and the `right` pointer is set to point to the new node itself. The `insertR` operation then tries to find the right-most node that has not been marked as `taken` (line 453-456). In line 463-465 the new node is inserted at the right of that right-most node.

Only after the new node is reachable from the `right` pointer of the SP deque pool the element is considered to be contained in the SP deque pool. If the SP deque pool is empty, then the iteration in line 453-456 ends at the left-most node of the SP deque pool. To consistently insert the new node, this left-most node becomes a new sentinel node (line 459-461) and the `left` pointer of the SP deque pool is changed to point to this sentinel node. The new node is then inserted to the right of the sentinel node. By using the sentinel node we guarantee that both the `right` pointer and the `left` pointer of the SP deque pool always point to the same doubly-linked list.

All invariants are preserved by `insertR`: Invariant 1 is preserved because the new element is inserted to the right of all not-taken nodes in the list. Invariant 2 is preserved for the same reason: any pointers changed in `insertR` only point to nodes which are already marked as `taken`, or are self references marking the end of the linked list. Invariant 3 is preserved because a new element inserted with `insertR` is always more right than any element already in the SP deque pool. This is because there is only a single thread inserting elements into the SP deque pool and thus we do not have to consider concurrent insertion. Invariant 4 is preserved because the index of a newly-inserted node is greater than the index of any node already in the SP deque pool.

The `getRightmost` operation iterates over the linked list starting at the `right` pointer and following `left` pointers until it finds a node with the `taken` flag unset. If the iteration reaches the end of the linked list, or if it gets to a node with an index lower than the index of the left-most node at the beginning of the iteration, we can stop the iteration because we know from Invariant 4 that any node further left can only contain elements inserted after the invocation of `getRightmost`. Because of the condition in line 429 in the TS deque any element inserted at the left after the invocation of `getRightmost` would be ignored.

The `removeR` operation tries to set the `taken` flag with a `CAS` and returns `true` and the element if it succeeds. Otherwise the `removeR` operation returns `false` and `NULL`. After succeeding in the first `CAS` the operation additionally tries to adjust the `right` pointer of the SP deque pool with a `CAS` to unlink nodes with a set `taken` flag.

Unlinking `taken` nodes could only violate Invariant 1 because it only changes the `right` pointer of the SP deque pool, no internal pointers of the linked list. We prove now that Invariant 1 is not violated. The old value of the `right` pointer for the `CAS` was read before `getRSP` started the iteration through the linked list. In this iteration the node removed by this `tryRemSP` was the first node with the `taken` flag not set. If a new element were inserted to the right of the old node, then the `right` pointer would have changed in the meantime and the `CAS` would fail.

Note that the unlinking of `taken` nodes is not complete, which means that with a bad interleaving the number of `taken` nodes in the SP deque pool is unbounded. Bounding the number of `taken` nodes in the SP deque pool remains future work.

The `insertL`, `getLeftmost`, and `removeL` operations work analogous to their counterparts `insertR`, `getRightmost`, and `removeL`. Except for swapping ‘right’ and ‘left’, the only difference is that negative indices are assigned when elements are inserted by an `insertL` operation to keep the doubly-linked list sorted, and that the comparison of node indices is changed to account for the negative indices (i.e., '<' is swapped with '>' and '>' is swapped with '<').

7.2 Performance of the TS Deque

In this section we evaluate the performance and scalability of the TS deque algorithm. As a baseline we use a flat-combining (FC) deque [HIST10]. A FC deque is a single-threaded implementation of a deque protected by a single lock. A thread which acquires the lock successfully executes its own operation on the deque and additionally the operations of all threads which tried to acquire the lock but did not get it.

The FC deque is not lock-free, and there exist lock-free concurrent deque implementations. These implementations are based on doubly-linked lists where pointers are modified by `CAS` instructions, e.g. [Mic03] and [ST08]. In our experiments we compare the TS deque with the FC deque because it is easier to implement and because flat-combining tends to out-perform `CAS`-based lock-free implementations [HIST10].

We evaluate the TS deque with the same producer-consumer benchmark as the TS stack and the TS queue. Since the TS deque provides two insert operations, a producer randomly chooses for each element to insert it either at the left side or the right side of the deque. Analogously a consumer chooses randomly to remove an element either at the left or the right side of the deque.

Figure 7.1 shows the performance and scalability results of the TS deque. With an increasing number of threads all TS deque configurations outperform the FC deque. One reason is that the TS deque benefits from elimination [HSY04], which is not provided by the FC deque.

	40-core machine	64-core machine
high-contention:		
TS-interval deque	13.5 µs	12 µs
TS-CAS deque	13.5 µs	10.5 µs
low-contention:		
TS-interval deque	10.5 µs	10.5 µs
TS-CAS deque	9 µs	9 µs

Table 7.1: Benchmark delay times for TS-interval deque/ TS-CAS deque.

Similar to the TS stack and the TS queue the TS-interval and TS-CAS variants of the TS deque are faster than the TS deques with the other timestamping algorithms with an increasing number of threads. With lower numbers of threads these TS deques are slower than the other TS deque variants because they are configured to achieve optimal performance with high numbers of threads, i.e. 80 threads on the 40-core machine and 64 threads on the 64-core machine. For lower numbers of threads these configurations are suboptimal.

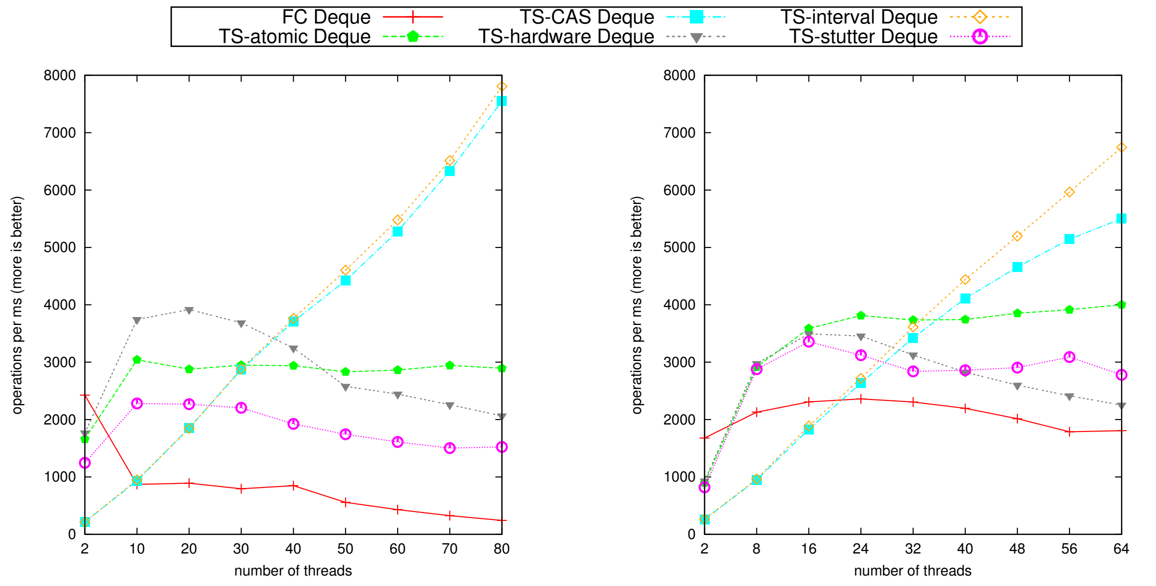
The configuration parameters of the TS-interval deque and the TS-CAS deque are shown in Table 7.1. As mentioned above these parameters are optimal for the producer-consumer benchmark when exercised with 80 threads on the 40-core machine and with 64 threads on the 64-core machine.

7.2.1 Analysis of the Interval Timestamping for the TS Deque

Figure 7.2 and Figure 7.3 show the performance of the TS-interval deque and the TS-CAS deque in the high contention producer-consumer benchmark with a varying delay parameter.

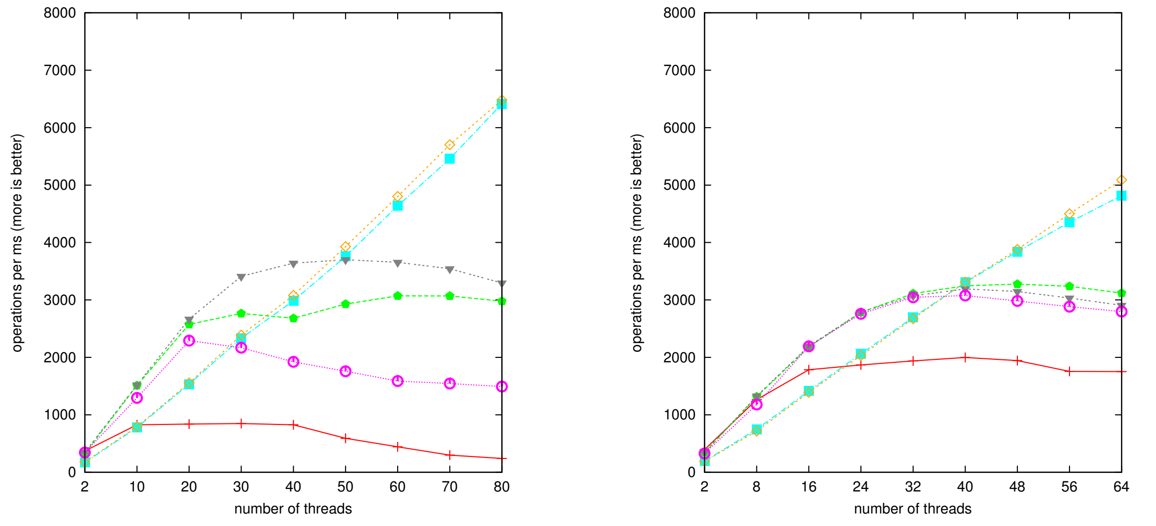
Similar to the TS stack the performance of the TS deque first improves with an increasing delay. The reason is that with an increasing delay more elements have overlapping timestamps and therefore more parallel removal is possible and as a consequence the number of retries in the remove operations decreases. Additionally more insert operations overlap with remove operations and therefore also more elimination is possible. However, eventually the delay gets too long and insert operations become slower than remove operations, and the overall performance decreases again. Note that the number of retries does not increase significantly when the delay get longer since many remove operations return EMPTY instead of retrying.

Note that on the 40-core machine the performance of the TS-CAS deque first decreases with an increasing delay, then it increases again. Additional measurements suggest that the TS-CAS timestamping algorithm is slower without a delay than with a small delay because of the contention on the atomic counter within TS-CAS. With this contention many `CAS` attempts on the atomic counter fail, leading to more overlapping timestamp intervals than with a short delay. Additionally, with the slower timestamping more elimination is possible. With an increasing delay the number of overlapping timestamps increases again, and therefore also the overall performance.



(a) High-contention producer-consumer benchmark on the 40-core machine.

(b) High-contention producer-consumer benchmark on the 64-core machine.



(c) Low contention producer-consumer benchmark on the 40-core machine.

(d) Low contention producer-consumer benchmark on the 64-core machine.

Figure 7.1: TS deque performance on the 40-core machine (left) and on the 64-core machine (right).

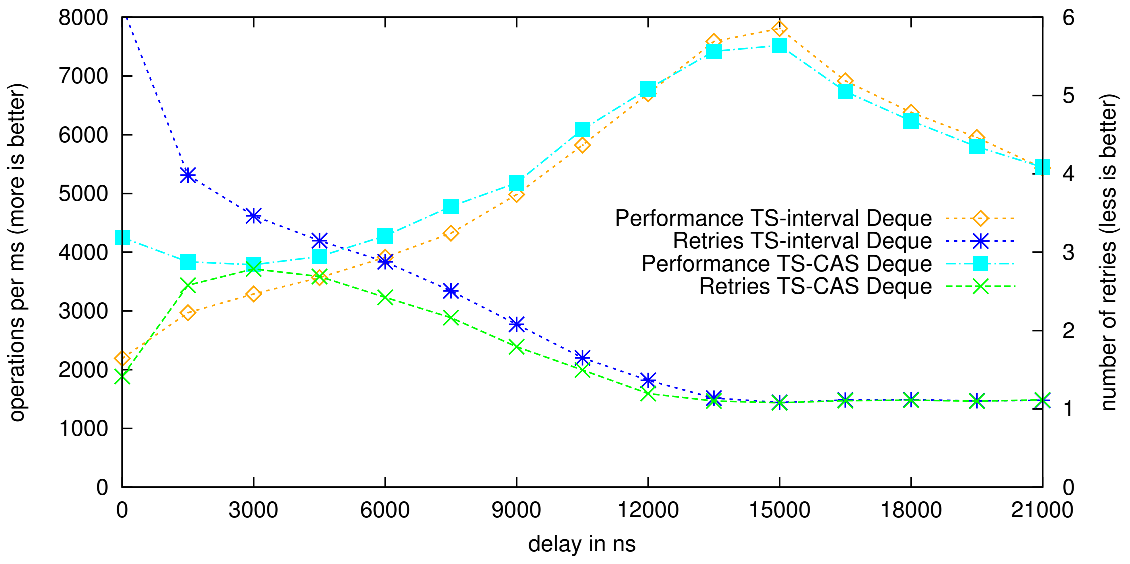


Figure 7.2: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 40-core machine, exercising 40 producers and 40 consumers.

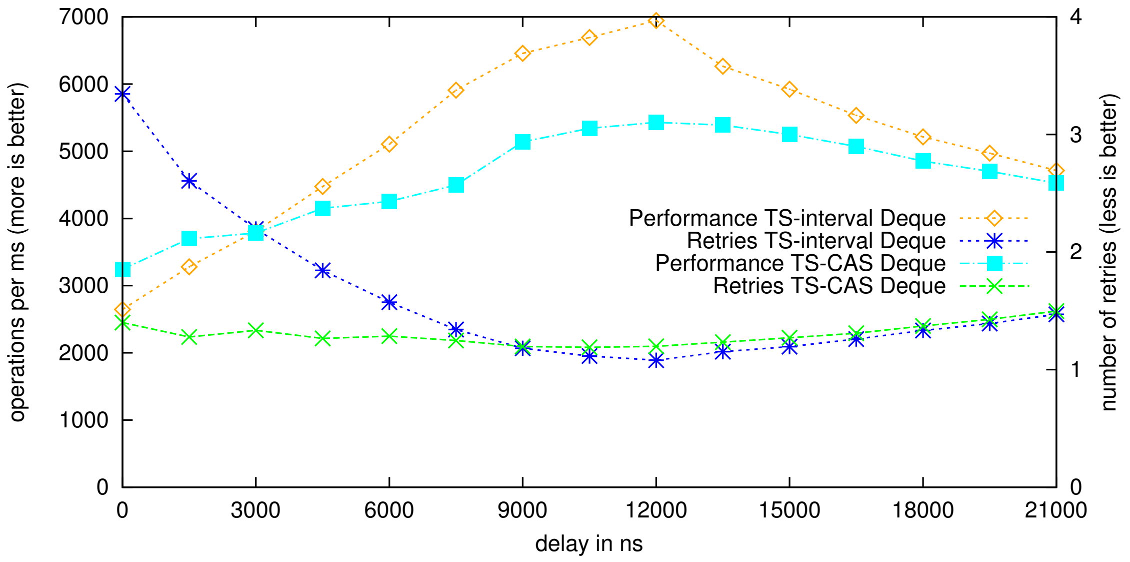


Figure 7.3: High-contention producer-consumer benchmark using TS-interval and TS-CAS timestamping with increasing delay on the 64-core machine, exercising 32 producers and 32 consumers.

Chapter 8

Order Deviation of Concurrent Queues

Linearizability [HW90] does not require operations to take effect in the order they were invoked. It only requires that operations take effect at some point in time between their invocation and response. Therefore operations may take effect in a different order than the order in which they were invoked.

We are interested in how much operations are reordered between their invocations and the time they take effect in various implementation. We analyze the amount of reordering with a new metric called order deviation [HKLP12]. Order deviation only allows to analyze queues. In this chapter we compare therefore the order deviation of the TS queue with the order deviation of other queues.

8.1 Order Deviation

To describe order deviation we start by considering a perfect linearizable queue implementation where all operations are executed in zero time. When operations execute in zero time, then all operations take effect immediately at their invocation, and no operations are reordered between their invocation and the time they take effect.

Now consider an execution using this perfect queue where all elements which get enqueued also get dequeued. Consider two enqueue-dequeue pairs $+a, -a, +b, -b$ with $+a \xrightarrow{\mathsf{val}} -a$ and $+b \xrightarrow{\mathsf{val}} -b$. If $+a$ is invoked before $+b$ and therefore also takes effect before $+b$, then according to queue semantics also $-a$ takes effect before $-b$ and therefore is invoked before $-b$. This means that if we order the elements in this execution by the invocation order of their enqueues, constructing a so-called enqueue order, and if we order elements by the invocation order of their dequeues, constructing a so-called dequeue order, then the enqueue order and the dequeue order would be the same.

In reality such a perfect queue is impossible, and the dequeue order may deviate from the enqueue order. We call this deviation the *order deviation*[1]. The main sources of order deviation are

1. that the time it takes different operations in an execution to take effect may vary due to hardware artifacts, scheduling artifacts, and failing of atomic instructions like a compare-and-set; and

2. a partial order on the elements in the queue which allows unordered elements to be dequeued in arbitrary order (e.g. elements which are unordered by their timestamps).

Many linearizable queue implementations enforce a total order on the elements in the queue and thereby eliminate the second source of order deviation. In the TS queue we preserve only a partial order on elements, which on the one hand introduces the second source of order deviation, on the other hand, however, improves performance and thereby reduces the first source of order deviation. In our analysis we are interested whether we can reduce order deviation through good performance more than we increase it with the partial order on elements due to timestamping. In our experiments we compare the order devation of the TS queeu with the order deviation of a lock-based (LB) queue, the Michael-Scott (MS) queue [MS96], the flat-combining (FC) queue [HIST10], the wait-free (WF) queue [KP11], and the LCRQ queue [AM13].

The LB queue is based on a singly-linked list where all accesses to the linked list are controlled by a global lock. The FC queue can be seen as an extension of the LB queue in the sense that it is also a singly-linked list protected by a global lock. However, a thread which acquires the lock does not only execute its own operation but also the operations of all threads which failed to acquire the lock. For this each threads writes its operation into a registration array before it tries to acquire the lock.

The MS queue is a singly-linked list accessed by a head and a tail pointer. Operations of the MS queue repeatedly try to modify the head or tail pointer with a compare-and-set instruction until they succeed. The WF queue is an extension of the MS queue to achieve wait-freedom. For this the WF queue uses a helping approach

[1]In [HKLP12] we called order deviation *element fairness.*

based on a registration array. With the registration array threads can detect whether other threads failed to execute their operation within a time limit and help these threads to finish their operations.

The LCRQ queue is a MS queue of array segments. Each array segment is a cyclic buffer accessed with two counters which are incremented by atomic fetch-and-increment instructions. Each of these cyclic buffers is a bounded size queue. If an array segment gets full, a new array segment is allocated and inserted into the MS queue. If an array segment becomes empty, then it is dequeued from the MS queue.

There exist other data structure implementations which approximate queue semantics [HKP+13] but are not linearizable with respect to Queue. These data structures also enforce only a partial order on elements and thereby gain performance and scalability. We call these data structures relaxed queues. In this chapter we also analyze the order deviation of relaxed queues. We measure the order deviation of the k-FIFO queue [KLP13], the 1-round-robin distributed queue (1RR DQ), the 2-round-robin distributed queue (2RR DQ), and the 1-random distributed queue (1RA DQ) [HHK+13].

The k-FIFO queue is similar to the LCRQ queue in the sense that it is based on a MS queue of array segments. However, elements in one segment are accessed in random order, which means that two elements in the same segment can be removed in a different order than the order in which they were inserted.

The DQ queue is an array of MS queues where for each operation a load balancer decides on which of the MS queues should be accessed by the operation. If a thread tries to dequeue from an empty MS queue, then it scans linearly over the other MS queues to dequeue an element from them. A dequeue operation returns empty if all MS queues are empty. In the 1RR DQ queue the load balancer consists of one round-robin counter for enqueue operations and one round-robin counter for dequeue operations. These round robin counters are incremented by an atomic fetch-and-increment instruction. In the 2RR DQ queue the load balancer consists of two round-robin counters per operation type, where half of the threads access only the first round-robin counter, the other half uses the second counter. In the 1RA DQ queue the load balancer randomly decides for one MS queue.

For relaxed queues the enqueue order may differ from the dequeue order even in a sequential execution due to their semantics. However, when relaxed queues provide good performance, operations may get reordered less between their invocation and the time they take effect. By measuring order deviation of relaxed queues we can compare the effect of performance with the effect of semantical deviation on the order in which elements are treated in concurrent queues.

8.2 Formal Definition of Order Deviation

In the following we define order deviation formally. We assume that $\mathcal{T}$ is a trace of an execution where all elements which get enqueued also get dequeued. Let $\langle \mathcal{A}, \mathsf{pr}, \mathsf{val} \rangle$ be the history extracted from $\mathcal{T}$, let $+a, -a \in \mathcal{A}$ with $+a \xrightarrow{\mathsf{val}} -a$ be the enqueue and dequeue operation of an element a, and let $+b, -b \in \mathcal{A}$ with $+b \xrightarrow{\mathsf{val}} -b$ be the enqueue and dequeue operation of an element b. For any operation $x \in \mathcal{A}$ let inv_x be the invocation of x in $\mathcal{T}$.

Definition 15 (Invocation Order). *We define the invocation order $<_{\mathsf{inv}}$ by $x <_{\mathsf{inv}} y$ if and only if $\mathsf{inv}_x <_{\mathcal{T}} \mathsf{inv}_y$.*

Definition 16 (Order Deviation). *Let $<_{\mathsf{inv}}$ be the invocation order of an execution. The order deviation of element a is defined as the number of elements b such that $+b <_{\mathsf{inv}} +a$ and $-a <_{\mathsf{inv}} -b$. Additionally the order deviation of a dequeue-empty operation $-e$ is the number of elements b such that $+b <_{\mathsf{inv}} -e$ and $-e <_{\mathsf{inv}} -b$.*

We say that the order deviation of an implementation in an execution is the average order deviation of all elements.

8.3 Experimental Setup

We determine the invocation order in an execution by recording the time of all operation invocations using the `RDTSCP` instruction. However, this is only an approximation of the actual invocation order because `RDTSCP` is not executed atomically within the invocation. Nevertheless, our experiments show that the approximation error is insignificant (less than 0.2 order deviation per element on average).

We measure order deviation in three benchmarks.

- We measure the order deviation of the whole data structure in a producer-consumer benchmark. Each producer enqueues 10000 elements, each consumer dequeues 10000 elements.

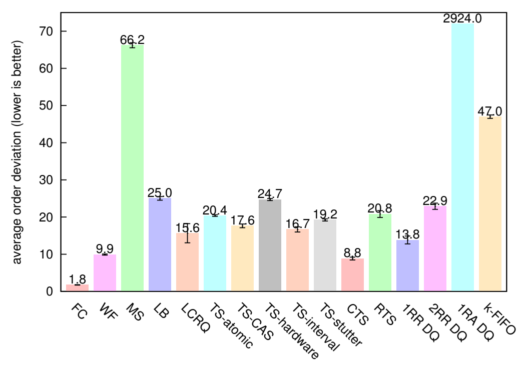
- We measure the order deviation introduced by enqueue operations by executing a multiple-producer single-consumer benchmark. The consumer starts after all producers enqueued 10000 elements each and dequeues all elements.
- We measure the order deviation introduced by dequeue operations by executing a single-producer multiple-consumer benchmark. The consumers start dequeueing 10000 elements each after the producer finished enqueueing 10000 elements for each consumer.

Note that in these benchmarks fewer operations access the data structure than in the performance analysis benchmarks. The reason is the time and space complexity of the order deviation analysis.

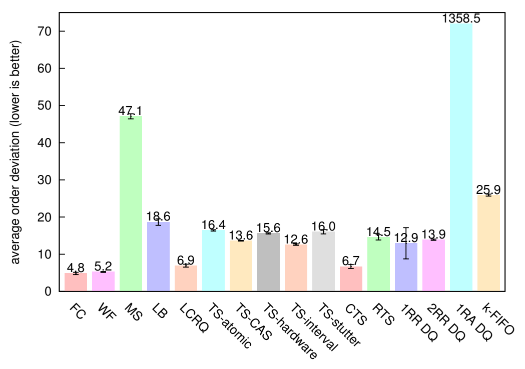
The producer-consumer benchmark is the same benchmark we use in our performance analysis in Section 6.4, except that we execute fewer operations. The other two benchmarks are used to show the order deviation of enqueue and dequeue operations separately. This is possible by executing only those operations concurrently for which we want to measure order deviation. All other operations have to execute sequentially. Operations which execute sequentially cannot introduce order deviation because, according to linearizability, sequentially executed operations have to take effect in the order they were executed. Therefore we can measure the order deviation of enqueue operations by executing dequeue operations in sequence, and we can measure the order deviation of dequeue operations by executing enqueue operations in sequence.

Note that this approach only works if e.g. dequeue operations behave the same no matter if enqueue operations execute sequentially or not. However, for the TS queue this is not true because if enqueue operations execute sequentially, then elements cannot share timestamps. As we showed in Section 6.4, timestamp sharing is essential for the performance of the TS queue. Nevertheless, we show the results of the single-producer multiple-consumer benchmark for completeness.

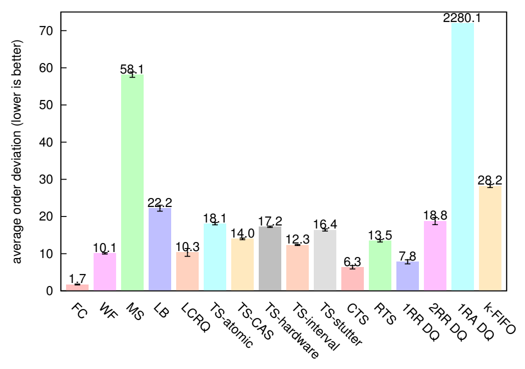
Note that the order deviation measured in the producer-consumer benchmark (i.e. the order deviation of both enqueue and dequeue operations) is not the order deviation of enqueues in the multiple-producer single-consumer benchmark plus the order deviation of dequeues in the single-producer multiple-consumer benchmark. Some additional order deviation may be introduced by the interaction between concurrent enqueue and dequeue operations (i.e. emptiness checks). It can also happen that order deviation introduced by enqueue operations is canceled again later by the order deviation of dequeue operations.



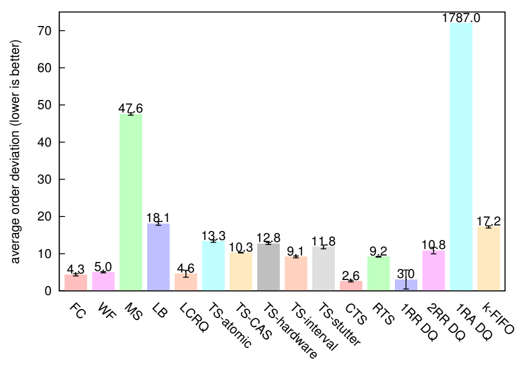
(a) High-contention producer-consumer benchmark on the 40-core machine.



(b) High-contention producer-consumer benchmark on the 64-core machine.



(c) Low-contention producer-consumer benchmark on the 40-core machine.



(d) Low-contention producer-consumer benchmark on the 64-core machine.

Figure 8.1: Order deviation in the producer-consumer benchmark on the 40-core machine with 40 producers and 40 consumers (left), and on the 64-core machine with 32 producers and 32 consumers (right).

8.4 Evaluation

8.4.1 Overall Order Deviation

Figure 8.1 shows the order deviation in the producer-consumer benchmark with 40 producers and 40 consumers on the 40-core, and with 32 producers and 32 consumers on the 64-core machine, averaged over 5 runs. The error bars show the standard deviation. The low standard deviation of most data structures indicates that the measurement method is robust.

In all measurements the TS-CAS queue and the TS-interval queue have a lower order deviation than the TS-atomic queue and the TS-hardware queue. This indicates that failing `tryRem` operations increases order deviation more than timestamp sharing.

The order deviation of the CTS queue is among the lowest of all queue implementations. The reason is that for the CTS queue the linearization point of a dequeue operation is not the point in time when the element is removed, but the time when the dequeue operation gets its timestamp. The order deviation of the RTS queue, where the linearization point of the dequeue operation is the removal of the element, is much higher than the order deviation of the CTS queue. Additionally the RTS queue is not linearizable with respect to QUEUE, which increases the order deviation further. One reason for the low order deviation of the CTS queue may be that it does not provide an emptiness check. Dealing with an empty queue state can increase the order deviation.

Of the other linearizable queue implementations the FC queue and the WF queue show the lowest order deviation. These queue implementations are not the fastest implementations, especially not the WF queue. However, in both queues threads help each other to complete their operations. It seems that helping can reduce order deviation.

The order deviation of both the MS queue and the LB queue are higher than the order deviation of all the TS queues, which indicates that enforcing a total order on elements can implicitly increase the order deviation more than timestamp sharing. Especially the order deviation of the MS queue is three times as high as the order deviation of the other queues. The reason is that the backoff algorithm which is used to improve the performance of the MS queue by a factor of 6 also increases the order deviation by a factor of two.

On the 40-core machine the order deviation of LCRQ shows a slightly high standard deviation. The high standard deviation seems to come from the slow path in the code which deals with an empty queue state. In three of the five runs of the high-contention benchmark there exists not a single dequeue-empty operation, which means that the slow path is not executed, resulting in a low order deviation. In one run, however, there are more than 15000 dequeue-empty operations, nearly 4% of all operations. It seems for LCRQ that reaching an empty state increases the probability that an empty state is reached again later. For 1RR DQ, which shows a higher standard deviation on the 64-core machine, there is no correlation between the number of dequeue-empty operations and the standard deviation.

Other than 1RA DQ, the relaxed queue implementations have an order deviation comparable to the order deviation of the linearizable queue implementations. In the low contention benchmarks 1RR DQ is even one of the data structures with the lowest order deviation. Also in the high-contention benchmark on the 40-core machine 1RR DQ shows a lower order deviation than most other queue implementations. This means that even if a data structure implementation allows some non-determinism according to its specification, the actual order deviation can still be less than the order deviation introduced by enforcing a deterministic specification.

The order deviation of 2RR DQ is always lower than the order deviation of the MS queue and the LB queue. Interestingly, 2RR DQ has a lower order deviation in the high-contention benchmark on the 64-core machine than 1RR DQ, although in principle more out-of-order behavior is possible in 2RR DQ than in 1RR DQ. This also suggests that good performance can impact order deviation more than a relaxed semantics.

The order deviation of the k-FIFO queue is higher than the order deviation of most linearizable queue implementations. However, on the 64-core machine the order deviation of the MS queue is twice as much as the order deviation of the k-FIFO queue.

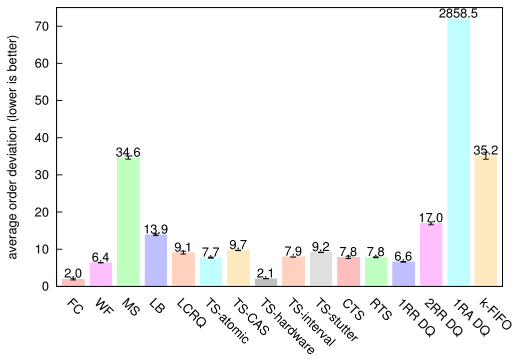
8.4.2 Order Deviation Introduced by Enqueue Operations

Figure 8.2 shows the order deviation introduced by enqueue operations measured in the multiple-producer single-consumer benchmark with 40 producers on the 40-core and with 32 producers on the 64-core machine, averaged over 5 runs.

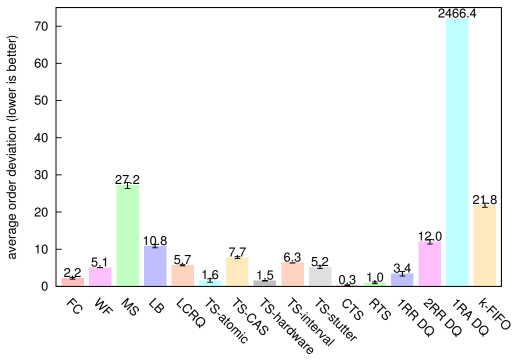
As expected the enqueue operations of the TS-hardware queue consistently show low order deviation. The reason is that the enqueue operation of the TS-hardware queue is really fast, and only few timestamp sharing is happening. If elements were timestamped before they were inserted into the SP pool, then an even lower order deviation would be possible.

The TS-interval queue and the TS-CAS queue intentionally increase timestamp sharing and therefore create a potential for order deviation. As a consequence the order deviation of these two TS queues is higher than the order deviation of the TS-hardware queue. However, the MS queue and the LB queue show much higher order deviation, although these queues introduce order deviation only implicitly through poor performance.

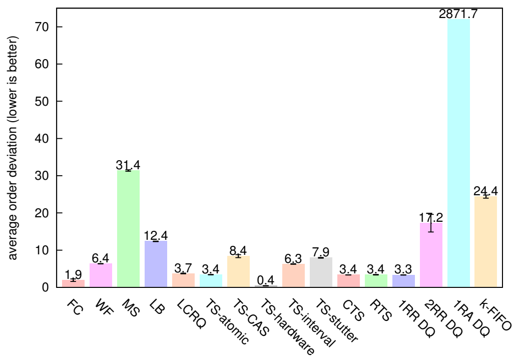
On the 40-core machine already the use of an atomic fetch-and-increment instruction (e.g. within TS-atomic) introduces some order deviation. In the high-contention benchmark it is even as much as the order deviation introduced by TS-interval timestamping and nearly as much as introduced by TS-CAS timestamping. This is sur-



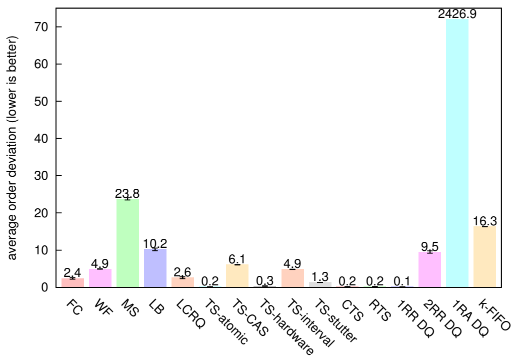
(a) High contention multiple-producer single-consumer benchmark on the 40-core machine.



(b) High contention multiple-producer single-consumer benchmark on the 64-core machine.



(c) Low contention multiple-producer single-consumer benchmark on the 40-core machine.



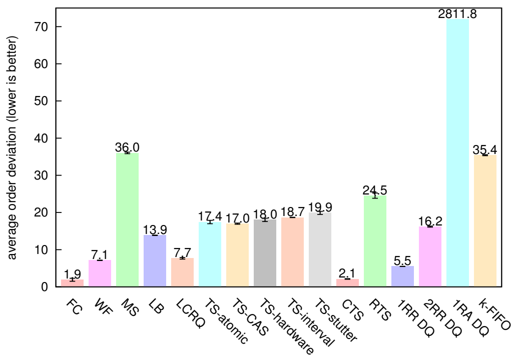
(d) Low contention multiple-producer single-consumer benchmark on the 64-core machine.

Figure 8.2: Order deviation in the multiple-producer single-consumer benchmark on the 40-core machine with 40 producers (left), and on the 64-core machine with 32 producers (right).

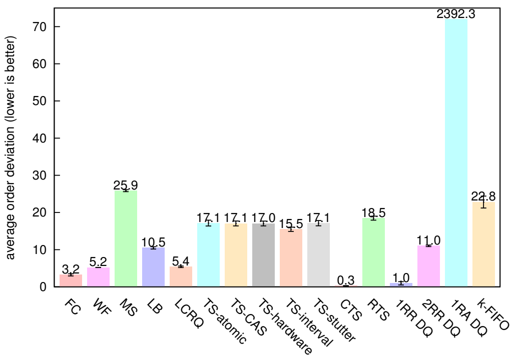
prising as both these timestamping algorithms introduce order deviation explicitly through timestamp sharing, whereas for the TS-atomic queue timestamp sharing is not possible. With less contention on the atomic counter, the order deviation of the TS-atomic queue decreases more than the order deviation of the TS-interval queue and the TS-CAS queue (Figure 8.2c). On the 64-core machine the use of an atomic fetch-and-increment instruction introduces only insignificant order deviation. Therefore the order deviation of the TS-atomic queue, the CTS queue, the RTS queue, and 1RR DQ is lower than on the 40-core machine.

In principle also the element order in LCRQ is mainly based on a fetch-and-increment instruction, which would suggest that also the order deviation of LCRQ on the 64-core machine should be low. However, LCRQ shows a higher order deviation than the other fetch-and-increment based queues. This indicates that whenever the element order in LCRQ is not defined by the fetch-and-increment, e.g. when a new segment is created, then significant order deviation is introduced.

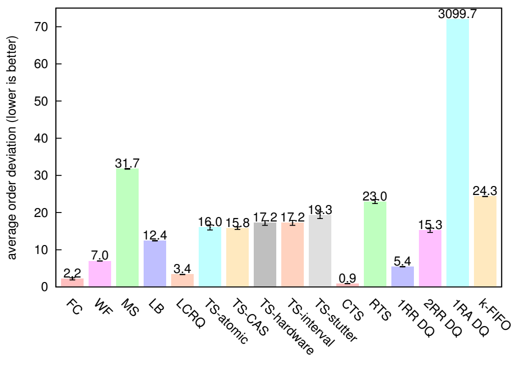
Although 1RR DQ is not linearizable with respect to the sequential specification of a queue, our results of order deviation suggest that it is unlikely that 1RR DQ violates queue semantics in our experiments. Also in the multiple-producer single-consumer benchmark the order deviation of 1RR DQ is among the lowest of all data structures, especially in the low-contention benchmarks. It seems that in the high-contention benchmark 1RR DQ gains additional performance at the cost of more order deviation.



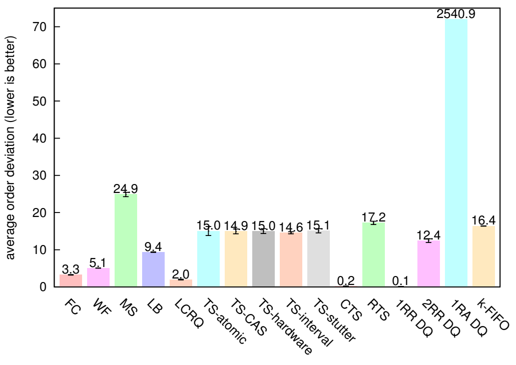
(a) High-contention single-producer multiple consumer benchmark on the 40-core machine.



(b) High-contention single-producer multiple consumer benchmark on the 64-core machine.



(c) Low-contention single-producer multiple consumer benchmark on the 40-core machine.



(d) Low-contention single-producer multiple consumer benchmark on the 64-core machine.

Figure 8.3: Order deviation in the single-producer multiple-consumer benchmark on the 40-core machine with 40 producers (left), and on the 64-core machine with 32 producers (right).

The order deviation of 2RR DQ is always much lower than the order deviation of the MS queue. The order deviation of the k-FIFO queue is always slightly higher than the order deviation of most other queues, but not as high as the MS queue.

8.4.3 Order Deviation Introduced by Dequeue Operations

Figure 8.3 shows the order deviation introduced by dequeue operations measured in the single-producer multiple-consumer benchmark with 40 consumers on the 40-core machine and with 32 consumers on the 64-core machine, averaged over 5 runs.

As expected, in this benchmark the order deviation of the TS queue is mostly independent of the used timestamping algorithm. The reason is that elements are enqueued sequentially and therefore all timestamping algorithms define a total order on the elements in the queue.

The order deviation of most queues (see Figure 8.1) is not the sum of the order deviation introduced by their enqueue operations (see Figure 8.2) and dequeue operations (see Figure 8.3). For some queues, e.g. the FC queue and the WF queue, the order deviation of the queue is less than the sum of the order deviation introduced by their operations. The reason is that the order deviation introduced by enqueue operations can be canceled by the order deviation introduced by dequeue operations. For example, if a scheduling artifact delays an enqueue operation and thereby introduces some order deviation, then another scheduling artifact can delay dequeue operations in the opposite way such that no global order deviation can be observed.

For other queues, like the MS queue, the order deviation of the queue is more than the sum of the order deviation introduced by their operations. The reason may be the interaction between enqueue and dequeue operations or additional contention on the memory. For the MS queue, the backoff algorithm is the reason why the order deviation in the producer-consumer benchmark is more than the sum of the order deviation in the other benchmarks. Without backoff algorithm the order deviation of the queue would be less than the sum of the order deviation introduced by its operations.

Chapter 9

Related Work and Conclusions

9.1 Related Work

We have already described other stack, queue, and deque implementations in the performance evaluation sections, Section 3.2, Section 6.4, and Section 7.2. In this chapter we describe work which is related to our timestamping approach, to the linearizability proof of the TS stack, and to our implementation.

Timestamping. Our approach was initially inspired by Attiya et al.'s Laws of Order paper [AGH$^+$11], which proves that any linearizable stack, queue, or deque necessarily uses the RAW or AWAR patterns in its remove operation. While attempting to extend this result to insert operations, we were surprised to discover a counter-example: our timestamping data structures. We believe the Basket Queue [HSS07] was the first algorithm to exploit in practice the fact that enqueues need not take effect in order of their atomic operations, although unlike our approach it does not avoid strong synchronization when inserting.

Gorelik and Hendler use timestamping in their AFC queue [GH13]. As in our timestamping data structures, enqueued elements are timestamped and stored in single-producer buffers. Our approach differs in several respects.

The AFC queue uses flat-combining-style consolidation – that is, a combiner thread merges timestamps into a total order. As a result, the AFC queue is blocking. In our timestamping approach we avoid enforcing an internal total order, and instead allow non-blocking parallel removal.

Removal in the AFC queue depends on the expensive consolidation process, and, as a result, their producer-consumer benchmark shows remove performance significantly worse than other flat-combining queues. Interval timestamping lets the timestamping data structures trade insertion and removal cost, avoiding this problem.

Timestamps in the AFC queue are generated only with Lamport clocks [Lam78]. We also experiment with Lamport clocks – see TS-stutter in Section 2.2. In our experiments, however, the Lamport clocks are actually slower than the other timestamping algorithms, see Section 3.2.

Finally, in the AFC queue elements are timestamped *before* being inserted – in our approach, this is reversed. This seemingly trivial difference is key for the correctness of the TS stack, see Figure 1.5 in the introduction. It also improves timestamp-based elimination, which is important to the TS stack's and TS deque's performance.

The LCRQ queue [AM13] and the SP queue [HPS13] both index elements using an atomic counter. However, dequeue operations do not look for *one of* the youngest elements as in our approach, but rather for the element with the enqueue index that matches the dequeue index *exactly*. Both approaches fall back to a slow path when the dequeue counter becomes higher than the enqueue counter. In contrast to indices, timestamps in our approach need not be unique or even ordered, and the performance in our approach does not depend on a fast path and a slow path, but only on the number of elements which share the same timestamp.

The LCRQ and the SP queue are quite similar to the CTS queue. However, different to these queues the CTS queue does not provide a linearizable emptiness check but instead blocks as long as no element is found in the queue.

The `RDTSCP` instruction we use to generate hardware timestamps has been used in the design of software transactional memory by Ruan et al. [RLS13], who investigate the instruction's multi-processor synchronisation behaviour.

Correctness. Our stack theorem lets us prove that the TS stack is linearizable with respect to sequential stack semantics. This theorem builds on an idea of Henzinger et al., who have a similar theorem for queues [HSV13]. Their theorem is defined (almost) entirely in terms of the sequential order on methods – what we call precedence, pr. That is, they need not generate a linearization order. In contrast, our stack theorem requires a relation between inserts and removes. We suspect it is impossible to define such a theorem for stacks without an auxiliary insert-remove relation (see Section 4.1.1).

A stack must respect several non-LIFO correctness properties: elements should not be lost or duplicated, and pop should correctly report when the stack is empty. Henzinger et al. build these properties into their theorem, making it more complex and arguably harder to use. Furthermore, each dequeue that returns EMPTY requires a partition 'before' and 'after' the operation, effectively reintroducing a partial linearization order. However, these correctness properties are orthogonal to LIFO ordering, and so we simply require that the algorithm also respects set semantics.

Implementation features. Our TS stack implementation reuses concepts from several previous data structures.

Storing elements in multiple partial data structures is used in the distributed queue [HHK+13], where insert and remove operations are distributed between partial queues using a load balancer. One can view the SP pools as partial queues and the TS stack itself as the load balancer. The emptiness check we use in the TS stack, TS queue, and TS deque also originates from the distributed queues. However, our data structures leverages the performance of distributed queues while preserving sequential semantics.

The idea of removing elements first logically and then unlinking them from the link structure later has already been proposed by Harris [Har01] in a non-blocking linked list algorithm.

Elimination originates in the elimination-backoff stack [HSY04]. However, in the TS stack, elimination works by comparing timestamps rather than by accessing a collision array. As a result, in the TS stack a pop which eliminates a concurrent push is faster than a normal uncontended pop. In the elimination-backoff stack such an eliminating pop is slower, as synchronization on the collision array requires at least three successful `CAS` operations instead of just one.

9.2 Conclusions

In this thesis we presented a novel approach to implementing concurrent ordered data structures like stacks, queues, and deques. Instead of ordering elements in a link structure (e.g. a linked list) we order elements by timestamps. Thereby we can avoid ordering elements which do not have to be ordered, for example elements which got inserted concurrently. Elements are removed by first searching for a candidate element and then removing it. Depending on the semantics of the data structure, the candidate element may be the element with the latest timestamp (e.g. stack) or the element with the earliest timestamp (e.g. queue).

We provide stack and queue algorithms which use our approach, we prove their correctness in terms of linearizability, and we analyze their performance in experiments. In our experiments the TS stack outperforms the elimination backoff stack [HSY04], the fastest concurrent stack we are aware of. The TS queue is outperformed only by LCRQ queue [AM13].

Additionally to the TS stack and the TS queue we sketch a TS deque algorithm and evaluate its performance. The performance of the TS deque is significantly better than the performance of a flat combining dequeue [HIST10] in our experiments. A correctness proof, however, remains future work.

For the linearizability proof of the TS queue we use the queue theorem published by Henzinger et al. [HSV13]. This queue theorem defines a set of conditions which imply linearizability with respect to queue semantics. For the linearizability proof of the TS stack we first established a similar theorem for stacks and proved it correct with the theorem prover Isabelle HOL [NWP02]. A deque theorem, which seems to be essential for the linearizability proof of the TS deque, remains future work.

In linearizable data structure implementations, operations may not take effect in the order they were invoked. We analyzed experimentally how much operations are reordered between their invocation and the time they take effect in various concurrent queue implementations. Our experiments show that in implementations like the Michael Scott queue [MS96] operations can be reordered more than in the TS queue.

References

[AGH$^+$11] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M.M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*. ACM, 2011.

[AHS94] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41, 1994.

[AM13] Y. Afek and A Morrison. Fast concurrent queues for x86 processors. In *PPoPP*. ACM, 2013.

[BDG13] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*. ACM, 2013.

[BEEH15] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *POPL*. ACM, 2015.

[BNHS11] G. Bar-Nissan, D. Hendler, and A. Suissa. A dynamic elimination-combining stack algorithm. In *OPODIS*. Springer, 2011.

[DHK14] M. Dodds, A. Haas, and C. M. Kirsch. Fast concurrent data-structures through explicit timestamping. Technical Report TR 2014–03, Department of Computer Sciences, University of Salzburg, 2014.

[DHK15] M. Dodds, A. Haas, and C.M. Kirsch. A scalable, correct time-stamped stack. In *POPL*. ACM, 2015.

[DHM13] D. Dice, D. Hendler, and I. Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par*. Springer-Verlag, 2013.

[GH13] M. Gorelik and D. Hendler. Brief announcement: an asymmetric flat-combining based queue algorithm. In *PODC*. ACM, 2013.

[Har01] T.L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*. Springer, 2001.

[HHK+13] A. Haas, T.A. Henzinger, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed queues in shared memory—multicore performance and scalability through quantitative relaxation. In *CF*. ACM, 2013.

[HHK+15] A. Haas, T. Hütter, C.M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova. Scal: A benchmarking suite for concurrent data structures. In *NETYS*. Springer, 2015.

[HHL+05] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*. Springer, 2005.

[HIST10] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*. ACM, 2010.

[HKLP12] A. Haas, C.M. Kirsch, M. Lippautz, and H. Payer. How FIFO is your concurrent FIFO queue? In *RACES*. ACM, 2012.

[HKP+13] T.A. Henzinger, C.M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *POPL*. ACM, 2013.

[HPS13] T.A. Henzinger, H. Payer, and A. Sezgin. Replacing competition with cooperation to achieve scalable lock-free FIFO queues. Technical Report IST-2013-124-v1+1, IST Austria, 2013.

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.

[HSS07] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *OPODIS*. Springer, 2007.

[HSV13] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*. Springer, 2013.

[HSY04] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*. ACM, 2004.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[Int13] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 3b: System programming guide, part 2, 2013. URL: http://download.intel.com/products/processor/manual/253669.pdf.

[KLP13] C.M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. In *PaCT*. Springer, 2013.

[KP11] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPoPP*. ACM, 2011.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21, July 1978.

[Mic03] M. Michael. CAS-based lock-free algorithm for shared deques. In *Euro-Par*. Springer, 2003.

[MNSS05] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA*. ACM, 2005.

[MS96] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*. ACM, 1996.

[NR07] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*. ACM, 2007.

[NWP02] T. Nipkow, M. Wenzel, and L.C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[ope] OpenJDK: An open-source implementation of the Java platform, standard edition. URL: http://openjdk.java.net.

[RLS13] W. Ruan, Y. Liu, and M. Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. *ACM Trans. Archit. Code Optim.*, 10, December 2013.

[SSO^{+}10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7), 2010.

[ST08] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68, 2008.

[Tre86] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

List of figures

List of tables

List of listings

List of Theorems

Index

Appendix A

Isabelle Proofs

A.1 The Stack Theorem

theory *stack_theorem*
imports *Main concurrent_histories remove_relaxed_stack elimination_stack empty_stack*

begin

theorem *thm_stack_theorem : "linearizable_set pre val emp ⟹ order_correct pre val ⟹ linearizable_stack pre val emp"*
by *(metis (poly_guards_query) concurrent_empty_stack_linearizable elimination_theorem linearizable_set_def linearizable_set_then_wellformed order_correct_then_remove_relaxed_stack pre_conc_history_then_non_empty_pre_conc_history remove_relaxed_stack_linearizable conc_history.abcd)*

end

A.2 Basic Definitions

theory *concurrent_histories*
imports *Main*
begin

definition *abcd :: "'a rel ⇒ bool"*
where
"abcd r ⟷ (∀a b c d. (a, b) ∈ r ⟶ (c, d) : r ⟶ (a, d)∈r ∨ (c, b)∈r)"

lemma *abcdI : "(⋀a b c d. (a, b) ∈ r ⟹ (c, d) ∈ r ⟹ (a, d) ∈ r ∨ (c, b) ∈ r) ⟹ abcd r"*
by *(metis abcd_def)*

lemma *abcdD : "abcd r ⟹ (a, b) ∈ r ⟹ (c, d) ∈ r ⟹ (a, d) ∈ r ∨ (c, b) ∈ r"*

```
by (metis abcd_def)

lemma lem_total_then_abcd : "total r ⟹ trans r ⟹ abcd r"
apply(unfold abcd_def)
by (metis UNIV_I total_on_def transD)

lemma lem_transitive_irreflexive_then_acyclic : "trans r ⟹ irrefl r ⟹
acyclic r"
by (metis acyclic_irrefl trancl_id)

lemma lem_irreflexive_then_not_equal : "irrefl r ⟹ (a, b) ∈ r ⟹ a ≠
b"
by (metis irrefl_def)


locale conc_history =
  fixes pre :: "'a rel"
  assumes pre_transitive: "trans pre"
  assumes pre_antisymmetric: "antisym pre"
  assumes pre_irreflexive: "irrefl pre"
  assumes abcd : "abcd pre"

lemma conc_historyI : "trans pre ⟹ antisym pre ⟹ irrefl pre ⟹ abcd
pre ⟹ conc_history pre"
apply (unfold conc_history_def)
by metis

lemma conc_historyD : "conc_history pre ⟹ trans pre ∧ antisym pre ∧
irrefl pre ∧ abcd pre"
apply (unfold conc_history_def)
by metis


definition val_order :: "'a rel ⇒ bool"
where
  "val_order val ⟷ (∀ ia ra. (ia, ra) ∈ val ⟶ ¬(∃ b. (b, ia) ∈ val ∨
(ra, b) ∈ val))"


definition push_once :: "'a rel ⇒ bool"
where
  "push_once val ⟷ (∀ ia ib ra. (ia, ra) ∈ val ⟶ (ib, ra) ∈ val ⟶
ia = ib)"

definition pop_once :: "'a rel ⇒ bool"
where
  "pop_once val ⟷ (∀ ia ra rb. (ia, ra) ∈ val ⟶ (ia, rb) ∈ val ⟶ ra
= rb)"

definition val_then_not_pr :: "'a rel ⇒ 'a rel ⇒ bool"
where
  "val_then_not_pr pre val ⟷ (∀ ia ra. (ia, ra) ∈ val ⟶(ra, ia)∉pre)"

lemma val_then_not_prI : "∀ ia ra . (ia, ra) ∈ val ⟶ (ra, ia) ∉ pre
⟹ val_then_not_pr pre val"
by (metis val_then_not_pr_def)

definition isPush :: "'a rel ⇒ 'a ⇒ bool"
```

```
where
  "isPush val a ⟷ (∃ b . (a, b) ∈ val)"

lemma isPushD : "isPush val a ⟹ ∃ b . (a, b) ∈ val"
by (metis isPush_def)

lemma isPushI : "(a, b) ∈ val ⟹ isPush val a"
by (metis isPush_def)

definition isPop :: "'a rel ⇒ 'a ⇒ bool"
where
  "isPop val a ⟷ (∃ b . (b, a) ∈ val)"

lemma isPopD : "isPop val a ⟹ ∃ b . (b, a) ∈ val"
by (metis isPop_def)

lemma isPopI : "(a, b) ∈ val ⟹ isPop val b"
by (metis isPop_def)


definition sameVal :: "'a rel ⇒ 'a rel"
where "sameVal val = {(a, b) . (a, b) ∈ val ∨ (b, a) ∈ val}"

lemma sameValD : "(a, b) ∈ sameVal val ⟹ (a, b) ∈ val ∨ (b, a) ∈ val"
apply (unfold sameVal_def)
by (simp add : set_eq_iff)

lemma sameValI1 : "(a, b) ∈ val ⟹ (a, b) ∈ sameVal val"
apply (unfold sameVal_def)
by (simp add : set_eq_iff)

lemma sameValI2 : "(a, b) ∈ val ⟹ (b, a) ∈ sameVal val"
apply (unfold sameVal_def)
by (simp add : set_eq_iff)

lemma sameVal_commutative : "(a, b) ∈ sameVal val ⟹ (b, a) ∈ sameVal
val"
by (metis sameValD sameValI1 sameValI2)

definition elimination_free :: "'a rel ⇒ 'a rel ⇒ 'a set"
where
  "elimination_free pre val = {a . (∃ b . (a, b) ∈ sameVal val ∧ ((a, b) ∈
pre ∨ (b, a) ∈ pre))}"

lemma elimination_freeD : "a ∈ elimination_free pre val  ⟹ ∃ b . (a, b)
∈ sameVal val ∧ ((a, b) ∈ pre ∨ (b, a) ∈ pre)"
apply (unfold elimination_free_def)
by (simp add: set_eq_iff)

lemma elimination_freeI1 : "(a, b) ∈ sameVal val ⟹ (a, b) ∈ pre ⟹ a
∈ elimination_free pre val"
apply (unfold elimination_free_def)
apply (simp add: set_eq_iff)
by metis

lemma elimination_free_in_push_pop : "a ∈ elimination_free pre val ⟹ a
∈ {a . isPush val a ∨ isPop val a}"
apply (simp add: set_eq_iff)
```

by *(metis elimination_freeD isPop_def isPush_def sameValD)*

lemma *elimination_free_subset_push_pop : "elimination_free pre val ⊆ {a . isPush val a ∨ isPop val a}"*
by *(metis elimination_free_in_push_pop subsetI)*

lemma *finite_elimination_free : "finite {a . isPush val a ∨ isPop val a} ⟹ finite (elimination_free pre val)"*
by *(metis elimination_free_subset_push_pop finite_subset)*

lemma *elimination_freeI2 : "(a, b) ∈ sameVal val ⟹ (a, b) ∈ pre ⟹ b ∈ elimination_free pre val"*
apply *(unfold elimination_free_def)*
apply *(simp add: set_eq_iff)*
by *(metis sameValD sameValI1 sameValI2)*

definition *elimination :: "'a rel ⇒ 'a rel ⇒ 'a set"*
  where *"elimination pre val = {a . (isPush val a ∨ isPop val a) ∧ a ∉ elimination_free pre val}"*

lemma *eliminationD : "a ∈ elimination pre val ⟹ ∃ b . (a, b) ∈ sameVal val ∧ (a, b) ∉ pre ∧ (b, a) ∉ pre"*
apply *(unfold elimination_def)*
apply *(simp add:set_eq_iff)*
by *(metis elimination_freeI1 elimination_freeI2 isPop_def isPush_def sameValI1 sameValI2)*

definition *elimination_free_pre :: "'a rel ⇒ 'a rel ⇒'a rel"*
where
  "elimination_free_pre pre val = {(a, b) . a ∈ elimination_free pre val ∧ b ∈ elimination_free pre val ∧ (a, b) ∈ pre}"

lemma *elimination_free_preD : "(a, b) ∈ elimination_free_pre pre val ⟹ a ∈ elimination_free pre val ∧ b ∈ elimination_free pre val ∧ (a, b) ∈ pre"*
apply *(unfold elimination_free_pre_def)*
by *(simp add: set_eq_iff)*

lemma *elimination_free_preI : "a ∈ elimination_free pre val ⟹ b ∈ elimination_free pre val ⟹ (a, b) ∈ pre ⟹ (a, b) ∈ elimination_free_pre pre val"*
apply *(unfold elimination_free_pre_def)*
by *(simp add: set_eq_iff)*

definition *elimination_free_val :: "'a rel ⇒ 'a rel ⇒ 'a rel"*
where
  "elimination_free_val pre val = {(a, b) . a ∈ elimination_free pre val ∧ b ∈ elimination_free pre val ∧ (a, b) ∈ val}"

lemma *elimination_free_valD : "(a, b) ∈ elimination_free_val pre val ⟹ a ∈ elimination_free pre val ∧ b ∈ elimination_free pre val ∧ (a, b) ∈ val"*
apply *(unfold elimination_free_val_def)*
by *(simp add: set_eq_iff)*

lemma *elimination_free_valI : "a ∈ elimination_free pre val ⟹ b ∈ elimination_free pre val ⟹ (a, b) ∈ val ⟹ (a, b) ∈ elimination_free_val pre val"*

```
apply (unfold elimination_free_val_def)
by (simp add: set_eq_iff)

definition elimination_free_ir :: "'a rel ⇒ 'a rel ⇒ 'a rel ⇒ 'a rel"
where
  "elimination_free_ir pre val ir = {(a, b) . a ∈ elimination_free pre val
∧ b ∈ elimination_free pre val ∧ (a, b) ∈ ir}"

lemma elimination_free_irD : "(a, b) ∈ elimination_free_ir pre val ir ⟹
a ∈ elimination_free pre val ∧ b ∈ elimination_free pre val ∧ (a, b) ∈
ir"
apply (unfold elimination_free_ir_def)
by (simp add : set_eq_iff)

lemma elimination_free_irI : "a ∈ elimination_free pre val ⟹ b ∈
elimination_free pre val ⟹ (a, b) ∈ ir ⟹ (a, b) ∈ elimination_free_ir
pre val ir"
apply (unfold elimination_free_ir_def)
by (simp add: set_eq_iff)

definition ins_order :: "'a rel ⇒ 'a rel ⇒ 'a rel ⇒ 'a rel"
where
  "(ins_order pre val ir) = {(a, b) . isPush val a ∧ isPush val b ∧ ((a, b)
∈ pre ∨ (∃ c . (a, c) ∈ pre ∧ (c, b) ∈ ir))}"

lemma ins_orderI1 : "isPush val a ⟹ isPush val b ⟹ (a, b) ∈ pre ⟹
(a, b) ∈ ins_order pre val ir"
apply (unfold ins_order_def)
by (simp add: set_eq_iff)

lemma ins_orderI2 : "isPush val a ⟹ isPush val b ⟹ (a, c) ∈ pre ⟹
(c, b) ∈ ir ⟹ (a, b) ∈ ins_order pre val ir"
apply (unfold ins_order_def)
apply (simp add: set_eq_iff)
by metis

lemma ins_orderD : "(a, b) ∈ (ins_order pre val ir) ⟹ isPush val a ∧
isPush val b ∧ ((a, b) ∈ pre ∨(∃ c . (a, c) ∈ pre ∧ (c, b) ∈ ir))"
apply (unfold ins_order_def)
by (simp add: set_eq_iff)

definition rem_order :: "'a rel ⇒ 'a rel ⇒ 'a rel ⇒ 'a rel"
where
  "(rem_order pre val ir) = {(a, b) . isPop val a ∧ isPop val b ∧ ((a, b)
∈ pre ∨ (∃ c . (a, c) ∈ ir ∧ (c, b) ∈ ir))}"

lemma rem_orderI1 : "isPop val a ⟹ isPop val b ⟹ (a, b) ∈ pre ⟹ (a,
b) ∈ rem_order pre val ir"
apply (unfold rem_order_def)
by (simp add: set_eq_iff)

lemma rem_orderI2 : "isPop val a ⟹ isPop val b ⟹ (a, c) ∈ ir ⟹ (c,
b) ∈ ir ⟹ (a, b) ∈ rem_order pre val ir"
apply (unfold rem_order_def)
apply (simp add: set_eq_iff)
by metis
```

```
lemma rem_orderD : "(a, b) ∈ (rem_order pre val ir) ⟹ isPop val a ∧
isPop val b ∧ ((a, b) ∈ pre ∨(∃ c . (a, c) ∈ ir ∧ (c, b) ∈ ir))"
apply (unfold rem_order_def)
by (simp add: set_eq_iff)

definition alternating :: "'a rel ⇒ 'a rel ⇒ bool"
where
  " alternating val ir ⟷ (∀ a b . (a, b) ∈ ir ⟶ (isPush val a ∧ isPop
val b) ∨ (isPop val a ∧ isPush val b))
                                    ∧ (∀ a b . isPush val a ⟶ isPop val b
⟶ (a, b) ∈ ir ∨ (b, a) ∈ ir)"

lemma alternatingD : "alternating val ir ⟹ (∀ a b . (a, b) ∈ ir ⟶
(isPush val a ∧ isPop val b) ∨ (isPop val a ∧ isPush val b))
                                    ∧ (∀ a b . isPush val a ⟶ isPop val b
⟶ (a, b) ∈ ir ∨ (b, a) ∈ ir)"
by (smt alternating_def)

lemma alternatingI : "(∀ a b . (a, b) ∈ ir ⟶ (isPush val a ∧ isPop val
b) ∨ (isPop val a ∧ isPush val b))
                                    ⟹ (∀ a b . isPush val a ⟶ isPop val
b ⟶ (a, b) ∈ ir ∨ (b, a) ∈ ir)
                                    ⟹ alternating val ir"
by (metis (full_types) alternating_def)

definition order_correct :: "'a rel ⇒ 'a rel ⇒ bool"
where
  "order_correct pre val ⟷ (∃ ir . alternating val ir ∧ (acyclic (pre ∪
ir) ∧
    (∀ ia ra ib . (ia, ra) ∈ val ∧ (ia, ra) ∈ pre  ∧
       (ia, ib) ∈ (ins_order pre val ir) ∧ (ib, ra) ∈ ir ⟶
            (∃ rb . (ib, rb) ∈ val ∧ (ra, rb) ∉ (rem_order pre val
ir)))))"



lemma order_correctD : "order_correct pre val ⟹ (∃ ir . alternating val
ir ∧ (acyclic (pre ∪ ir) ∧
    (∀ ia ra ib . (ia, ra) ∈ val ∧ (ia, ra) ∈ pre ∧
       (ia, ib) ∈ (ins_order pre val ir) ∧ (ib, ra) ∈ ir ⟶
            (∃ rb . (ib, rb) ∈ val ∧ (ra, rb) ∉ (rem_order pre val
ir)))))"
by (metis order_correct_def)

lemma order_correctI : "alternating val ir ⟹ acyclic (pre ∪ ir) ⟹
    (∀ ia ra ib . (ia, ra) ∈ val ∧ (ia, ra) ∈ pre ∧
       (ia, ib) ∈ (ins_order pre val ir) ∧ (ib, ra) ∈ ir ⟶
            (∃ rb . (ib, rb) ∈ val ∧ (ra, rb) ∉ (rem_order pre val ir)))
⟹ order_correct pre val"
apply (unfold order_correct_def)
apply (rule_tac x="ir" in exI)
by (metis (erased, hide_lams))


locale wellformed_history =

  fixes pre :: "'a rel"
  fixes val :: "'a rel"

  fixes f :: "'a ⇒ nat"

  assumes pre_transitive : "trans pre"
  assumes pre_antisymmetric : "antisym pre"
  assumes pre_irreflexive : "irrefl pre"

  assumes operations_countable : "inj f"

  assumes operations_finite : "finite {a . isPush val a ∨ isPop val a}"


  assumes def_push_once : "push_once val"

  assumes def_pop_once : "pop_once val"

  assumes def_value : "val_order val"

  assumes def_push_or_pop : "(a, b) ∈ pre ⟹ (∃ c. (a, c) ∈ val ∨ (c, a)
∈ val) ∧ (∃ c . (b, c) ∈ val ∨ (c, b) ∈ val)"

  assumes def_val_then_not_pr: "val_then_not_pr pre val"

lemma wellformed_historyI : "⋀ f::'a⇒nat. trans pre ⟹ antisym pre ⟹
irrefl pre ⟹ inj f ⟹ finite {a . isPush val a ∨ isPop val a} ⟹
  push_once val ⟹ pop_once val ⟹ val_order val ⟹
  (∀ a b .(a, b) ∈ pre ⟶ (∃ c. (a, c) ∈ val ∨ (c, a) ∈ val) ∧ (∃ c .
(b, c) ∈ val ∨ (c, b) ∈ val)) ⟹
  val_then_not_pr pre val ⟹ wellformed_history pre val f"
apply (unfold wellformed_history_def)
by metis

lemma wellformed_historyD : "wellformed_history pre val f ⟹ trans pre ∧
antisym pre ∧ irrefl pre ∧ inj f ∧ finite {a . isPush val a ∨ isPop val a}
∧
  push_once val ∧ pop_once val ∧ val_order val ∧
  (∀ a b .(a, b) ∈ pre ⟶ (∃ c. (a, c) ∈ val ∨ (c, a) ∈ val) ∧ (∃ c .
(b, c) ∈ val ∨ (c, b) ∈ val)) ∧
  val_then_not_pr pre val "

apply (unfold wellformed_history_def)
by metis

context wellformed_history
begin

definition is_push :: "'a ⇒ bool"
where
  "is_push a ⟷ (∃b.(a, b)∈val)"

definition is_pop :: "'a ⇒ bool"
where
  "is_pop a ⟷ (∃b.(b, a)∈val)"

lemma is_popI : "(ia, ra) ∈ val ⟹ is_pop ra"
by (metis is_pop_def)
```

```
lemma is_popD : "is_pop ra ⟹ ∃ ia . (ia, ra) ∈ val"
by (metis is_pop_def)

lemma is_pushI : "(ia, ra) ∈ val ⟹ is_push ia"
by (metis is_push_def)

lemma is_pushD : "is_push ia ⟹ ∃ ra . (ia, ra) ∈ val"
by (metis is_push_def)

lemma lem_push_or_pop1: "(a, b) ∈ pre ⟹ is_push a ∨ is_pop a"
by (metis def_push_or_pop is_popI is_pushI)

lemma lem_push_or_pop2: "(a, b) ∈ pre ⟹ is_push b ∨ is_pop b"
by (metis def_push_or_pop is_popI is_pushI)

lemma lem_push_then_pop_exists: "is_push a ⟹ ∃b.(a, b)∈val ∧ is_pop b"
by (metis is_pop_def is_push_def)

lemma lem_pop_then_push_exists: "is_pop a ⟹ ∃b.(b, a)∈val ∧ is_push b"
by (metis is_pop_def is_push_def)

lemma lem_val_then_push_and_pop: "(a, b)∈val ⟹ (is_push a ∧ is_pop b)"
by (metis is_pop_def is_push_def)

end

lemma val_preserves_ef1 : "wellformed_history pre val f ⟹ (a, b) ∈ val
⟹ a ∈ elimination_free pre val ⟹ b ∈ elimination_free pre val"
by (metis elimination_freeD elimination_freeI2 pop_once_def sameValD
val_order_def val_then_not_pr_def wellformed_history.def_pop_once
wellformed_history.def_val_then_not_pr wellformed_history.def_value)

lemma val_preserves_ef2 : "wellformed_history pre val f ⟹ (a, b) ∈ val
⟹ b ∈ elimination_free pre val ⟹ a ∈ elimination_free pre val"
by (metis elimination_freeD elimination_freeI1 elimination_freeI2
push_once_def sameValD sameValI1 val_order_def
wellformed_history.def_push_once wellformed_history.def_value)

lemma alternating_then_ef_alternating : "wellformed_history pre val f ⟹
alternating val ir ⟹ alternating (elimination_free_val pre val)
(elimination_free_ir pre val ir)"
apply (rule alternatingI)
apply (rule allI)+
apply (rename_tac a b)
apply (rule impI)
apply (drule elimination_free_irD)
apply (erule conjE)+
apply (drule alternatingD)
apply (erule conjE)
apply (erule_tac x="a" in allE)
apply (erule_tac x="a" in allE)
apply (erule_tac x="b" in allE)
apply (erule_tac x="b" in allE)
apply (simp)
apply (erule disjE)
apply (erule conjE)
apply (rule disjI1)
apply (rule conjI)
apply (frule isPushD)
apply (erule exE)
apply (rename_tac ra)
apply (rule_tac b="ra" in isPushI)
apply (metis elimination_free_valI val_preserves_ef1)
apply (metis elimination_free_valI isPop_def val_preserves_ef2)
apply (rule disjI2)
apply (rule conjI)
apply (metis elimination_free_valI isPop_def val_preserves_ef2)
apply (metis elimination_free_valI isPush_def val_preserves_ef1)
apply (rule allI)+
apply (rename_tac a b)
apply (rule impI)+
by (metis (mono_tags, lifting) alternatingD elimination_free_irI
elimination_free_valD isPopD isPopI isPushD isPushI)


locale sequential_set = wellformed_history +
  assumes total_on_push : "is_push a ⟹ is_push b ⟹ a ≠ b ⟹ (a, b) ∈
pre ∨ (b, a) ∈ pre"
  assumes total_on_pop : "is_pop a ⟹ is_pop b ⟹ a ≠ b ⟹ (a, b) ∈
pre ∨ (b, a) ∈ pre"
  assumes total_on_push_pop : "is_push a ⟹ is_pop b ⟹ (a, b) ∈ pre ∨
(b, a) ∈ pre"

lemma sequential_setD : "sequential_set pre val f ⟹ wellformed_history
pre val f ∧
  (∀ a b . isPush val a ⟶ isPush val b ⟶ a ≠ b ⟶ (a, b) ∈ pre ∨ (b,
a) ∈ pre) ∧
  (∀ a b . isPop val a ⟶ isPop val b ⟶ a ≠ b ⟶ (a, b) ∈ pre ∨ (b,
a) ∈ pre)  ∧
  (∀ a b . isPush val a ⟶ isPop val b ⟶(a, b) ∈ pre ∨ (b, a) ∈ pre)"
apply (unfold sequential_set_def)
apply (erule conjE)
apply (rule conjI)
apply (assumption)
apply (unfold sequential_set_axioms_def)
apply (erule conjE)+
apply (rule conjI)
apply (metis isPushD wellformed_history.is_pushI)
by (metis isPopD isPushD wellformed_history.is_popI
wellformed_history.is_pushI)

lemma sequential_setI : "wellformed_history pre val f ⟹
  (∀ a b . isPush val a ⟶ isPush val b ⟶ a ≠ b ⟶ (a, b) ∈ pre ∨ (b,
a) ∈ pre) ⟹
  (∀ a b . isPop val a ⟶ isPop val b ⟶ a ≠ b ⟶ (a, b) ∈ pre ∨ (b,
a) ∈ pre)  ⟹
  (∀ a b . isPush val a ⟶ isPop val b ⟶(a, b) ∈ pre ∨ (b, a) ∈ pre)
⟹ sequential_set pre val f"
apply (unfold sequential_set_def)
apply (rule conjI)
apply (assumption)
apply (unfold sequential_set_axioms_def)
apply (rule conjI)+
apply (metis isPushI wellformed_history.is_pushD)
apply (rule conjI)
apply (metis isPopI wellformed_history.is_popD)
```

```
by (metis isPopI isPushI wellformed_history.is_popD
wellformed_history.is_pushD)


definition sequential_lifo :: "'a rel ⇒ 'a rel ⇒ bool"
where
  "sequential_lifo pre val ⟷ (∀ ia ib ra rb . (ia, ra) ∈ val ⟶ (ib,
rb) ∈ val ⟶ (ia, ib) ∈ pre ⟶ (ib, ra) ∈ pre ⟶ (rb, ra) ∈ pre)"

lemma ef_conc_history : "conc_history pre ⟹ conc_history
(elimination_free_pre pre val)"
apply (drule conc_historyD)
apply (erule conjE)+
apply (rule conc_historyI)
apply (metis elimination_free_preD elimination_free_preI trans_def)
apply (metis antisym_def elimination_free_preD)
apply (metis elimination_free_preD irrefl_def)
by (smt abcd_def elimination_free_preD elimination_free_preI)

lemma ef_finite: "finite {a. isPush val a ∨ isPop val a} ⟹
finite {a. isPush (elimination_free_val pre val) a ∨
             isPop (elimination_free_val pre val) a}"
by (metis (lifting, no_types) Collect_mono elimination_free_valD isPop_def
isPush_def rev_finite_subset)

lemma val_then_ef_val : "a ∈ elimination_free pre val ⟹ (a, b) ∈ val
⟹val_order val ⟹ pop_once val ⟹
 (a, b) ∈ elimination_free_val pre val"
apply (drule elimination_freeD)
apply (rule elimination_free_valI)
apply (metis elimination_freeI1 elimination_freeI2 sameValD sameValI1
sameValI2)
apply (metis elimination_freeI1 elimination_freeI2 pop_once_def sameValD
sameValI2 val_order_def)
by metis

lemma val_then_not_pr_mono : "val_then_not_pr prt val ⟹ pre ⊆ prt ⟹
val_then_not_pr pre val"
by (metis set_rev_mp val_then_not_pr_def)


definition all_popped_or_empty :: "'a rel ⇒ ('a ⇒ bool) ⇒ bool"
where
  "all_popped_or_empty val emp ⟷ (∀a. emp a ∨ (∃ b . (a, b) ∈ val ∨
(b, a) ∈ val))"

definition empty_then_not_val :: "'a rel ⇒ ('a ⇒ bool) ⇒ bool"
where
  "empty_then_not_val val emp ⟷ (∀a . emp a ⟶ ¬(∃ b . (a, b) ∈ val)
∧ ¬(∃ b . (b, a) ∈ val))"


locale wellformed_empty_history =

  fixes pre :: "'a rel"

  fixes val :: "'a rel"
  fixes emp :: "'a ⇒ bool"

  fixes f :: "'a ⇒ nat"

  assumes pre_transitive : "trans pre"
  assumes pre_antisymmetric : "antisym pre"
  assumes pre_irreflexive : "irrefl pre"
  assumes operations_countable : "inj f"

  assumes operations_finite : "finite {a . isPush val a ∨ isPop val a}"


  assumes def_push_once : "push_once val"

  assumes def_pop_once : "pop_once val"

  assumes def_val_order : "val_order val"

  assumes def_all_popped_or_empty : "all_popped_or_empty val emp"

  assumes def_empty_then_not_val : "empty_then_not_val val emp"

  assumes def_val_then_not_pr : "val_then_not_pr pre val"

lemma wellformed_empty_historyD : " wellformed_empty_history pre val emp f
⟹
trans pre
∧ antisym pre
∧ irrefl pre
∧ inj f
∧ finite {a . isPush val a ∨ isPop val a}
∧  push_once val
∧ pop_once val
∧ val_order val
∧ all_popped_or_empty val emp
∧ empty_then_not_val val emp
∧ val_then_not_pr pre val"
apply (unfold wellformed_empty_history_def)
by metis

lemma wellformed_empty_historyI :
  "⋀ f::'a⇒nat.
  trans pre ⟹
  antisym pre ⟹
  irrefl pre ⟹
  inj f ⟹
   finite {a . isPush val a ∨ isPop val a} ⟹
   push_once val ⟹
   pop_once val ⟹
   val_order val ⟹
   all_popped_or_empty val emp ⟹
   empty_then_not_val val emp ⟹
   val_then_not_pr pre val ⟹ wellformed_empty_history pre val emp f"
apply (unfold wellformed_empty_history_def)
by metis

context wellformed_empty_history
begin
```

```
definition is_push :: "'a ⇒ bool"
where
  "is_push a ⟷ (∃b.(a, b)∈val)"

definition is_pop :: "'a ⇒ bool"
where
  "is_pop a ⟷ (∃b.(b, a)∈val)"

lemma lem_empty_or_push_or_pop : "emp a ∨ is_push a ∨ is_pop a"
by (metis all_popped_or_empty_def def_all_popped_or_empty is_pop_def
is_push_def)

lemma lem_push_then_pop_exists: "is_push a ⟹ ∃b.(a, b)∈val ∧ is_pop b"
by (metis is_pop_def is_push_def)

lemma lem_pop_then_push_exists: "is_pop a ⟹ ∃b.(b, a)∈val ∧ is_push b"
by (metis is_pop_def is_push_def)

lemma valD: "(a, b) ∈ val ⟹ (is_push a ∧ is_pop b)"
by (metis is_pop_def is_push_def)

end


locale sequential_empty_history = wellformed_empty_history +
  assumes def_total : "total pre"

lemma sequential_empty_historyD : "sequential_empty_history pre val emp f
⟹
wellformed_empty_history pre val emp f
∧ total pre"
apply (unfold sequential_empty_history_def)
apply (erule conjE)
apply (rule conjI)
apply (assumption)
apply (unfold sequential_empty_history_axioms_def)
by (assumption)

lemma sequential_empty_historyI : "wellformed_empty_history pre val emp f
⟹ total pre ⟹
      sequential_empty_history pre val emp f"
apply (unfold sequential_empty_history_def)
apply (rule conjI)
apply (assumption)
apply (unfold sequential_empty_history_axioms_def)
by (assumption)


definition correct_empty :: "'a rel ⇒ 'a rel ⇒ ('a ⇒ bool) ⇒ bool"
where
  "correct_empty pre val emp ⟷ (∀ a . emp a ⟶ (∀ ib rb . (ib, rb) ∈
val ⟶ ((ib, a) ∈ pre ∧ (rb, a) ∈ pre) ∨ ((a, ib) ∈ pre ∧ (a, rb) ∈
pre)))"

definition non_empty_pre :: "'a rel ⇒ ('a ⇒ bool) ⇒ 'a rel"
where
   "non_empty_pre pre emp = {(a, b) . (a, b) ∈ pre ∧ ¬ emp a ∧ ¬ emp b}"
```

```
lemma non_empty_preI : "(a, b) ∈ pre ⟹ ¬ emp a ⟹ ¬ emp b ⟹ (a, b)
∈ non_empty_pre pre emp"
apply (unfold non_empty_pre_def)
by (simp add: set_eq_iff)

lemma non_empty_preD : "(a, b) ∈ non_empty_pre pre emp ⟹ ¬ emp a ∧ ¬
emp b ∧ (a, b) ∈ pre"
apply (unfold non_empty_pre_def)
by (simp add: set_eq_iff)

lemma non_empty_pre_then_pre : "(a, b) ∈ non_empty_pre pre emp ⟹ (a, b)
∈ pre ∧ ¬ emp a ∧ ¬ emp b"
by (metis (lifting) mem_Collect_eq non_empty_pre_def split_conv)

lemma pre_conc_history_then_non_empty_pre_conc_history : "conc_history pre
⟹ conc_history (non_empty_pre pre emp)"
apply (rule conc_historyI)
apply (drule conc_historyD)
apply (erule conjE)+
apply (rule transI)
apply (drule non_empty_pre_then_pre)+
apply (erule conjE)+
apply (metis non_empty_preI transE)
apply (metis antisym_subset conc_historyD non_empty_pre_then_pre subrelI)
apply (metis conc_historyD irrefl_def non_empty_pre_then_pre)
by (smt abcdD abcdI conc_history.abcd non_empty_preI
non_empty_pre_then_pre)

lemma ef_all_popped : "∀a b. (a, b) ∈ pre ⟶
             (emp a ∨ (∃c. (a, c) ∈ val ∨
                  (c, a) ∈ val)) ∧
             (emp b ∨ (∃c. (b, c) ∈ val ∨
                  (c, b) ∈ val)) ⟹
              val_order val ⟹ pop_once val ⟹ push_once val ⟹
              (a, b) ∈ elimination_free_pre (non_empty_pre pre emp) val ⟹
             (∃c. (a, c) ∈ elimination_free_val (non_empty_pre pre emp) val
∨
            (c, a) ∈ elimination_free_val (non_empty_pre pre emp) val) ∧
      (∃c. (b, c) ∈ elimination_free_val (non_empty_pre pre emp) val ∨
            (c, b) ∈ elimination_free_val (non_empty_pre pre emp) val)"
apply (drule elimination_free_preD)
apply (erule conjE)
apply (erule_tac x="a" in allE)
apply (erule_tac x="b" in allE)
apply (erule conjE)
apply (erule impE)
apply (metis non_empty_preD)
apply (erule conjE)
apply (rule conjI)
apply (erule disjE)
apply (metis non_empty_preD)
apply (erule exE)+
apply (rule_tac x="c" in exI)
apply (erule disjE)
apply (rule disjI1)
apply (rule elimination_free_valI)
apply (assumption)
apply (metis non_empty_preD)
apply (metis non_empty_preD)
apply (metis elimination_freeD elimination_freeI1 elimination_freeI2
push_once_def sameValD sameValI1 val_order_def val_then_ef_val)
by (metis elimination_freeD elimination_freeI1 elimination_freeI2
non_empty_pre_then_pre push_once_def sameValD sameValI1 val_then_ef_val)

lemma ef_empty_mono : "pre ⊆ prt ⟹ (elimination_free_pre (non_empty_pre
pre emp) val) ⊆ prt"
apply (rule subrelI)
apply (drule elimination_free_preD)
apply (erule conjE)+
by (metis non_empty_preD set_rev_mp)

lemma ef_val_then_val : "(a, b) ∈ (elimination_free_val (non_empty_pre pre
emp) val) ⟹ (a, b) ∈ val"
by (metis elimination_free_valD)

lemma ef_push_then_empty_push : "isPush (elimination_free_val pre val) a
⟹ empty_then_not_val val emp ⟹
       isPush (elimination_free_val (non_empty_pre pre emp) val) a"
apply (drule isPushD)
apply (erule exE)
apply (rule_tac b="b" in isPushI)
apply (drule elimination_free_valD)
apply (erule conjE)+
apply (rule elimination_free_valI)
apply (drule elimination_freeD)
apply (erule exE)
apply (erule conjE)
apply (erule disjE)
apply (rule_tac b="ba" in elimination_freeI1)
apply (assumption)
apply (metis empty_then_not_val_def non_empty_preI sameValD)
apply (metis (hide_lams, mono_tags) elimination_freeI2
empty_then_not_val_def non_empty_preI sameValD sameValI1 sameValI2)
apply (drule_tac a="b" in elimination_freeD)
apply (erule exE)
apply (erule conjE)
apply (erule disjE)
apply (rule_tac b="ba" in elimination_freeI1)
apply (assumption)
apply (metis empty_then_not_val_def non_empty_preI sameValD)
apply (metis (hide_lams, mono_tags) elimination_freeI2
empty_then_not_val_def non_empty_preI sameValD sameValI1 sameValI2)
by assumption

lemma empty_push_then_ef_push : "isPush (elimination_free_val
(non_empty_pre pre emp) val) a ⟹
  isPush (elimination_free_val pre val) a"
apply (drule isPushD)
apply (erule exE)
apply (rule_tac b="b" in isPushI)
apply (drule elimination_free_valD)
apply (erule conjE)+
apply (rule elimination_free_valI)
apply (metis elimination_freeD elimination_freeI1 elimination_freeI2
non_empty_pre_then_pre sameVal_commutative)
```

```
apply (metis elimination_freeD elimination_freeI1 elimination_freeI2
non_empty_pre_then_pre sameVal_commutative)
by metis

lemma ef_pop_then_empty_pop : "isPop (elimination_free_val pre val) a ⟹
empty_then_not_val val emp ⟹ val_order val ⟹ pop_once val ⟹ push_once
val ⟹
       isPop (elimination_free_val (non_empty_pre pre emp) val) a"
apply (drule isPopD)
apply (erule exE)
apply (rule_tac a="b" in isPopI)
apply (drule elimination_free_valD)
apply (erule conjE)+
apply (rule elimination_free_valI)
apply (drule elimination_freeD)
apply (erule exE)
apply (erule conjE)
apply (erule disjE)
apply (rule_tac b="ba" in elimination_freeI1)
apply (assumption)
apply (metis empty_then_not_val_def non_empty_preI sameValD)
apply (metis (hide_lams, mono_tags) elimination_freeI2
empty_then_not_val_def non_empty_preI sameValD sameValI1 sameValI2)
apply (drule_tac a="b" in elimination_freeD)
apply (erule exE)
apply (erule conjE)
apply (erule disjE)
apply (rule_tac a="b" in elimination_freeI2)
apply (metis sameValI1)
apply (metis empty_then_not_val_def non_empty_preI pop_once_def sameValD
val_order_def)
apply (metis (full_types) elimination_freeI1 empty_then_not_val_def
non_empty_preI pop_once_def sameValD sameVal_commutative val_order_def)
by assumption

lemma empty_pop_then_ef_pop : "isPop (elimination_free_val (non_empty_pre
pre emp) val) a ⟹
  isPop (elimination_free_val pre val) a"
apply (drule isPopD)
apply (erule exE)
apply (rule_tac a="b" in isPopI)
apply (drule elimination_free_valD)
apply (erule conjE)+
apply (rule elimination_free_valI)
apply (metis elimination_freeD elimination_freeI1 elimination_freeI2
non_empty_pre_then_pre sameVal_commutative)
apply (metis elimination_freeD elimination_freeI1 elimination_freeI2
non_empty_pre_then_pre sameVal_commutative)
by metis

lemma ef_ins_then_ins : "(ia, ib)
       ∈ ins_order (elimination_free_pre (non_empty_pre pre emp) val)
           (elimination_free_val (non_empty_pre pre emp) val) ir ⟹ (ia,
ib) ∈ ins_order pre val ir"
apply (drule ins_orderD)
apply (erule conjE)+
apply (erule disjE)
apply (metis ef_val_then_val elimination_free_preD ins_orderI1 isPushD
isPushI non_empty_pre_then_pre)
by (metis ef_val_then_val elimination_free_preD ins_orderI2 isPushD isPushI
non_empty_pre_then_pre)

lemma ef_rem_then_rem : "(ra, rb)
       ∈ rem_order (elimination_free_pre (non_empty_pre pre emp) val)
           (elimination_free_val (non_empty_pre pre emp) val) ir ⟹ (ra,
rb) ∈ rem_order pre val ir"
apply (drule rem_orderD)
apply (erule conjE)+
apply (erule disjE)
apply (metis ef_val_then_val elimination_free_preD isPopD isPopI
non_empty_pre_then_pre rem_orderI1)
by (metis ef_val_then_val isPopD isPopI rem_orderI2)


locale sequential_stack = sequential_set +
  assumes def_sequential_lifo : "sequential_lifo pre val"

lemma sequential_stackD : "sequential_stack pre val f ⟹ sequential_set
pre val f ∧ sequential_lifo pre val"
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
by metis

lemma sequential_stackI : "sequential_set pre val f ⟹ sequential_lifo
pre val ⟹ sequential_stack pre val f"
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
by metis


locale sequential_empty_set = sequential_empty_history +
    assumes def_correct_empty : "correct_empty pre val emp"

lemma sequential_empty_setD : "sequential_empty_set pre val emp f ⟹
sequential_empty_history pre val emp f ∧
correct_empty pre val emp"
apply (unfold sequential_empty_set_def)
apply (unfold sequential_empty_set_axioms_def)
by metis

lemma sequential_empty_setI : "sequential_empty_history pre val emp f ⟹
correct_empty pre val emp ⟹
      sequential_empty_set pre val emp f"
apply (unfold sequential_empty_set_def)
apply (unfold sequential_empty_set_axioms_def)
by metis


locale sequential_empty_stack = sequential_empty_history +
  assumes def_sequential_lifo : "sequential_lifo pre val"
  assumes def_correct_empty : "correct_empty pre val emp"

lemma sequential_empty_stackD : "sequential_empty_stack pre val emp f ⟹
sequential_empty_history pre val emp f ∧
sequential_lifo pre val ∧
```

```
correct_empty pre val emp"
apply (unfold sequential_empty_stack_def)
apply (unfold sequential_empty_stack_axioms_def)
by metis

lemma sequential_empty_stackI : "sequential_empty_history pre val emp f
⟹
sequential_lifo pre val ⟹
correct_empty pre val emp ⟹
sequential_empty_stack pre val emp f"
apply (unfold sequential_empty_stack_def)
apply (unfold sequential_empty_stack_axioms_def)
by metis


definition linearizable_set :: "'a rel ⇒ 'a rel ⇒ ('a ⇒ bool) ⇒ bool"
where
  "linearizable_set pre val emp ⟷ conc_history pre ∧ (∃ prt f . pre ⊆
prt ∧ sequential_empty_set prt val emp f)"

lemma linearizable_setD : "linearizable_set pre val emp ⟹ conc_history
pre ∧(∃ prt f . pre ⊆ prt ∧ sequential_empty_set prt val emp f)"
by (smt linearizable_set_def)

lemma linearizable_setI : "conc_history pre ⟹  pre ⊆ prt ⟹
sequential_empty_set prt val emp f ⟹ linearizable_set pre val emp"
by (metis linearizable_set_def)

definition linearizable_non_empty_stack :: "'a rel ⇒ 'a rel ⇒ bool"
where
  "linearizable_non_empty_stack pre val ⟷ conc_history pre ∧ (∃ prt f .
pre ⊆ prt ∧ sequential_stack prt val f)"

lemma linearizable_non_empty_stackI : "conc_history pre ⟹ pre ⊆ prt ⟹
sequential_stack prt val f ⟹ linearizable_non_empty_stack pre val"
by (metis linearizable_non_empty_stack_def)

lemma linearizable_non_empty_stackD : "linearizable_non_empty_stack pre
val ⟹ conc_history pre ∧ (∃ prt f . pre ⊆ prt ∧ sequential_stack prt
val f)"
by (smt linearizable_non_empty_stack_def)

definition linearizable_stack :: "'a rel ⇒ 'a rel ⇒ ('a ⇒ bool) ⇒ bool"
where
  "linearizable_stack pre val emp ⟷ conc_history pre ∧ (∃ prt f. pre ⊆
prt ∧ sequential_empty_stack prt val emp f)"

lemma linearizable_stackD : "linearizable_stack pre val emp ⟹
conc_history pre ∧ (∃ prt f. pre ⊆ prt ∧ sequential_empty_stack prt val
emp f)"
by (smt linearizable_stack_def)

lemma linearizable_stackI : "conc_history pre ⟹ pre ⊆ prt ⟹
sequential_empty_stack prt val emp f ⟹ linearizable_stack pre val emp"
by (metis linearizable_stack_def)

lemma linearizable_set_then_wellformed_empty : "linearizable_set pre val
emp ⟹ ∃ f . wellformed_empty_history pre val emp f"
```

apply *(drule linearizable_setD)*
apply *(erule conjE)*
apply *(erule exE)+*
apply *(erule conjE)*
apply *(drule sequential_empty_setD)*
apply *(erule conjE)*
apply *(drule sequential_empty_historyD)*
apply *(erule conjE)*
apply *(rule_tac x="f"* **in** *exI)*
apply *(rule wellformed_empty_historyI)*
apply *(metis conc_historyD)*
apply *(metis conc_history.pre_antisymmetric)*
apply *(metis conc_history.pre_irreflexive)*
apply *(metis wellformed_empty_history.operations_countable)*
apply *(metis wellformed_empty_historyD)*
apply *(metis wellformed_empty_history.def_push_once)*
apply *(metis wellformed_empty_history.def_pop_once)*
apply *(metis wellformed_empty_history.def_val_order)*
apply *(metis wellformed_empty_history.def_all_popped_or_empty)*
apply *(metis wellformed_empty_history.def_empty_then_not_val)*
by *(metis val_then_not_pr_mono wellformed_empty_history.def_val_then_not_pr)*

lemma *wellformed_empty_then_wellformed : "wellformed_empty_history pre val emp f* $\Longrightarrow$ *wellformed_history (non_empty_pre pre emp) val f"*
apply *(drule wellformed_empty_historyD)*
apply *(erule conjE)+*
apply *(rule wellformed_historyI)*
apply *(metis (lifting, no_types) non_empty_preI non_empty_pre_then_pre trans_def)*
apply *(metis (lifting, no_types) antisymI irrefl_def non_empty_pre_then_pre trans_def)*
apply *(metis irrefl_def non_empty_preD)*
apply *metis*
apply *metis*
apply *metis*
apply *metis*
apply *metis*
apply *(metis (hide_lams, no_types) all_popped_or_empty_def non_empty_pre_then_pre)*
by *(metis non_empty_pre_then_pre val_then_not_pr_def)*

lemma *linearizable_set_then_wellformed : "linearizable_set pre val emp* $\Longrightarrow$ $\exists$ *f .wellformed_history (non_empty_pre pre emp) val f"*
by *(metis linearizable_set_then_wellformed_empty wellformed_empty_then_wellformed)*

lemma *wellformed_then_ef_wellformed : "wellformed_history pre val f* $\Longrightarrow$ *wellformed_history (elimination_free_pre pre val) (elimination_free_val pre val) f"*
apply *(drule wellformed_historyD)*
apply *(erule conjE)+*
apply *(rule wellformed_historyI)*
apply *(metis (no_types, hide_lams) elimination_free_preD elimination_free_preI transE transI)*
apply *(metis (poly_guards_query) antisym_def elimination_free_preD)*
apply *(metis elimination_free_preD irrefl_def)*
apply *metis*

apply *(metis ef_finite)*
apply *(metis elimination_free_valD push_once_def)*
apply *(metis (full_types) elimination_free_valD pop_once_def)*
apply *(metis (full_types) elimination_free_valD val_order_def)*
apply *(rule allI)+*
apply *(rule impI)*
apply *(erule_tac x="a"* **in** *allE)*
apply *(erule_tac x="b"* **in** *allE)*
apply *(frule elimination_free_preD)*
apply *(erule conjE)+*
apply *(erule impE)*
apply *(assumption)*
apply *(erule conjE)*
apply *(erule exE)+*
apply *(erule disjE)+*
apply *(rule conjI)*
apply *(rule_tac x="c"* **in** *exI)*
apply *(metis val_then_ef_val)*
apply *(rule_tac x="ca"* **in** *exI)*
apply *(metis val_then_ef_val)*
apply *(rule conjI)*
apply *(rule_tac x="c"* **in** *exI)*
apply *(metis val_then_ef_val)*
apply *(rule_tac x="ca"* **in** *exI)*
apply *(metis (no_types, hide_lams) elimination_freeD elimination_freeI1 push_once_def sameValD sameValI1 val_order_def val_then_ef_val val_then_not_pr_def)*
apply *(rule conjI)*
apply *(rule_tac x="c"* **in** *exI)*
apply *(metis (no_types, hide_lams) elimination_freeD elimination_freeI1 push_once_def sameValD sameValI1 val_order_def val_then_ef_val val_then_not_pr_def)*
apply *(rule_tac x="ca"* **in** *exI)*
apply *(erule disjE)*
apply *(metis val_then_ef_val)*
apply *(metis (no_types, hide_lams) elimination_freeD elimination_freeI1 push_once_def sameValD sameValI1 val_order_def val_then_ef_val val_then_not_pr_def)*
by *(metis elimination_free_preD elimination_free_valD val_then_not_pr_def)*

lemma *ef_ir_in_ir : "elimination_free_ir (non_empty_pre pre emp) val ir* $\subseteq$ *ir"*
by *(metis elimination_free_irD subrelI)*

lemma *help_me : "pre Un (elimination_free_ir (non_empty_pre pre emp) val ir)* $\subseteq$ *pre Un ir"*
by *(metis ef_ir_in_ir order_refl sup.mono)*

lemma *ef_val_then_pre : "wellformed_history (non_empty_pre pre emp) val f* $\Longrightarrow$ *(ia, ra)* $\in$ *elimination_free_val (non_empty_pre pre emp) val* $\Longrightarrow$ *(ia, ra)* $\in$ *pre"*
apply *(drule elimination_free_valD)*
apply *(erule conjE)+*
apply *(drule elimination_freeD)+*
apply *(erule exE)+*
apply *(erule conjE)+*
apply *(erule disjE)+*
apply *(drule sameValD)+*

apply *(metis (erased, hide_lams) val_order_def val_then_not_pr_def wellformed_history.def_val_then_not_pr wellformed_history.def_value)*
apply *(drule sameValD)+*
apply *(metis (erased, hide_lams) non_empty_pre_then_pre push_once_def val_order_def wellformed_history.def_push_once wellformed_history.def_value)*
apply *(drule sameValD)+*
by *(metis (erased, hide_lams) val_order_def val_then_not_pr_def wellformed_history.def_val_then_not_pr wellformed_history.def_value)*

lemma *elim_ef_rem_then_rem: "(ra, rb) ∈ rem_order (elimination_free_pre pre val) (elimination_free_val pre val) (elimination_free_ir pre val ir) ⟹*
  wellformed_history pre val f ⟹
  (ra, rb) ∈ rem_order pre val ir"
apply *(drule rem_orderD)*
apply *(erule conjE)+*
apply *(erule disjE)*
apply *(rule rem_orderI1)*
apply *(metis elimination_free_valD isPop_def)*
apply *(metis elimination_free_valD isPop_def)*
apply *(metis elimination_free_preD)*
apply *(erule exE)*
apply *(erule conjE)*
apply *(rule_tac c="c"* **in** *rem_orderI2)*
apply *(metis elimination_free_valD isPop_def)*
apply *(metis elimination_free_valD isPop_def)*
apply *(metis elimination_free_irD)*
by *(metis elimination_free_irD)*

lemma *empty_rem_then_rem : "(ra, rb) ∈ rem_order (non_empty_pre pre emp) val ir ⟹ wellformed_history (non_empty_pre pre emp) val f ⟹ (ra, rb) ∈ rem_order pre val ir"*
by *(metis (erased, hide_lams) non_empty_preD rem_orderD rem_orderI1 rem_orderI2)*

end

A.3 Dealing with Pop-Empty Operations

theory *empty_stack*
imports *Main concurrent_histories*

begin
locale *concurrent_empty_stack = wellformed_empty_history +*
 fixes *slin :: "'a rel"*
 fixes *elin :: "'a rel"*
 assumes *stack_linearizable: "sequential_stack slin val f"*
 assumes *pr_in_slin : "(a, b) ∈ pre ⟹ ¬ emp a ⟹ ¬ emp b ⟹ (a, b) ∈ slin"*
 assumes *set_linearizable: "sequential_empty_set elin val emp f"*
 assumes *pr_in_elin : "(a, b) ∈ pre ⟹ (a, b) ∈ elin"*

context *wellformed_empty_history*
begin

lemma *concurrent_empty_stackI :*

```
"wellformed_empty_history pre val emp f ⟹
 sequential_stack slin val f ⟹
 ∀ a b . (a, b) ∈ pre ⟶ ¬ emp a ⟶ ¬ emp b ⟶ (a, b) ∈ slin ⟹
 sequential_empty_set elin val emp f ⟹
 ∀ a b . (a, b) ∈ pre ⟶ (a, b) ∈ elin ⟹
 concurrent_empty_stack pre val emp f slin elin"
apply (unfold concurrent_empty_stack_def)
apply (rule conjI)
apply (assumption)
apply (unfold concurrent_empty_stack_axioms_def)
apply (rule conjI)+
apply (assumption)+
apply (rule conjI)
by (assumption)+

end

context concurrent_empty_stack
begin

lemma slin_transitive : "trans slin"
apply (insert stack_linearizable)
apply (unfold sequential_stack_def)
apply (erule  conjE)
apply (unfold sequential_set_def)
apply (erule  conjE)
apply (unfold wellformed_history_def)
apply (erule  conjE)
by metis

lemma slin_irreflexive : "irrefl slin"
apply (insert stack_linearizable)
apply (unfold sequential_stack_def)
apply (erule  conjE)
apply (unfold sequential_set_def)
apply (erule  conjE)
apply (unfold wellformed_history_def)
apply (erule  conjE)
by metis

lemma slin_antisymmetric : "antisym slin"
apply (insert stack_linearizable)
apply (unfold sequential_stack_def)
apply (erule  conjE)
apply (unfold sequential_set_def)
apply (erule  conjE)
apply (unfold wellformed_history_def)
apply (erule  conjE)

by metis

lemma slin_total : "¬ emp a ⟹ ¬ emp b ⟹ a=b ∨ (a, b) ∈ slin ∨ (b,
a) ∈ slin"
apply (insert stack_linearizable)
apply (unfold sequential_stack_def)
apply (erule  conjE)
apply (unfold sequential_set_def)
apply (erule  conjE)
apply (unfold sequential_set_axioms_def)
by (metis is_pop_def is_push_def lem_empty_or_push_or_pop
wellformed_history.lem_val_then_push_and_pop)

lemma val_then_slin : "(a, b) ∈ val ⟹ (a, b) ∈ slin"
apply (insert stack_linearizable)
apply (unfold sequential_stack_def)
apply (erule conjE)
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold wellformed_history_def)
apply (erule conjE)
by (metis (hide_lams, no_types) def_empty_then_not_val
empty_then_not_val_def slin_total val_order_def val_then_not_pr_def)

lemma slin_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈ slin
⟹ (ib, ra) ∈ slin ⟹ (rb, ra) ∈ slin"
apply (insert stack_linearizable)
apply (unfold sequential_stack_def)
apply (erule conjE)
apply (unfold sequential_stack_axioms_def)
apply (unfold sequential_lifo_def)
by metis

lemma elin_transitive : "trans elin"
apply (insert set_linearizable)
apply (unfold sequential_empty_set_def)
apply (erule  conjE)
apply (unfold sequential_empty_history_def)
apply (erule  conjE)
by (metis wellformed_empty_history.pre_transitive)


lemma elin_irreflexive : "irrefl elin"
apply (insert set_linearizable)
apply (unfold sequential_empty_set_def)
apply (erule  conjE)
apply (unfold sequential_empty_history_def)
apply (erule  conjE)
by (metis wellformed_empty_history.pre_irreflexive)

lemma elin_antisymmetric : "antisym elin"
apply (insert set_linearizable)
apply (unfold sequential_empty_set_def)
apply (erule  conjE)
apply (unfold sequential_empty_history_def)
apply (erule  conjE)
by (metis wellformed_empty_history.pre_antisymmetric)

lemma elin_total : "a=b ∨ (a, b) ∈ elin ∨ (b, a) ∈ elin"
apply (insert set_linearizable)
apply (unfold sequential_empty_set_def)
apply (erule  conjE)
apply (unfold sequential_empty_history_def)
apply (erule  conjE)
apply (unfold sequential_empty_history_axioms_def)
by (metis UNIV_I total_on_def)
```

```
lemma elin_correct_empty : "emp a ⟹ (ib, rb) ∈ val ⟹
   ((ib, a) ∈ elin ∧ (rb, a) ∈ elin) ∨ ((a, ib) ∈ elin ∧ (a, rb) ∈
elin)"
apply (insert set_linearizable)
apply (unfold sequential_empty_set_def)
apply (erule  conjE)
apply (unfold sequential_empty_set_axioms_def)
apply (unfold correct_empty_def)
by metis

definition w1 :: "'a rel"
  where "w1 = {(a, b) . ¬ emp a ∧ ¬ emp b ∧ (∃ c . emp c ∧ (a, c) ∈ elin
∧ (c, b) ∈ elin)}"

lemma w1D : "(a, b) ∈ w1 ⟹ ¬ emp a ∧ ¬ emp b ∧ (∃ c . emp c ∧ (a, c)
∈ elin ∧ (c, b) ∈ elin)"
apply (unfold w1_def)
by (simp add: set_eq_iff)

lemma w1I : "¬ emp a ⟹ ¬ emp b ⟹ emp c ⟹ (a, c) ∈ elin ⟹ (c, b)
∈ elin ⟹ (a, b) ∈ w1"
apply (unfold w1_def)
apply (simp add: set_eq_iff)
by metis

lemma w1_transitive : "trans w1"
by (metis elin_transitive trans_def w1D w1I)

lemma w1_antisymmetric : "antisym w1"
by (metis antisym_def elin_antisymmetric transE w1D w1_transitive)

lemma w1_irreflexive : "irrefl w1"
by (metis antisymD elin_antisymmetric irrefl_def w1D)

lemma w1_then_not_inv_pr : "(a, b) ∈ w1 ⟹ ¬ (b, a) ∈ pre"
by (metis antisymD elin_antisymmetric elin_transitive pr_in_elin transD
w1D)

lemma not_w1_then_same_with_empty : "(a, b) ∉ w1 ⟹ (b, a) ∉ w1 ⟹ ¬
emp a ⟹ ¬ emp b ⟹ emp c ⟹ (a, c) ∈ elin ⟹ (b, c) ∈ elin"
apply (insert elin_total)
apply (drule_tac x="b" in meta_spec)
apply (drule_tac x="c" in meta_spec)
apply (erule disjE)
apply metis
apply (erule disjE)
apply metis
by (metis w1I)

lemma val_then_not_w1 : "(a, b) ∈ val ⟹ (a, b) ∉ w1 ∧ (b, a) ∉ w1"
apply (rule conjI)
apply (rule notI)
apply (drule w1D)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)+
apply (metis antisymD elin_antisymmetric elin_correct_empty)
by (metis antisym_def elin_antisymmetric elin_correct_empty w1D)
```

```
lemma val_then_same_w11 : "(ia, ra) ∈ val ⟹ (b, ia) ∈ w1 ⟹ (b, ra) ∈
w1"
by (smt def_empty_then_not_val elin_correct_empty empty_then_not_val_def
val_then_not_w1 w1D w1I)

lemma val_then_same_w12 : "(ia, ra) ∈ val ⟹ (b, ra) ∈ w1 ⟹ (b, ia) ∈
w1"
by (smt def_empty_then_not_val elin_correct_empty empty_then_not_val_def
val_then_not_w1 w1D w1I)

lemma val_then_same_w13 : "(ia, ra) ∈ val ⟹ (ia, b) ∈ w1 ⟹ (ra, b) ∈
w1"
by (smt def_empty_then_not_val elin_correct_empty empty_then_not_val_def
val_then_not_w1 w1D w1I)

lemma val_then_same_w14 : "(ia, ra) ∈ val ⟹ (ra, b) ∈ w1 ⟹ (ia, b) ∈
w1"
by (smt def_empty_then_not_val elin_correct_empty empty_then_not_val_def
val_then_not_w1 w1D w1I)

definition witness :: "'a rel"
where "witness = {(a, b) . (emp a ∧ (a, b) ∈ elin) ∨
                            (emp b ∧ (a, b) ∈ elin) ∨
                            (a, b) ∈ w1 ∨
                            (¬ emp a ∧ ¬ emp b ∧ (a, b) ∉ w1 ∧ (b, a) ∉
w1 ∧ (a, b) ∈ slin)}"

lemma witnessD : "(a, b) ∈ witness ⟹ (emp a ∧ (a, b) ∈ elin) ∨
                            (emp b ∧ (a, b) ∈ elin) ∨
                            (a, b) ∈ w1 ∨
                            (¬ emp a ∧ ¬ emp b ∧ (a, b) ∉ w1 ∧ (b, a) ∉
w1 ∧ (a, b) ∈ slin)"
apply (unfold witness_def)
by (simp add: set_eq_iff)

lemma witnessI1 : "emp a ⟹ (a, b) ∈ elin ⟹ (a, b) ∈ witness"
apply (unfold witness_def)
by (simp add: set_eq_iff)

lemma witnessI2 : "emp b ⟹ (a, b) ∈ elin ⟹ (a, b) ∈ witness"
apply (unfold witness_def)
by (simp add: set_eq_iff)

lemma witnessI3 : "(a, b) ∈ w1 ⟹ (a, b) ∈ witness"
apply (unfold witness_def)
by (simp add: set_eq_iff)

lemma witnessI4 : "¬ emp a ⟹ ¬ emp b ⟹ (a, b) ∉ w1 ⟹ (b, a) ∉ w1
⟹ (a, b) ∈ slin ⟹ (a, b) ∈ witness"
apply (unfold witness_def)
by (simp add: set_eq_iff)

lemma pr_in_witness : "pre ⊆ witness"
apply (rule subrelI)
apply (case_tac "emp x")
apply (metis pr_in_elin witnessI1)
apply (case_tac "emp y")
apply (metis pr_in_elin witnessI2)
apply (case_tac "(x, y) ∈ w1")
apply (rule witnessI3)
apply (assumption)
by (metis pr_in_slin w1_then_not_inv_pr witnessI4)

lemma witness_transitive : "trans witness"
apply (rule transI)
apply (drule witnessD)+
apply (erule disjE)+
apply (metis elin_transitive transE witnessI1)
apply (erule disjE)
apply (metis elin_transitive transD witnessI1)
apply (erule disjE)
apply (metis elin_transitive trans_def w1D witnessI1)
apply (erule conjE)+
apply (metis elin_total w1I witnessI1)
apply (erule disjE)+
apply (metis elin_transitive transE w1I witnessI1 witnessI2 witnessI3)
apply (erule disjE)+
apply (metis w1D)
apply metis
apply (erule disjE)
apply (metis elin_transitive trans_def w1D witnessI2)
apply (erule disjE)
apply (metis antisym_def elin_total w1D w1I w1_antisymmetric witnessI2)
apply (erule disjE)+
apply (metis transD w1_transitive witnessI3)
apply (metis not_w1_then_same_with_empty transE w1D w1I w1_transitive
witnessI3)
apply (erule disjE)
apply (metis not_w1_then_same_with_empty w1D w1I witnessI3)
by (metis not_w1_then_same_with_empty slin_transitive transD w1D w1I
witnessI4)

lemma witness_irreflexive : "irrefl witness"
by (metis elin_irreflexive irrefl_def slin_irreflexive w1_irreflexive
witnessD)

lemma witness_antisymmetric : "antisym witness"
by (metis antisymI lem_irreflexive_then_not_equal trans_def
witness_irreflexive witness_transitive)

lemma witness_total : "total witness"
by (metis elin_total slin_total total_on_def witnessI1 witnessI2 witnessI3
witnessI4)

lemma witness_correct_empty : "emp a ⟹ (ib, rb) ∈ val ⟹
   ((ib, a) ∈ witness ∧ (rb, a) ∈ witness) ∨ ((a, ib) ∈ witness ∧ (a, rb)
∈ witness)"
by (metis elin_correct_empty witnessI1 witnessI2)

lemma val_then_not_witness : "(ia, ra) ∈ val ⟹ (ra, ia) ∉ witness"
by (metis antisym_def def_empty_then_not_val def_val_order
empty_then_not_val_def slin_antisymmetric val_order_def val_then_not_w1
val_then_slin witnessD)
```

```
lemma witness_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈
witness ⟹ (ib, ra) ∈ witness ⟹ (rb, ra) ∈ witness"
apply (drule witnessD)+
apply (erule disjE)+
apply (metis def_empty_then_not_val empty_then_not_val_def)
apply (erule disjE)
apply (metis def_empty_then_not_val empty_then_not_val_def)
apply (erule disjE)
apply (metis val_then_same_w12 w1D)
apply (metis elin_correct_empty elin_irreflexive irrefl_def)
apply (erule disjE)+
apply (metis def_empty_then_not_val empty_then_not_val_def)
apply (erule disjE)
apply (metis w1D)
apply metis
apply (erule disjE)+
apply (metis def_empty_then_not_val empty_then_not_val_def)
apply (erule disjE)
apply (metis w1D)
apply metis
apply (erule disjE)+
apply (metis val_then_same_w13 w1D)
apply (metis elin_correct_empty elin_irreflexive irrefl_def)
apply (erule disjE)+
apply (metis val_then_same_w13 witnessI3)
apply (metis val_then_same_w13)
apply (erule disjE)
apply (metis val_then_same_w12)
by (metis def_empty_then_not_val empty_then_not_val_def slin_ooo
val_then_same_w12 val_then_same_w14 witnessI4)

lemma witness_sequential_empty_stack : "sequential_empty_stack witness val
emp f"
apply (rule sequential_empty_stackI)
apply (rule sequential_empty_historyI)
apply (rule wellformed_empty_historyI)
apply (metis witness_transitive)
apply (metis witness_antisymmetric)
apply (metis witness_irreflexive)
apply (metis operations_countable)
apply (metis operations_finite)
apply (metis def_push_once)
apply (metis def_pop_once)
apply (metis def_val_order)
apply (metis def_all_popped_or_empty)
apply (metis def_empty_then_not_val)
apply (metis val_then_not_prI val_then_not_witness)
apply (metis witness_total)
apply (smt sequential_lifo_def witness_ooo)
by (smt correct_empty_def witness_correct_empty)

theorem concurrent_empty_stack_linearizable : "∃ prt .( pre ⊆ prt ∧
sequential_empty_stack prt val emp f)"
apply (rule_tac x="witness" in exI)
apply (rule conjI)
apply (rule pr_in_witness)
by (rule witness_sequential_empty_stack)
```

```
end

lemma stack_set_then_set: "linearizable_non_empty_stack (non_empty_pre pre
emp) val ⟹ linearizable_set pre val emp ⟹
  ∃ elin slin f . concurrent_empty_stack pre val emp f slin elin"
apply (unfold linearizable_non_empty_stack_def)
apply (erule conjE)+
apply (erule exE)+
apply (rename_tac slin f)
apply (erule conjE)
apply (unfold linearizable_set_def)
apply (erule conjE)+
apply (erule exE)+
apply (rename_tac elin fa)
apply (erule conjE)
apply (rule_tac x="elin" in exI)
apply (rule_tac x="slin" in exI)
apply (rule_tac x="f" in exI)
apply (frule sequential_empty_setD)
apply (erule conjE)
apply (frule sequential_empty_historyD)
apply (erule conjE)
apply (frule wellformed_empty_historyD)
apply (erule conjE)+
apply (rule wellformed_empty_history.concurrent_empty_stackI)
apply (rule wellformed_empty_historyI)
apply (metis conc_historyD)
apply (metis conc_historyD)
apply (metis conc_historyD)
apply (metis sequential_setD sequential_stackD wellformed_historyD)
apply (assumption)+
apply (metis set_rev_mp val_then_not_prI val_then_not_pr_def)
apply (rule wellformed_empty_historyI)
apply (metis conc_historyD)
apply (metis conc_historyD)
apply (metis conc_historyD)
apply (metis sequential_setD sequential_stackD wellformed_historyD)
apply (assumption)+
apply (metis in_mono val_then_not_pr_def)
apply (assumption)+
apply (rule allI)+
apply (rule impI)+
apply (frule_tac a="a" and b="b" and pre="pre" and emp="emp" in
non_empty_preI)
apply (assumption)+
apply (metis in_mono)
apply (rule sequential_empty_setI)
apply (rule sequential_empty_historyI)
apply (rule wellformed_empty_historyI)
apply (drule wellformed_empty_historyD)
apply (drule sequential_empty_setD)
apply (drule sequential_empty_historyD)
apply (erule conjE)+
apply (assumption)+
apply (metis sequential_setD sequential_stackD wellformed_historyD)
apply (assumption)+
by (metis set_rev_mp)
```

```
theorem concurrent_empty_stack_linearizable : "linearizable_non_empty_stack
(non_empty_pre pre emp) val ⟹ linearizable_set pre val emp
 ⟹ linearizable_stack pre val emp"
apply (drule stack_set_then_set)
apply (assumption)
apply (erule exE)
apply (erule exE)
by (metis concurrent_empty_stack.concurrent_empty_stack_linearizable
linearizable_set_def linearizable_stack_def)

end
```

A.4 Dealing with Elimination

```
theory elimination_stack
imports Main concurrent_histories
begin

context wellformed_history
begin

end

locale elimination_stack = wellformed_history +
  fixes lin :: "'a rel"
  assumes ef_linearizable_def: "sequential_stack lin (elimination_free_val
pre val) f"
  assumes pr_in_lin : "(a, b) ∈ elimination_free_pre pre val ⟹ (a, b) ∈
lin"

  assumes finite_set_def : "finite (elimination_free pre val)"
  assumes pre_abcd : "abcd pre"

lemma elimination_stackI : "wellformed_history pre val f ⟹
  sequential_stack lin (elimination_free_val pre val) f ⟹
  ∀ a b . (a, b) ∈ elimination_free_pre pre val ⟶ (a, b) ∈ lin ⟹
  finite (elimination_free pre val) ⟹
  abcd pre ⟹
  elimination_stack pre val f lin"
apply (unfold elimination_stack_def)
apply (rule conjI)
apply (assumption)
apply (unfold elimination_stack_axioms_def)
apply (rule conjI)+
apply (assumption)+
apply (rule conjI)
by (assumption)+

context elimination_stack
begin

definition same_val :: "'a rel"
  where "same_val = {(a, b) . (a, b) ∈ val ∨ (b, a) ∈ val}"

lemma same_valD : "(a, b) ∈ same_val ⟹ (a, b) ∈ val ∨ (b, a) ∈ val"
apply (unfold same_val_def)
```

```
by (simp add : set_eq_iff)

lemma same_valI1 : "(a, b) ∈ val ⟹ (a, b) ∈ same_val"
apply (unfold same_val_def)
by (simp add : set_eq_iff)

lemma same_valI2 : "(a, b) ∈ val ⟹ (b, a) ∈ same_val"
apply (unfold same_val_def)
by (simp add : set_eq_iff)

lemma always_same_val1 : "(a, b) ∈ pre ⟹  ∃ c . (a, c) ∈ same_val"
by (metis def_push_or_pop same_valI1 same_valI2)

lemma always_same_val2 : "(a, b) ∈ pre ⟹  ∃ c . (b, c) ∈ same_val"
by (metis def_push_or_pop same_valI1 same_valI2)

lemma not_same_valD : "(a, b) ∈ pre ⟹ (a, b) ∉ same_val ⟹ a = b ∨
    (∃ ra rb . (a, ra) ∈ same_val ∧ (b, rb) ∈ same_val ∧ a ≠ b ∧  a ≠
rb ∧ ra ≠ b ∧ ra ≠ rb)"
by (metis always_same_val1 always_same_val2 def_pop_once def_push_once
def_value pop_once_def push_once_def same_valD same_valI1 same_valI2
val_order_def)

lemma same_val_commutative : "(a, b) ∈ same_val ⟹ (b, a) ∈ same_val"
by (metis same_valD same_valI1 same_valI2)

lemma same_val_unique : "(a, b) ∈ same_val ⟹ (a, c) ∈ same_val ⟹ b =
c"
apply (drule same_valD)+
apply (erule disjE)+
apply (metis def_pop_once pop_once_def)
apply (metis def_value val_order_def)
apply (erule disjE)
apply (metis def_value val_order_def)
by (metis def_push_once push_once_def)

definition ef :: "'a set"
where
  "ef = {a . (∃ b . (a, b) ∈ same_val ∧ ((a, b) ∈ pre ∨ (b, a) ∈ pre))}"

lemma efD : "a ∈ ef ⟹ ∃ b . (a, b) ∈ same_val ∧ ((a, b) ∈ pre ∨ (b,
a) ∈ pre)"
apply (unfold ef_def)
by (simp add: set_eq_iff)

lemma efI1 : "(a, b) ∈ same_val ⟹ (a, b) ∈ pre ⟹ a ∈ ef"
apply (unfold ef_def)
apply (simp add: set_eq_iff)
by metis

lemma efI2 : "(a, b) ∈ same_val ⟹ (a, b) ∈ pre ⟹ b ∈ ef"
apply (unfold ef_def)
apply (simp add: set_eq_iff)
by (metis same_valD same_valI1 same_valI2)

lemma ef_val_then_ef : "a ∈ ef ⟹ (a, b) ∈ same_val ⟹ b ∈ ef"
by (metis efD efI1 efI2 same_val_commutative same_val_unique)
```

```
lemma ef_then_push_or_pop : "a ∈ ef ⟹ is_push a ∨ is_pop a"
by (metis efD is_popI is_pushI same_valD)

definition el :: "'a set"
  where "el = {a . (is_push a ∨ is_pop a) ∧ a ∉ ef}"

lemma elD : "a ∈ el ⟹ ∃ b . (a, b) ∈ same_val ∧ (a, b) ∉ pre ∧ (b, a)
∉ pre"
apply (unfold el_def)
apply (simp add:set_eq_iff)
by (metis efI1 efI2 is_popD lem_push_then_pop_exists same_valI1 same_valI2)

lemma elI1 : "(a, b) ∈ same_val ⟹ (a, b) ∉ pre ⟹ (b, a) ∉ pre ⟹ a
∈ el"
by (metis (lifting, no_types) efD el_def lem_val_then_push_and_pop
mem_Collect_eq same_valD same_val_unique)

lemma elI2 : "(a, b) ∈ same_val ⟹ (a, b) ∉ pre ⟹ (b, a) ∉ pre ⟹ b
∈ el"
by (metis elI1 same_val_commutative)

lemma elI3 : "a ∈ el ⟹ (a, b) ∈ same_val ⟹ b ∈ el"
by (metis elD elI2 same_val_unique)

lemma el_or_ef1 : "(a, b) ∈ pre ⟹ a ∈ el ∨ a ∈ ef"
by (metis always_same_val1 efI1 efI2 elI1 same_val_commutative)

lemma el_or_ef2 : "(a, b) ∈ pre ⟹ b ∈ el ∨ b ∈ ef"
by (metis (lifting) el_def lem_push_or_pop2 mem_Collect_eq)

lemma push_then_el_or_ef : "is_push a ⟹ a ∈ el ∨ a ∈ ef"
by (metis (lifting) el_def mem_Collect_eq)

lemma pop_then_el_or_ef : "is_pop a ⟹ a ∈ el ∨ a ∈ ef"
by (metis (lifting) el_def mem_Collect_eq)

lemma el_then_not_ef : "a ∈ el ⟹ a ∉ ef"
by (metis (lifting, no_types) el_def mem_Collect_eq)

lemma ef_then_not_el : "a ∈ ef ⟹ a ∉ el"
by (metis el_then_not_ef)

definition ef_pre :: "'a rel"
where
  "ef_pre = {(a, b) . a ∈ ef ∧ b ∈ ef ∧ (a, b) ∈ pre}"

lemma ef_preD : "(a, b) ∈ ef_pre ⟹ a ∈ ef ∧ b ∈ ef ∧ (a, b) ∈ pre"
apply (unfold ef_pre_def)
by (simp add: set_eq_iff)

lemma ef_preI : "a ∈ ef ⟹ b ∈ ef ⟹ (a, b) ∈ pre ⟹ (a, b) ∈
ef_pre"
apply (unfold ef_pre_def)
by (simp add: set_eq_iff)

definition ef_val :: "'a rel"
where
  "ef_val = {(a, b) . a ∈ ef ∧ b ∈ ef ∧ (a, b) ∈ val}"
```

lemma *ef_valD : "(a, b) ∈ ef_val ⟹ a ∈ ef ∧ b ∈ ef ∧ (a, b) ∈ val"*
apply *(unfold ef_val_def)*
by *(simp add: set_eq_iff)*

lemma *ef_valI : "a ∈ ef ⟹ b ∈ ef ⟹ (a, b) ∈ val ⟹ (a, b) ∈ ef_val"*
apply *(unfold ef_val_def)*
by *(simp add: set_eq_iff)*

lemma *ef_val_then_elimination_free_val : "(a, b) ∈ ef_val ⟹ (a, b) ∈ elimination_free_val pre val"*
by *(metis def_val_then_not_pr ef_valD elI1 el_then_not_ef elimination_freeI1 elimination_freeI2 elimination_free_valI sameValI1 same_valI2 val_then_not_pr_def)*

lemma *elimination_free_val_then_ef_val : "(a, b) ∈ elimination_free_val pre val ⟹ (a, b) ∈ ef_val"*
by *(metis def_pop_once def_val_then_not_pr def_value efI1 efI2 ef_valI elimination_freeD elimination_free_valD pop_once_def sameValD same_valI1 val_order_def val_then_not_pr_def)*

lemma *ef_val_equals_elimination_free_val : "elimination_free_val pre val = ef_val"*
apply *(simp add:set_eq_iff)*
by *(metis ef_val_then_elimination_free_val elimination_free_val_then_ef_val)*

lemma *ef_linearizable: "sequential_stack lin ef_val f"*
by *(metis ef_linearizable_def ef_val_equals_elimination_free_val)*

lemma *finite_set : "finite ef"*
by *(metis ef_def elimination_free_def finite_set_def sameVal_def same_val_def)*

definition *minf :: "'a rel"*
  where *"minf = {(a, b) . ∃ ra rb . (a, ra) ∈ same_val ∧ (b, rb) ∈ same_val ∧ (min (f a) (f ra)) < (min (f b) (f rb))}"*

lemma *minfD : "(a, b) ∈ minf ⟹ ∃ ra rb . (a, ra) ∈ same_val ∧ (b, rb) ∈ same_val ∧ (min (f a) (f ra)) < (min (f b) (f rb))"*
apply *(unfold minf_def)*
by *(simp add : set_eq_iff)*

lemma *minfI : "(a, ra) ∈ same_val ⟹ (b, rb) ∈ same_val ⟹ (min (f a) (f ra)) < (min (f b) (f rb)) ⟹ (a, b) ∈ minf"*
apply *(unfold minf_def)*
apply *(simp add : set_eq_iff)*
by *metis*

lemma *lem_f_orders : "a ≠ b ⟹ f a < f b ∨ f b < f a"*
by *(smt injD operations_countable)*

lemma *lem_min_f_orders : "a ≠ c ⟹ a ≠ d ⟹ b ≠ c ⟹ b ≠ d ⟹*

```
   (min (f a) (f b) ) < (min (f c) (f d)) ∨ (min (f c) (f d) ) < (min (f a)
(f b))"
by (smt injD operations_countable)

lemma always_minf : "(a, ra) ∈ same_val ⟹ (b, rb) ∈ same_val ⟹ a ≠ b
⟹ (a, b) ∉ same_val ⟹ (a, b) ∈ minf ∨ (b, a) ∈ minf"

by (metis lem_min_f_orders minfI same_val_commutative same_val_unique)

lemma not_minf_then_same_val : "(a, b) ∉ minf ⟹ (b, a) ∉ minf ⟹
  is_push a ∨ is_pop a ⟹ is_push b ∨ is_pop b ⟹ (a, b) ∈ same_val ∨
a=b"
by (metis always_minf is_popD is_push_def same_valI1 same_valI2)

lemma minf_irreflexive : "irrefl minf"
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (drule minfD)
apply (erule exE)+
apply (erule conjE)+
by (metis less_not_sym same_val_unique)

lemma minf_antisymmetric : "antisym minf"
by (smt antisymI minfD same_val_unique)

lemma minf_transitive : "trans minf"
apply (rule transI)
apply (drule minfD)+
by (metis (hide_lams, no_types) dual_order.strict_trans minfI
same_val_unique)

lemma same_val_then_not_minf : "(a, b) ∈ same_val ⟹ (a, b) ∉ minf"
by (metis min.commute minfD not_less_iff_gr_or_eq same_val_commutative
same_val_unique)

definition el_pre :: "'a rel"
  where "el_pre = {(a, b) . ∃ c . (b, c) ∈ same_val ∧ ((a, b) ∈ pre ∨
(a, c) ∈ pre)}"

lemma el_preD1 : "(a, b) ∈ el_pre ⟹ ∃ c . (b, c) ∈ same_val ∧ ((a, b)
∈ pre ∨ (a, c) ∈ pre)"
apply (unfold el_pre_def)
by (simp add : set_eq_iff)

lemma el_preD : "(a, b) ∈ el_pre ⟹ (b, c) ∈ same_val ⟹ (a, b) ∈ pre
∨ (a, c) ∈ pre"
apply (unfold el_pre_def)
apply (simp add : set_eq_iff)
by (metis def_pop_once def_push_once def_value pop_once_def push_once_def
same_valD val_order_def)

lemma el_preI1 : "(b, c) ∈ same_val ⟹ (a, b) ∈ pre ∨ (a, c) ∈ pre ⟹
(a, b) ∈ el_pre"
apply (unfold el_pre_def)
apply (simp add : set_eq_iff)
by metis
```

```
lemma el_preI2 : "(b, c) ∈ same_val ⟹ (a, b) ∈ pre ∨ (a, c) ∈ pre ⟹
(a, c) ∈ el_pre"
apply (unfold el_pre_def)
apply (simp add : set_eq_iff)
by (metis same_valD same_valI1 same_valI2)

lemma el_pre_abcd : "abcd el_pre"
apply (rule abcdI)
apply (drule el_preD1)+
apply (erule exE)+
apply (erule conjE)+
by (metis abcdD el_preI1 pre_abcd)

lemma lem_pos_exists : "b ∈ el ⟹
   (∀ a . a ∈ ef ⟶ (a, b) ∉ el_pre) ∨
   (∃ a . a ∈ ef ∧ (a, b) ∈ el_pre ∧ (∀ c . c ∈ ef ∧ (a, c) ∈ lin ⟶
(c, b) ∉ el_pre))"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (insert finite_set)
apply (erule finite_induct)
apply (metis emptyE)
apply (erule disjE)
apply (case_tac "(x, b) ∈ el_pre")
apply (rule disjI2)
apply (rule_tac x="x" in exI)
apply (rule conjI)
apply (metis insertI1)
apply (rule conjI)
apply metis
apply (metis insertE lem_irreflexive_then_not_equal)
apply (metis insertE)
apply (erule exE)
apply (erule conjE)+
apply (rule disjI2)
apply (case_tac "(x, b) ∈ el_pre")
apply (case_tac "(x, a) ∈ lin")
apply (metis insertE insertI2 transE)
apply (case_tac "(a, x) ∈ lin")
apply (metis insertCI insertE lem_irreflexive_then_not_equal transE)
apply (case_tac "a= x")
apply metis
apply (metis insert_iff)
by (metis insert_iff)


definition el_pos :: "'a rel"
  where "el_pos = {(a, b) . a ∈ ef ∧ b ∈ el ∧ (a, b) ∈ el_pre ∧ (∀ c .
c ∈ ef ∧ (a, c) ∈ lin ⟶ (c, b) ∉ el_pre)}"
```

```
lemma el_posD : "(a, b) ∈ el_pos ⟹ a ∈ ef ∧ b ∈ el ∧ (a, b) ∈ el_pre
∧ (∀ c . c ∈ ef ∧ (a, c) ∈ lin ⟶ (c, b) ∉ el_pre)"
apply (unfold el_pos_def)
by (simp add : set_eq_iff)

lemma el_posI : "a ∈ ef ⟹ b ∈ el ⟹ (a, b) ∈ el_pre ⟹ (∀ c . c ∈
ef ∧ (a, c) ∈ lin ⟶ (c, b) ∉ el_pre) ⟹ (a, b) ∈ el_pos"
apply (unfold el_pos_def)
by (simp add : set_eq_iff)

lemma lem_is_push_ef_then_is_ef_push : "is_push a ⟹ a ∈ ef ⟹
wellformed_history.is_push ef_val a"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
by (metis ef_valI ef_val_then_ef is_pushD same_valI1
wellformed_history.is_push_def)

lemma lem_is_pop_ef_then_is_ef_pop : "is_pop a ⟹ a ∈ ef ⟹
wellformed_history.is_pop ef_val a"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
by (metis ef_valI ef_val_then_ef is_popD same_valI2
wellformed_history.is_pop_def)

lemma one_el_pos : "(a, b) ∈ el_pos ⟹ (c, b) ∈ el_pos ⟹ a=c"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (drule el_posD)+
apply (erule conjE)+
apply (frule_tac a="a" in ef_then_push_or_pop)
apply (frule_tac a="c" in ef_then_push_or_pop)
apply (erule disjE)+
apply (case_tac "a=c")
apply metis
apply (metis lem_is_push_ef_then_is_ef_push)
apply (metis lem_is_pop_ef_then_is_ef_pop lem_is_push_ef_then_is_ef_push)
by (metis lem_is_pop_ef_then_is_ef_pop lem_is_push_ef_then_is_ef_push)

lemma not_el_posD : "¬(∃ a . (a, b) ∈ el_pos) ⟹ b ∈ el ⟹ (∀ a . a ∈
ef ⟶ (a, b) ∉ el_pre)"
by (metis el_posI lem_pos_exists)

definition ef_el :: "'a rel"
```

```
  where "ef_el = {(x, y) .
(x ∈ el ∧ y ∈ ef ∧ (∀ a . a ∈ ef ⟶ (a, x) ∉ el_pre) ) ∨
(x ∈ el ∧ y ∈ ef ∧ (∃ a . (a, x) ∈ el_pos ∧ (a, y) ∈ lin)) ∨
(y ∈ el ∧ x ∈ ef ∧ (x, y) ∈ el_pos) ∨
(y ∈ el ∧ x ∈ ef ∧ (∃ a . (a, y) ∈ el_pos ∧ (x, a) ∈ lin))
 }"

lemma ef_elD : "(x, y) ∈ ef_el ⟹ (x ∈ el ∧ y ∈ ef ∧ (∀ a . a ∈ ef
⟶ (a, x) ∉ el_pre) ) ∨
(x ∈ el ∧ y ∈ ef ∧ (∃ a . (a, x) ∈ el_pos ∧ (a, y) ∈ lin)) ∨
(y ∈ el ∧ x ∈ ef ∧ (x, y) ∈ el_pos) ∨
(y ∈ el ∧ x ∈ ef ∧ (∃ a . (a, y) ∈ el_pos ∧ (x, a) ∈ lin))"
apply (unfold ef_el_def)
by (simp add : set_eq_iff)

lemma ef_elI1 : "x ∈ el ⟹ y∈ ef ⟹ (∀ a . a ∈ ef ⟶ (a, x) ∉
el_pre) ⟹ (x, y) ∈ ef_el"
apply (unfold ef_el_def)
by (simp add : set_eq_iff)

lemma ef_elI2 : "x ∈ el ⟹ y∈ ef ⟹ (a, x) ∈ el_pos ⟹ (a, y) ∈ lin
⟹ (x, y) ∈ ef_el"
apply (unfold ef_el_def)
apply (simp add : set_eq_iff)
by metis

lemma ef_elI3 : "x ∈ el ⟹ y ∈ ef ⟹ (a, x) ∈ el_pos ⟹ (y, a) ∈ lin
⟹ (y, x) ∈ ef_el"
apply (unfold ef_el_def)
apply (simp add : set_eq_iff)
by metis

lemma ef_elI4 : "x ∈ el ⟹ (y, x) ∈ el_pos ⟹ (y, x) ∈ ef_el"
apply (unfold ef_el_def)
apply (simp add : set_eq_iff)
by (metis el_posD)

lemma ef_el_total : "x ∈ el ⟹ y ∈ ef ⟹ (x, y) ∈ ef_el ∨ (y, x) ∈
ef_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (frule lem_pos_exists)
apply (erule disjE)
apply (metis ef_elI1)
apply (erule exE)
apply (erule conjE)+
apply (case_tac "(a, y) ∈ lin")
apply (metis ef_elI2 el_posI)
apply (case_tac "(y, a) ∈ lin")
apply (metis ef_elI3 el_posI)
apply (case_tac "a=y")
apply (metis ef_elI4 el_posI)
```

by *(metis efD lem_is_pop_ef_then_is_ef_pop lem_is_push_ef_then_is_ef_push lem_push_or_pop1 lem_push_or_pop2)*

lemma *pr_in_ef_el_help : "x ∈ ef ⟹ y ∈ el ⟹ (x, y) ∈ pre ⟹ ¬ (∃ a . (a, y) ∈ el_pos) ⟹ False"*
apply *(drule not_el_posD)*
apply *(assumption)*
by *(metis elD el_preI1)*

lemma *pr_in_ef_el_help2 : "x ∈ ef ⟹ y ∈ el ⟹ (x, y) ∈ pre ⟹ (a, y) ∈ el_pos ⟹ (x = a) ∨ (x, a) ∈ lin"*
apply *(insert ef_linearizable)*
apply *(unfold sequential_stack_def)*
apply *(unfold sequential_stack_axioms_def)*
apply *(erule conjE)+*
apply *(unfold sequential_set_def)*
apply *(erule conjE)*
apply *(unfold sequential_set_axioms_def)*
apply *(case_tac "x=a")*
apply *metis*
apply *(case_tac "(x, a) ∈ lin")*
apply *metis*
apply *(case_tac "(a, x) ∈ lin")*
apply *(metis elD el_posD el_preI1)*
by *(metis ef_then_push_or_pop el_posD lem_is_pop_ef_then_is_ef_pop lem_is_push_ef_then_is_ef_push)*

lemma *pr_then_not_rev_pr : "x ∈ el ⟹ (x, y) ∈ same_val ⟹ (x, a) ∈ pre ⟹ (a, y) ∉ pre"*
apply *(rule notI)*
apply *(drule elD)*
by *(smt pre_transitive pre_irreflexive el_preD el_preI1 irrefl_def transD)*

lemma *ef_then_elimination_free : "a ∈ ef ⟹ a ∈ elimination_free pre val"*
by *(metis efD ef_valI ef_val_equals_elimination_free_val ef_val_then_ef elimination_free_valD same_valD)*

lemma *ef_pre_then_elimination_free_pre : "(a, b) ∈ ef_pre ⟹ (a, b) ∈ elimination_free_pre pre val"*
apply *(drule ef_preD)*
by *(metis ef_then_elimination_free elimination_free_preI)*

lemma *pr_then_lin : "x ∈ el ⟹ y ∈ ef ⟹ (x, y) ∈ pre ⟹ (a, x) ∈ el_pos ⟹ (a, y) ∈ lin"*
apply *(frule el_posD)*
apply *(erule conjE)+*
apply *(frule elD)*
apply *(erule exE)*
apply *(erule conjE)+*
by *(metis (lifting, no_types) pre_transitive abcdD ef_preI ef_pre_then_elimination_free_pre el_preD1 pr_in_lin pre_abcd same_val_unique transD)*

lemma *pr_in_ef_el1 : "x ∈ ef ⟹ y ∈ el ⟹ (x, y) ∈ pre ⟹ (x, y) ∈ ef_el"*
by *(metis ef_elI3 ef_elI4 pr_in_ef_el_help pr_in_ef_el_help2)*

```
lemma pr_in_ef_el2 : "x ∈ el ⟹ y ∈ ef ⟹ (x, y) ∈ pre ⟹ (x, y) ∈
ef_el"
by (metis ef_elI1 ef_elI2 not_el_posD pr_then_lin)

lemma ef_el_then_ef_and_el : "(a, b) ∈ ef_el ⟹ (a ∈ el ∧ b ∈ ef) ∨ (a
∈ ef ∧ b ∈ el)"
by (smt ef_elD)

lemma ef_el_irreflexive : "irrefl ef_el"
by (metis ef_el_then_ef_and_el ef_then_not_el irrefl_def)

lemma ef_el_antisymmetric : "antisym ef_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (rule antisymI)
apply (drule ef_elD)+
by (metis ef_then_not_el el_posD transD)

definition el_el :: "'a rel"
  where "el_el = {(x, y) . x ∈ el ∧ y ∈ el ∧
    ( (x, y) ∈ val ∨
      (¬(∃ a . (a, x) ∈ el_pos) ∧ (∃ b . (b, y) ∈ el_pos)) ∨
      (¬(∃ a . (a, x) ∈ el_pos) ∧ ¬(∃ b . ( b, y) ∈ el_pos) ∧ (∃ c. (c,
x) ∉ el_pre ∧ (c, y) ∈ el_pre)) ∨
      (¬(∃ a . (a, x) ∈ el_pos) ∧ ¬(∃ b . ( b, y) ∈ el_pos) ∧ (∀ c. (c,
x) ∈ el_pre ⟶ (c, y) ∈ el_pre) ∧ (x, y) ∈ minf) ∨
      (∃ a b . (a, x) ∈ el_pos ∧ (b, y) ∈ el_pos ∧ (a, b) ∈ lin) ∨
      (∃ a . (a, x) ∈ el_pos ∧ (a, y) ∈ el_pos ∧ (∃ c. (a, c) ∈ el_pos ∧
(c, x) ∉ el_pre ∧ (c, y) ∈ el_pre)) ∨
      ((∃ a . (a, x) ∈ el_pos ∧ (a, y) ∈ el_pos ∧ (∀ c. (a, c) ∈ el_pos
∧ (c, x) ∈ el_pre ⟶ (c, y) ∈ el_pre)) ∧ (x, y) ∈ minf)
 ) }"

lemma el_elD : "(x, y) ∈ el_el ⟹ x ∈ el ∧ y ∈ el ∧
    ( (x, y) ∈ val ∨
      (¬(∃ a . (a, x) ∈ el_pos) ∧ (∃ b . (b, y) ∈ el_pos)) ∨
      (¬(∃ a . (a, x) ∈ el_pos) ∧ ¬(∃ b . ( b, y) ∈ el_pos) ∧ (∃ c. (c,
x) ∉ el_pre ∧ (c, y) ∈ el_pre)) ∨
      (¬(∃ a . (a, x) ∈ el_pos) ∧ ¬(∃ b . ( b, y) ∈ el_pos) ∧ (∀ c. (c,
x) ∈ el_pre ⟶ (c, y) ∈ el_pre) ∧ (x, y) ∈ minf) ∨
      (∃ a b . (a, x) ∈ el_pos ∧ (b, y) ∈ el_pos ∧ (a, b) ∈ lin) ∨
      (∃ a . (a, x) ∈ el_pos ∧ (a, y) ∈ el_pos ∧ (∃ c. (a, c) ∈ el_pos ∧
(c, x) ∉ el_pre ∧ (c, y) ∈ el_pre)) ∨
      ((∃ a . (a, x) ∈ el_pos ∧ (a, y) ∈ el_pos ∧ (∀ c. (a, c) ∈ el_pos
∧ (c, x) ∈ el_pre ⟶ (c, y) ∈ el_pre)) ∧ (x, y) ∈ minf))"
apply (unfold el_el_def)
by (simp add : set_eq_iff)


lemma el_elI1 : "x ∈ el ⟹ y ∈ el ⟹(x, y) ∈ val ⟹ (x, y) ∈ el_el"
apply (unfold el_el_def)
```

```
by (simp add : set_eq_iff)

lemma el_elI2 : "x ∈ el ⟹ y ∈ el ⟹ ¬(∃ a . (a, x) ∈ el_pos) ⟹ (b,
y) ∈ el_pos ⟹ (x, y) ∈ el_el"
apply (unfold el_el_def)
apply (simp add : set_eq_iff)
by metis

lemma el_elI3 : "x ∈ el ⟹ y ∈ el ⟹ ¬(∃ a . (a, x) ∈ el_pos) ⟹
¬(∃ b . ( b, y) ∈ el_pos) ⟹ (c, x) ∉ el_pre ⟹ (c, y) ∈ el_pre ⟹
(x, y) ∈ el_el"
apply (unfold el_el_def)
apply (simp add : set_eq_iff)
by metis

lemma el_elI4 : "x ∈ el ⟹ y ∈ el ⟹ ¬(∃ a . (a, x) ∈ el_pos) ⟹
¬(∃ b . ( b, y) ∈ el_pos) ⟹ (∀ c. (c, x) ∈ el_pre ⟶ (c, y) ∈ el_pre)
⟹ (x, y) ∈ minf ⟹ (x, y) ∈ el_el"
apply (unfold el_el_def)
by (simp add : set_eq_iff)

lemma el_elI5 : "x ∈ el ⟹ y ∈ el ⟹ (a, x) ∈ el_pos ⟹ (b, y) ∈
el_pos ⟹ (a, b) ∈ lin ⟹ (x, y) ∈ el_el"
apply (unfold el_el_def)
apply (simp add : set_eq_iff)
by metis

lemma el_elI6 : "x ∈ el ⟹ y ∈ el ⟹ (a, x) ∈ el_pos ⟹ (a, y) ∈
el_pos ⟹ (∃ c. (a, c) ∈ el_pos ∧ (c, x) ∉ el_pre ∧ (c, y) ∈ el_pre)
⟹ (x, y) ∈ el_el"
apply (unfold el_el_def)
apply (simp add : set_eq_iff)
by metis

lemma el_elI7 : "x ∈ el ⟹ y ∈ el ⟹ (a, x) ∈ el_pos ⟹ (a, y) ∈
el_pos ⟹ (∀ c. (a, c) ∈ el_pos ∧ (c, x) ∈ el_pre ⟶ (c, y) ∈ el_pre)
⟹ (x, y) ∈ minf ⟹ (x, y) ∈ el_el"
apply (unfold el_el_def)
apply (simp add : set_eq_iff)
by metis

lemma el_el_then_el_and_el : "(a, b) ∈ el_el ⟹ a ∈ el ∧ b ∈ el"
apply (drule el_elD)
by smt

lemma el_el_total : "x ∈ el ⟹ y ∈ el ⟹ (x = y) ∨ (x, y) ∈ el_el ∨
(y, x) ∈ el_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (case_tac "x=y")
apply metis
apply (case_tac "(x, y) ∈ same_val")
apply (metis el_elI1 same_valD)
apply (case_tac "∃ a. (a, x) ∈ el_pos")
apply (case_tac "∃ b. (b, y) ∈ el_pos")
apply (erule exE)+
apply (case_tac "a=b")
apply (case_tac "(∀ c. (a, c) ∈ el_pos ∧ (c, x) ∈ el_pre ⟶ (c, y) ∈
el_pre)")
apply (case_tac "(x, y) ∈ minf")
apply (rule disjI2)
apply (rule disjI1)
apply (rule el_elI7)
apply (assumption)+
apply metis
apply (assumption)+
apply (case_tac "(∀ c. (b, c) ∈ el_pos ∧ (c, y) ∈ el_pre ⟶ (c, x) ∈
el_pre)")
apply (case_tac "(y, x) ∈ minf")
apply (rule disjI2)+
apply (rule el_elI7)
apply (assumption)+
apply metis
apply (assumption)+
apply (metis (lifting) el_def mem_Collect_eq not_minf_then_same_val)
apply (metis (lifting, no_types) el_elI6)
apply (metis (lifting, no_types) el_elI6)
apply (drule el_posD)+
apply (erule conjE)+
apply (frule_tac a="a" in ef_then_push_or_pop)
apply (frule_tac a="b" in ef_then_push_or_pop)
apply (erule disjE)+
apply (case_tac "a=b")
apply metis
apply (smt el_elI5 el_posI el_then_not_ef lem_is_push_ef_then_is_ef_push)
apply (drule lem_is_push_ef_then_is_ef_push)
apply (assumption)
apply (drule lem_is_pop_ef_then_is_ef_pop)
apply (assumption)
apply (metis (lifting, no_types) el_elI5 el_posI)
apply (erule disjE)
apply (drule lem_is_push_ef_then_is_ef_push)
apply (assumption)
apply (drule lem_is_pop_ef_then_is_ef_pop)
apply (assumption)
apply (metis (lifting, no_types) el_elI5 el_posI)
apply (case_tac "a=b")
apply metis
apply (drule lem_is_pop_ef_then_is_ef_pop)
apply (assumption)
apply (drule lem_is_pop_ef_then_is_ef_pop)
apply (assumption)
apply (metis (lifting, no_types) el_elI5 el_posI)
apply (metis (lifting, no_types) el_elI2)
by (metis (lifting, no_types) always_minf elD el_elI2 el_elI3 el_elI4)

lemma lem_pr_in_el_el_help : "x ∈ el ⟹ y ∈ el ⟹ (x, y) ∈ pre ⟹ (a,
x) ∈ el_pos ⟹
  (b, y) ∈ el_pos ⟹ a = b ∨ (a, b) ∈ lin"
```

```
by (metis pre_transitive abcd_def elD el_posD el_preD pr_in_ef_el_help2
pre_abcd trans_def)

lemma lem_pr_in_el_el_help2 : "x ∈ el ⟹ y ∈ el ⟹ (x, y) ∈ pre ⟹
(a, x) ∈ el_pos ⟹
  ∃ b .(b, y) ∈ el_pos"
by (metis pre_transitive abcd_def elD el_posD el_preD pr_in_ef_el_help
pre_abcd trans_def)

lemma lem_pr_in_el_el : "x ∈ el ⟹ y ∈ el ⟹ (x, y) ∈ pre ⟹ (x, y)
∈ el_el"
apply (case_tac "x=y")
apply (metis pre_irreflexive lem_irreflexive_then_not_equal)
apply (case_tac "(x, y) ∈ same_val")
apply (metis def_val_then_not_pr el_elI1 same_valD val_then_not_pr_def)
apply (case_tac "∃ a. (a, x) ∈ el_pos")
apply (case_tac "∃ b. (b, y) ∈ el_pos")
apply (metis pre_irreflexive elD el_elI5 el_elI6 el_preD el_preI1
lem_irreflexive_then_not_equal lem_pr_in_el_el_help)
apply (metis lem_pr_in_el_el_help2)
by (metis pre_irreflexive elD el_elI2 el_elI3 el_preD el_preI1
lem_irreflexive_then_not_equal)

lemma lem_position_exists : "(ia, ra) ∈ val ⟹ (ia, ra) ∉ pre ⟹
  (∀ a . a ∈ ef ⟶ ((a, ia) ∉ pre ∧ (a, ra) ∉ pre)) ∨
  (∃ a . a ∈ ef ∧ ((a, ia) ∈ pre ∨ (a, ra) ∈ pre) ∧ (∀ b . b ∈ ef ∧ (a,
b) ∈ lin ⟶ (b, ia) ∉ pre ∧ (b, ra) ∉ pre))"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (insert finite_set)
apply (erule finite_induct)
apply (metis empty_iff)
apply (erule disjE)
apply (case_tac "(x, ia) ∈ pre")
apply (rule disjI2)
apply (rule_tac x="x" in exI)
apply (rule conjI)
apply (metis insertCI)
apply (rule conjI)
apply metis
apply (rule allI)
apply (rule impI)
apply (erule_tac x="b" in allE)
apply (metis insertE lem_irreflexive_then_not_equal)
apply (metis insertE lem_irreflexive_then_not_equal)
apply (erule exE)+
apply (erule conjE)+
apply (erule disjE)
apply (smt  insert_iff irrefl_def trans_def)
apply (rule disjI2)
apply (case_tac "(x, ia) ∈ pre")
apply (case_tac "(x, a) ∈ lin")
apply (rule_tac x="a" in exI)
apply (metis antisym_def insert_iff)
apply (case_tac "(a, x) ∈ lin")
apply (rule_tac x="x" in exI)
apply (metis (full_types) insert_iff irrefl_def transD)
apply (metis (full_types) insert_iff)
apply (case_tac "(x, ra) ∈ pre")
apply (case_tac "(x, a) ∈ lin")
apply (rule_tac x="a" in exI)
apply (metis antisym_def insert_iff)
apply (case_tac "(a, x) ∈ lin")
apply (rule_tac x="x" in exI)
apply (metis (full_types) insert_iff irrefl_def transD)
apply (metis (full_types) insert_iff)
apply (rule_tac x="a" in exI)
by (metis (full_types) insert_iff)

lemma el_el_irreflexive : "irrefl el_el"
apply (insert ef_linearizable)
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (drule el_elD)
by (metis  def_value el_posD lem_irreflexive_then_not_equal
minf_irreflexive val_order_def)

lemma el_el_antisymmetric : "antisym el_el"
apply (insert ef_linearizable)
apply (rule antisymI)
apply (drule el_elD)+
apply (erule conjE)+
apply (erule disjE)+
apply (metis def_value val_order_def)
apply (smt el_posD el_preD el_preI2 not_el_posD same_valI1 same_valI2
same_val_then_not_minf)
apply (erule disjE)
apply (smt el_posD el_posI el_preD el_preI2 same_valI1 same_valI2
same_val_then_not_minf)
apply (erule disjE)
apply smt
apply (erule disjE)
apply smt
apply (erule disjE)+
apply (metis abcd_def el_pre_abcd)
apply smt
apply (erule disjE)
apply smt
apply (erule disjE)+
apply (metis antisymD minf_antisymmetric)
apply smt
apply (erule disjE)
apply metis
apply (erule disjE)+
apply (metis abcdD el_posD el_pre_abcd)
apply (metis el_posD one_el_pos)
apply (erule disjE)
apply (metis el_posD one_el_pos)
apply (erule disjE)
apply (metis abcdD el_posD el_pre_abcd one_el_pos)
apply (erule disjE)
apply (metis one_el_pos)
by (metis antisymD minf_antisymmetric)

lemma val_not_el_pos_then_not_el_pos : "(x, y) ∈ same_val ⟹ ¬ (∃a. (a,
y) ∈ el_pos) ⟹ ¬ (∃a. (a, x) ∈ el_pos)"
by (metis elD elI2 el_posD el_preD el_preI2 not_el_posD same_val_unique)

lemma val_minf_then_minf : "(x, y) ∈ same_val ⟹ (y, z) ∈ minf ⟹ (x,
z) ∈ minf"
by (metis min.commute minfD minfI same_val_commutative same_val_unique)

lemma val_minf_then_minf2 : "(x, y) ∈ same_val ⟹ (z, y) ∈ minf ⟹ (z,
x) ∈ minf"
by (metis min.commute minfD minfI same_val_commutative same_val_unique)

lemma val_then_el_pos1 : "(x, y) ∈ same_val ⟹ (a, x) ∈ el_pos ⟹ (a,
y) ∈ el_pos"
by (smt elI3 el_posD el_posI el_preD el_preI2 same_val_commutative)

lemma val_then_el_pos2 : "(x, y) ∈ same_val ⟹ (a, y) ∈ el_pos ⟹ (a,
x) ∈ el_pos"
by (smt elI3 el_posD el_posI el_preD el_preI2 same_val_commutative)

lemma val_then_el_pre : "(x, y) ∈ same_val ⟹ (a, x) ∈ el_pre ⟹ (a, y)
∈ el_pre"
by (metis el_preD el_preI2)

lemma el_el_transitive_help : "(y, z) ∈ same_val ⟹ (∀c. (a, c) ∈ el_pos
∧ (c, x) ∈ el_pre ⟶ (c, y) ∈ el_pre) ⟹
  (∀c. (a, c) ∈ el_pos ∧ (c, x) ∈ el_pre ⟶ (c, z) ∈ el_pre)"
by (metis val_then_el_pre)

lemma el_el_transitive : "trans el_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (rule transI)
apply (drule el_elD)+
apply (erule conjE)+
apply (erule disjE)+
apply (metis def_value val_order_def)
apply (erule disjE)
apply (metis (hide_lams, no_types) el_elI2 same_valI2 val_then_el_pos2)
apply (erule disjE)
apply (smt el_elI3 el_posD el_preD el_preI2 not_el_posD same_valI1)
apply (erule disjE)
apply (erule conjE)+
apply (frule same_valI1)
apply (frule val_not_el_pos_then_not_el_pos)
apply (assumption)
apply (frule_tac x="x" and y="y" and z="z" in val_minf_then_minf)
apply (assumption)
apply (metis (hide_lams, no_types) el_elI4 el_preD el_preI2)
apply (erule disjE)
apply (metis (hide_lams, no_types) el_elI5 same_valI1 val_then_el_pos2)
apply (erule disjE)
apply (smt el_elI6 el_preD el_preI1 same_valI1 same_valI2 val_then_el_pos1)
apply (smt el_elI7 el_preD el_preI1 same_valI1 same_valI2
val_minf_then_minf val_then_el_pos2)
apply (erule disjE)+
apply (metis (hide_lams, no_types) el_elI2 same_valI1 val_then_el_pos1)
apply (erule disjE)
apply (metis (hide_lams, no_types) antisym_def el_elI1 el_elI2 el_elI3
el_el_antisymmetric)
apply (erule disjE)
apply (smt el_elI4 el_preD el_preI1 same_valI1 same_valI2
val_minf_then_minf2 val_then_el_pos1)
apply (erule disjE)
apply (metis (hide_lams, no_types) el_elI5 same_valI1 val_then_el_pos1)
apply (erule disjE)
apply (smt el_elI6 el_preD1 el_preI2 same_valI1 same_val_unique
val_then_el_pos1)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)+
apply (frule same_valI1)
apply (frule same_val_commutative)
apply (frule_tac x="z" and y="y" and z="x" in val_minf_then_minf2)
apply (assumption)
apply (frule_tac y="y" and z="z" and x="x" and a="a" in
el_el_transitive_help)
apply (assumption)
apply (frule_tac x="y" and y="z" and a="a" in val_then_el_pos1)
apply (assumption)
apply (metis (hide_lams, no_types) el_elI7)
apply (erule disjE)
apply (smt el_elI2)
apply (erule disjE)
apply (smt el_elI2)
apply (erule disjE)
apply (smt abcd_def el_elI3 el_pre_abcd)
apply (erule disjE)
apply (smt el_elI3)
apply (erule disjE)
apply (smt el_elI4 minf_transitive trans_def)
apply (erule disjE)
apply metis
apply (erule disjE)
apply (metis (full_types) el_elI5 one_el_pos transE)
apply (erule disjE)
apply (metis (hide_lams, no_types) el_elI5 one_el_pos)
apply (erule disjE)
apply (smt abcd_def el_elI6 el_pre_abcd one_el_pos)
apply (erule disjE)
apply (metis (hide_lams, no_types) el_elI6 one_el_pos)
apply (erule conjE)+
apply (erule exE)+
apply (rule_tac x="x" and y="z" and a="a" in el_elI7)
apply (assumption)+
apply metis
apply (metis one_el_pos)
apply (metis one_el_pos)
by (metis minf_transitive transD)

lemma ef_el_ef_then_lin : "x ∈ ef ⟹ y ∈ el ⟹ z ∈ ef ⟹ (x, y) ∈
ef_el ⟹ (y, z) ∈ ef_el ⟹ (x, z) ∈ lin"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (drule ef_elD)+
by (smt ef_then_not_el el_posD one_el_pos transD)

lemma el_ef_el_then_lin : "x ∈ el ⟹ y ∈ ef ⟹ z ∈ el ⟹ (x, y) ∈
ef_el ⟹ (y, z) ∈ ef_el ⟹ (x, z) ∈ el_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (drule ef_elD)+
by (smt el_elI2 el_elI5 el_posD el_then_not_ef trans_def)

lemma el_el_ef_then_lin : "x ∈ el ⟹ y ∈ el ⟹ z ∈ ef ⟹ (x, y) ∈
el_el ⟹ (y, z) ∈ ef_el ⟹ (x, z) ∈ ef_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (drule el_elD)
apply (drule ef_elD)
apply (erule conjE)+
apply (erule disjE)+
apply (metis (hide_lams, no_types) ef_elI1 same_valI1 val_then_el_pre)
apply (erule disjE)
apply (metis el_posD)
apply (smt abcdD ef_elI1 el_posD el_pre_abcd)
apply (erule disjE)
apply (metis (hide_lams, no_types) ef_elI2 ef_then_not_el same_valI2
val_then_el_pos1)
apply (erule disjE)+
apply (metis (hide_lams, no_types) ef_elI1 not_el_posD)
apply (erule disjE)
apply metis
apply (smt ef_elI2 one_el_pos trans_def)
apply (erule disjE)+
apply (metis ef_then_not_el)
apply (metis ef_then_not_el)
by (metis ef_then_not_el)

lemma ef_el_el_then_lin : "x ∈ ef ⟹ y ∈ el ⟹ z ∈ el ⟹ (x, y) ∈
ef_el ⟹ (y, z) ∈ el_el ⟹ (x, z) ∈ ef_el"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (drule el_elD)
apply (drule ef_elD)
apply (erule disjE)
apply (metis (hide_lams, no_types) el_then_not_ef)
apply (erule disjE)
apply (metis ef_then_not_el)
apply (erule disjE)
apply (smt ef_elI3 ef_elI4 one_el_pos same_valI1 val_then_el_pos1)
apply (erule conjE)+
apply (erule disjE)
apply (metis (hide_lams, no_types) ef_elI3 same_valI1 val_then_el_pos1)
apply (erule disjE)
apply metis
by (smt ef_elI2 ef_elI3 ef_el_ef_then_lin el_posD one_el_pos)

definition ef_lin :: "'a rel"
 where "ef_lin = {(a, b) . (a, b) ∈ lin ∧ a ∈ ef ∧ b ∈ ef}"

lemma ef_linI : "(a, b) ∈ lin ⟹ a ∈ ef ⟹ b ∈ ef ⟹ (a, b) ∈
ef_lin"
apply (unfold ef_lin_def)
by (simp add: set_eq_iff)

lemma ef_linD : "(a, b) ∈ ef_lin ⟹ (a, b) ∈ lin ∧ a ∈ ef ∧ b ∈ ef"
apply (unfold ef_lin_def)
by (simp add: set_eq_iff)

lemma ef_lin_transitive: "trans ef_lin"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
by (metis ef_linD ef_linI trans_def)
```

```
definition witness :: "'a rel"
  where "witness = ef_lin Un ef_el Un el_el"

lemma lin_then_witness : "(a, b) ∈ ef_lin ⟹ (a, b) ∈ witness"
by (metis UnCI witness_def)

lemma ef_el_then_witness : "(a, b) ∈ ef_el ⟹ (a, b) ∈ witness"
by (metis UnCI witness_def)

lemma el_el_then_witness : "(a, b) ∈ el_el ⟹ (a, b) ∈ witness"
by (metis UnCI witness_def)

lemma pr_in_witness : "(a, b) ∈ pre ⟹ (a, b) ∈ witness"
by (metis ef_el_then_witness ef_linI ef_then_elimination_free
el_el_then_witness el_or_ef1 el_or_ef2 elimination_free_preI lem_pr_in_el_el
lin_then_witness pr_in_ef_el1 pr_in_ef_el2 pr_in_lin)

lemma ef_then_in_lin : "a ∈ ef ⟹ b ∈ ef ⟹ a ≠ b ⟹ (a, b) ∈ lin ∨
(b, a) ∈ lin"
by (metis ef_linearizable ef_then_push_or_pop lem_is_pop_ef_then_is_ef_pop
lem_is_push_ef_then_is_ef_push sequential_set.total_on_pop
sequential_set.total_on_push sequential_set.total_on_push_pop
sequential_stackD)

lemma ef_el_b_ef_then_a_el : "(a, b) ∈ ef_el ⟹ b ∈ ef ⟹ a ∈ el"
by (metis ef_el_then_ef_and_el ef_then_not_el)

lemma witness_transitive : "trans witness"
apply (rule transI)
apply (unfold witness_def)
apply (erule UnE)+
apply (metis ef_lin_transitive lin_then_witness transE witness_def)
apply (drule ef_elD)
apply (erule disjE)
apply (erule conjE)+
apply (metis ef_linD ef_then_not_el)
apply (metis (lifting) Un_def ef_elI3 ef_linD ef_linI ef_lin_transitive
el_def el_posD mem_Collect_eq trans_def)
apply (erule UnE)
apply (drule ef_linD)
apply (erule conjE)+
apply (frule ef_el_b_ef_then_a_el)
apply (assumption)
apply (rule UnI1)
apply (rule UnI2)
apply (drule ef_elD)
apply (erule disjE)
apply (metis ef_elI1)
apply (erule disjE)
apply (metis ef_elI2 ef_linD ef_linI ef_lin_transitive el_posD transE)
apply (erule disjE)
apply (metis ef_then_not_el)
apply (metis ef_then_not_el)
apply (metis ef_el_ef_then_lin ef_el_then_ef_and_el ef_linI
el_ef_el_then_lin el_el_then_witness el_then_not_ef lin_then_witness
witness_def)
```

```
apply (erule UnE)
apply (metis ef_linD el_el_then_el_and_el el_then_not_ef)
apply (metis ef_el_el_then_lin ef_el_then_ef_and_el ef_el_then_witness
el_el_then_el_and_el el_then_not_ef witness_def)
apply (erule UnE)+
apply (metis ef_linD el_el_then_el_and_el el_then_not_ef)
apply (metis ef_el_then_ef_and_el ef_el_then_witness el_el_ef_then_lin
el_el_then_el_and_el el_then_not_ef witness_def)
by (metis el_el_then_witness el_el_transitive trans_def witness_def)

lemma Un_irrefl : "irrefl A ⟹ irrefl B ⟹ irrefl (A Un B)"
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
by (metis UnE)

lemma witness_irreflexive : "irrefl witness"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (unfold witness_def)
apply (erule UnE)+
apply (metis ef_linD)
apply (metis ef_el_irreflexive lem_irreflexive_then_not_equal)
by (metis el_el_irreflexive lem_irreflexive_then_not_equal)

lemma witness_antisymmetric : "antisym witness"
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (rule antisymI)
apply (unfold witness_def)
apply (erule UnE)+
apply (metis antisymD ef_linD)
apply (metis ef_el_b_ef_then_a_el ef_linD el_then_not_ef)
apply (metis UnE antisymD ef_el_antisymmetric ef_el_then_ef_and_el ef_linD
ef_then_not_el)
apply (erule UnE)
apply (metis ef_linD ef_then_not_el el_el_then_el_and_el)
apply (metis ef_el_then_ef_and_el el_el_then_el_and_el el_then_not_ef)
by (metis el_el_then_witness lem_irreflexive_then_not_equal trans_def
witness_def witness_irreflexive witness_transitive)
```

lemma *witness_total_on_push : "is_push a ⟹ is_push b ⟹ a ≠ b ⟹ (a, b) ∈ witness ∨ (b, a) ∈ witness"*
apply *(insert ef_linearizable)*
apply *(unfold sequential_stack_def)*
apply *(unfold sequential_stack_axioms_def)*
apply *(erule conjE)+*
apply *(unfold sequential_set_def)*
apply *(erule conjE)*
apply *(unfold sequential_set_axioms_def)*
apply *(unfold wellformed_history_def)*
apply *(erule conjE)*
apply *(frule_tac a="a"* **in** *push_then_el_or_ef)*
apply *(frule_tac a="b"* **in** *push_then_el_or_ef)*
apply *(erule disjE)+*
apply *(metis el_el_then_witness el_el_total)*
apply *(metis ef_el_then_witness ef_el_total)*
by *(metis ef_el_then_witness ef_el_total ef_linI ef_then_in_lin lin_then_witness)*

lemma *witness_total_on_pop : "is_pop a ⟹ is_pop b ⟹ a ≠ b ⟹ (a, b) ∈ witness ∨ (b, a) ∈ witness"*
apply *(insert ef_linearizable)*
apply *(unfold sequential_stack_def)*
apply *(unfold sequential_stack_axioms_def)*
apply *(erule conjE)+*
apply *(unfold sequential_set_def)*
apply *(erule conjE)*
apply *(unfold sequential_set_axioms_def)*
apply *(unfold wellformed_history_def)*
apply *(erule conjE)*
apply *(frule_tac a="a"* **in** *pop_then_el_or_ef)*
apply *(frule_tac a="b"* **in** *pop_then_el_or_ef)*
apply *(erule disjE)+*
apply *(metis el_el_then_witness el_el_total)*
apply *(metis ef_el_then_witness ef_el_total)*
by *(metis ef_el_then_witness ef_el_total ef_linI ef_then_in_lin lin_then_witness)*

lemma *witness_total_on_push_pop : "is_push a ⟹ is_pop b ⟹ (a, b) ∈ witness ∨ (b, a) ∈ witness"*
apply *(insert ef_linearizable)*
apply *(unfold sequential_stack_def)*
apply *(unfold sequential_stack_axioms_def)*
apply *(erule conjE)+*
apply *(unfold sequential_set_def)*
apply *(erule conjE)*
apply *(unfold sequential_set_axioms_def)*
apply *(unfold wellformed_history_def)*
apply *(erule conjE)*
apply *(frule_tac a="a"* **in** *push_then_el_or_ef)*
apply *(frule_tac a="b"* **in** *pop_then_el_or_ef)*
apply *(erule disjE)+*
apply *(metis def_value el_el_then_witness el_el_total is_popD is_pushD val_order_def)*
apply *(metis ef_el_then_witness ef_el_total)*
by *(metis ef_el_then_witness ef_el_total ef_linI lem_is_pop_ef_then_is_ef_pop lem_is_push_ef_then_is_ef_push lin_then_witness)*

```
lemma el_then_same_witness : "x ∈ el ⟹ (x, y) ∈ same_val ⟹ (a, x) ∈
witness ⟹ (x, a) ∈ same_val ∨ (a, y) ∈ witness"
apply (unfold witness_def)
apply (erule UnE)+
apply (metis ef_linD el_then_not_ef)
apply (drule ef_elD)
apply (erule disjE)
apply (metis el_then_not_ef)
apply (erule disjE)
apply (metis ef_then_not_el)
apply (erule disjE)
apply (metis ef_elI4 ef_el_then_witness elI3 val_then_el_pos1 witness_def)
apply (metis ef_elI3 ef_el_then_witness elI3 val_then_el_pos1 witness_def)
apply (drule el_elD)
apply (erule conjE)+
apply (erule disjE)
apply (metis same_valI2)
apply (erule disjE)
apply (metis el_elI2 el_el_then_witness el_posD val_then_el_pos1
witness_def)
apply (erule disjE)
apply (metis elI3 el_elI2 el_elI3 el_el_then_witness val_then_el_pre
witness_def)
apply (erule disjE)
apply (erule conjE)+
apply (frule same_val_commutative)
apply (frule_tac x="y" and y="x" and z="a" in val_minf_then_minf2)
apply (assumption)
apply (smt elI3 el_elI4 el_el_then_witness val_then_el_pos1 val_then_el_pre
witness_def)
apply (erule disjE)
apply (metis elI3 el_elI5 el_el_then_witness val_then_el_pos1 witness_def)
apply (erule disjE)
apply (smt elI3 el_elI6 el_el_then_witness el_el_transitive_help
val_then_el_pos1 witness_def)
apply (erule conjE)+
apply (frule same_val_commutative)
apply (frule_tac x="y" and y="x" and z="a" in val_minf_then_minf2)
apply (assumption)
apply (erule exE)
apply (erule conjE)+
apply (frule_tac x="x" and y="y" and a="aa" in val_then_el_pos1)
apply (assumption)
by (smt elI3 el_elI7 el_el_then_witness minfD same_val_commutative
same_val_unique val_then_el_pre witness_def)

lemma witness_then_push_or_pop1 : "(a, b) ∈ witness ⟹ is_push a ∨
is_pop a"
by (smt UnE ef_elD ef_linD ef_then_push_or_pop elD el_el_then_el_and_el
lem_val_then_push_and_pop same_valD witness_def)

lemma witness_then_push_or_pop2 : "(a, b) ∈ witness ⟹ is_push b ∨
is_pop b"
apply (unfold witness_def)
apply (erule UnE)+
apply (metis ef_linD ef_then_push_or_pop)
apply (metis ef_el_then_ef_and_el ef_then_push_or_pop elD
lem_val_then_push_and_pop same_valD)
by (metis elD el_el_then_el_and_el lem_val_then_push_and_pop same_valD)

lemma el_then_same_witness2 : "x ∈ el ⟹ (x, y) ∈ same_val ⟹ (x, a) ∈
witness ⟹ (x, a) ∈ same_val ∨ (y, a) ∈ witness"
apply (rotate_tac -1)
apply (erule contrapos_pp)
apply (safe)
apply (case_tac "(a, y) ∈ witness")
apply (metis antisymD elI3 el_then_same_witness
lem_irreflexive_then_not_equal same_val_commutative same_val_unique
witness_antisymmetric witness_irreflexive)
by (metis is_popI is_pushI same_valD witness_then_push_or_pop2
witness_total_on_pop witness_total_on_push witness_total_on_push_pop)

lemma ef_ef_then_lin : "a ∈ ef ⟹ b ∈ ef ⟹ (a, b) ∈ witness ⟹ (a,
b) ∈ ef_lin"
apply (unfold witness_def)
apply (erule UnE)+
apply (assumption)
apply (metis ef_el_b_ef_then_a_el el_then_not_ef)
by (metis el_el_then_el_and_el el_then_not_ef)

lemma ef_val_ef : "(a, b) ∈ val ⟹ a ∈ ef ⟹ b ∈ ef"
by (metis ef_val_then_ef same_valI1)

lemma witness_ooo : "sequential_lifo witness val"
apply (unfold sequential_lifo_def)
apply (insert ef_linearizable)
apply (unfold sequential_stack_def)
apply (unfold sequential_stack_axioms_def)
apply (erule conjE)+
apply (unfold sequential_set_def)
apply (erule conjE)
apply (unfold sequential_set_axioms_def)
apply (unfold wellformed_history_def)
apply (erule conjE)
apply (rule allI)+
apply (rule impI)+
apply (frule_tac ia="ia" in is_pushI)
apply (drule push_then_el_or_ef)
apply (erule disjE)
apply (frule_tac x="ia" and y="ra" and a="ib" in  el_then_same_witness2)
apply (metis same_valI1)
apply assumption
apply (erule disjE)
apply (metis def_value same_valD val_order_def)
apply (metis antisymD def_value val_order_def witness_antisymmetric)
apply (frule_tac ia="ib" in is_pushI)
apply (drule push_then_el_or_ef)
apply (erule disjE)
apply (frule_tac x="ib" and y="rb" and a="ra" in el_then_same_witness2)
apply (metis same_valI1)
apply assumption
apply (erule disjE)
apply (metis elI3 el_then_not_ef same_valI2)
apply metis
apply (frule_tac a="ia" and b="ra" in ef_val_then_ef)
apply (metis same_valI1)
apply (frule_tac a="ia" and b="ib" in ef_ef_then_lin)
apply (assumption)+
apply (frule_tac a="ib" and b="ra" in ef_ef_then_lin)
apply (assumption)+
apply (rule lin_then_witness)
apply (frule_tac a="ib" and b="rb" in ef_val_ef)
apply (assumption)
by (smt ef_linD ef_linI ef_valI sequential_lifo_def)

lemma val_then_not_witness : "(ia, ra) ∈ val ⟹ (ra, ia) ∉ witness"
apply (frule_tac ia="ia" in is_pushI)
apply (drule push_then_el_or_ef)
apply (erule disjE)
apply (frule_tac y="ra" in el_elI1)
apply (metis elI3 same_valI1)
apply (assumption)
apply (drule el_el_then_witness)
apply (rule notI)
apply (metis antisymD def_value val_order_def witness_antisymmetric)
by (metis antisym_def def_val_then_not_pr elI2 el_then_not_ef pr_in_witness
same_valI2 val_then_not_pr_def witness_antisymmetric)

lemma witness_wellformed_history : "wellformed_history witness val f"
apply (rule wellformed_historyI)
apply (metis witness_transitive)
apply (metis witness_antisymmetric)
apply (metis witness_irreflexive)
apply (metis operations_countable)
apply (metis operations_finite)
apply (metis def_push_once)
apply (metis def_pop_once)
apply (metis def_value)
apply (metis is_pop_def is_push_def witness_then_push_or_pop1
witness_then_push_or_pop2)
by (metis val_then_not_pr_def val_then_not_witness)

lemma witness_sequential_history : "sequential_set witness val f"
apply (unfold sequential_set_def)
apply (rule conjI)
apply (rule witness_wellformed_history)
apply (unfold sequential_set_axioms_def)
apply (rule conjI)
apply (metis witness_total_on_push)
apply (rule conjI)
apply (metis witness_total_on_pop)
by (metis witness_total_on_push_pop)

lemma witness_sequential_stack : "sequential_stack witness val f"
apply (unfold sequential_stack_def)
apply (rule conjI)
apply (rule witness_sequential_history)
apply (unfold sequential_stack_axioms_def)
by (rule witness_ooo)

theorem elimination_stack_linearizable : "∃ prt .( pre ⊆ prt ∧
sequential_stack prt val f)"
```

```
apply (rule_tac x="witness" in exI)
by (metis pr_in_witness subrelI witness_sequential_stack)

end



theorem help_elimination_stack : "wellformed_history pre val f ⟹ abcd
pre ⟹
  linearizable_non_empty_stack (elimination_free_pre pre val)
(elimination_free_val pre val)
    ⟹ ∃ lin . elimination_stack pre val f lin"
apply (unfold linearizable_non_empty_stack_def)
apply (erule conjE)+
apply (erule exE)+
apply (rule_tac x="prt" in exI)
apply (rule elimination_stackI)
apply (assumption)+
apply (erule conjE)
apply (frule sequential_stackD)
apply (rule sequential_stackI)
apply (rule sequential_setI)
apply (rule wellformed_historyI)
apply (erule conjE)
apply (drule sequential_setD)
apply (erule conjE)+
apply (drule_tac pre="prt" in wellformed_historyD)
apply (erule conjE)+
apply (assumption)+
apply (drule wellformed_historyD)
apply (erule conjE)+
apply (drule sequential_setD)
apply (erule conjE)+
apply (drule_tac pre="prt" in wellformed_historyD)
apply (erule conjE)+
apply (assumption)
apply (metis sequential_setD wellformed_history.pre_irreflexive)
apply (metis wellformed_history.operations_countable)
apply (metis sequential_setD wellformed_history.operations_finite)
apply (metis elimination_free_valD push_once_def
wellformed_history.def_push_once)
apply (metis elimination_free_valD pop_once_def
wellformed_history.def_pop_once)
apply (metis elimination_free_valD val_order_def
wellformed_history.def_value)
apply (erule conjE)+
apply (drule sequential_setD)
apply (erule conjE)+
apply (drule wellformed_historyD)
apply (metis (full_types) wellformed_historyD)
apply (erule conjE)+
apply (drule wellformed_historyD)
apply (metis sequential_setD wellformed_historyD)
apply (metis sequential_setD)
apply (metis sequential_setD)
apply (metis sequential_setD)
apply metis
apply (metis in_mono)
apply (metis finite_elimination_free wellformed_historyD)
by metis



theorem elimination_theorem : "wellformed_history pre val f ⟹ abcd pre
⟹
  linearizable_non_empty_stack (elimination_free_pre pre val)
(elimination_free_val pre val) ⟹
  linearizable_non_empty_stack pre val"
apply (drule linearizable_non_empty_stackD)
apply (erule conjE)
apply (erule exE)+
apply (erule conjE)
apply (frule_tac pre="pre" and val="val" in help_elimination_stack)
apply (assumption)+
apply (frule wellformed_historyD)
apply (erule conjE)+
apply (rule_tac pre="(elimination_free_pre pre val)" and prt="prt" and
f="fa" and val="(elimination_free_val pre val)" in
linearizable_non_empty_stackI)
apply (rule conc_historyI)
apply (metis conc_historyD)
apply (metis conc_historyD)
apply (metis conc_historyD)
apply (rule abcdI)
apply (drule elimination_free_preD)+
apply (erule conjE)+
apply (metis (full_types) abcd_def elimination_free_preI)
apply metis
apply metis
by (metis conc_historyI elimination_stack.elimination_stack_linearizable
linearizable_non_empty_stack_def wellformed_history.pre_antisymmetric
wellformed_history.pre_irreflexive wellformed_history.pre_transitive)

end
```

A.5 Histories without Elimination and Pop-Empty Operations

```
theory remove_relaxed_stack
imports Main insert_pr_relaxed_stack
begin

locale remove_relaxed_stack = wellformed_history +
  fixes ir :: "'a rel"
  assumes pr_abcd : "abcd pre"
  assumes val_then_pr: "(a, b) ∈ val ⟹ (a, b) ∈ pre"
  assumes ir_alternating: "alternating val ir"
  assumes pr_u_ir_acyclic: "acyclic (pre ∪ ir)"
  assumes rem_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈
(ins_order pre val ir) ⟹
                    (ib, ra) ∈ ir ⟹ (ra, rb) ∉ (rem_order pre val ir)"
  assumes pops_finite_def : "finite {a. isPop val a}"
```

```
lemma remove_relaxed_stackI : "wellformed_history pre val f ⟹ abcd pre
⟹
∀ a b . (a, b) ∈ val ⟶ (a, b) ∈ pre ⟹
alternating val ir ⟹
acyclic (pre ∪ ir) ⟹
∀ ia ra ib rb . (ia, ra) ∈ val ⟶ (ib, rb) ∈ val ⟶ (ia, ib) ∈
(ins_order pre val ir) ⟶
                                    (ib, ra) ∈ ir ⟶ (ra, rb) ∉ (rem_order
pre val ir) ⟹
remove_relaxed_stack pre val f ir"
apply (unfold remove_relaxed_stack_def)
apply (rule conjI)
apply (assumption)
apply (unfold remove_relaxed_stack_axioms_def)
apply (rule conjI)+
apply assumption
apply (rule conjI)
apply assumption+
apply (rule conjI)+
apply assumption+
apply (rule conjI)
apply (assumption)
by (metis finite_Collect_disjI wellformed_historyD)

lemma remove_relaxed_stackD : "remove_relaxed_stack pre val f ir ⟹
wellformed_history pre val f ∧ abcd pre ∧
(∀ a b . (a, b) ∈ val ⟶ (a, b) ∈ pre) ∧
alternating val ir ∧
acyclic (pre ∪ ir) ∧
(∀ ia ra ib rb . (ia, ra) ∈ val ⟶ (ib, rb) ∈ val ⟶ (ia, ib) ∈
(ins_order pre val ir) ⟶
                                    (ib, ra) ∈ ir ⟶ (ra, rb) ∉ (rem_order
pre val ir))"
apply (unfold remove_relaxed_stack_def)
apply (unfold remove_relaxed_stack_axioms_def)
by metis

lemma order_correct_then_remove_relaxed_stack : "linearizable_set pre val
emp ⟹ order_correct pre val ⟹
  ∃ ir f . remove_relaxed_stack (elimination_free_pre (non_empty_pre pre
emp) val) (elimination_free_val (non_empty_pre pre emp) val) f
(elimination_free_ir (non_empty_pre pre emp) val ir)"
apply (frule linearizable_set_then_wellformed)
apply (erule exE)
apply (drule order_correctD)
apply (erule exE)
apply (rule_tac x="ir" in exI)
apply (rule_tac x="f" in exI)
apply (erule conjE)+
apply (rule remove_relaxed_stackI)
apply (metis wellformed_then_ef_wellformed)
apply (metis conc_history.abcd ef_conc_history linearizable_set_def
pre_conc_history_then_non_empty_pre_conc_history)
apply (rule allI)+
apply (rule impI)
apply (drule elimination_free_valD)
apply (rule elimination_free_preI)
apply metis
apply metis
apply (erule conjE)+
apply (drule elimination_freeD)
apply (drule wellformed_historyD)
apply (erule exE)
apply (erule conjE)+
apply (erule disjE)
apply (drule sameValD)
apply (erule disjE)
apply (metis pop_once_def)
apply (metis val_order_def)
apply (metis sameValD val_order_def val_then_not_pr_def)
apply (metis alternating_then_ef_alternating)
apply (rule_tac s="(pre ∪ ir)" in acyclic_subset)
apply (metis (erased, hide_lams) Un_commute)
apply (metis ef_empty_mono ef_ir_in_ir subset_refl sup.mono)
apply (rule allI)+
apply (rule impI)+
apply (rule notI)
apply (erule_tac x="ia" in allE)
apply (erule_tac x="ra" in allE)
apply (erule_tac x="ib" in allE)
apply (erule impE)
apply (rule conjI)
apply (metis elimination_free_valD)
apply (rule conjI)
apply (metis ef_val_then_pre)
apply (rule conjI)
apply (drule ins_orderD)
apply (erule conjE)+
apply (erule disjE)
apply (rule ins_orderI1)
apply (metis elimination_free_valD isPushI)
apply (metis elimination_free_valD isPushI)
apply (metis elimination_free_preD non_empty_pre_then_pre)
apply (metis (erased, hide_lams) ef_ins_then_ins elimination_free_irD
ins_orderI2)
apply (drule ins_orderD)
apply (erule conjE)+
apply (erule disjE)
apply (metis elimination_free_irD)
apply (metis elimination_free_irD)
apply (erule exE)
apply (erule conjE)
apply (frule_tac f="f" in elim_ef_rem_then_rem)
apply (assumption)
by (metis elimination_free_valD empty_rem_then_rem pop_once_def
wellformed_history.def_pop_once)

context remove_relaxed_stack
begin

lemma pops_finite : "finite {a. is_pop a}"
by (metis Collect_cong isPopD isPopI is_popD lem_val_then_push_and_pop
pops_finite_def)

lemma push_pop_ordered : "is_push a ⟹is_pop b ⟹ (a, b) : ir ∨ (b,a) ∈
ir"
```

```
by (metis alternating_def ir_alternating isPop_def isPush_def is_pushD
lem_pop_then_push_exists)

lemma ir_push_and_pop : "(a, b) ∈ ir ⟹ (is_push a ∧ is_pop b) ∨ (is_pop
a ∧ is_push b)"
by (metis alternating_def ir_alternating isPop_def isPush_def
lem_val_then_push_and_pop)

definition ins :: "'a rel"
where
  "ins = ins_order pre val ir"

lemma lem_pr_then_ins : " is_push a ⟹ is_push b ⟹ (a, b) ∈ pre ⟹
(a, b) ∈ ins"
apply (unfold ins_def)
apply (unfold ins_order_def)
apply (simp add: set_eq_iff)
by (metis isPush_def is_pushD)

lemma ir_push_and_pop1 : "(a, b) ∈ ir ⟹ is_push a ⟹ is_pop b"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma ir_push_and_pop2 : "(a, b) ∈ ir ⟹ is_push b ⟹ is_pop a"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma ir_push_and_pop3 : "(a, b) ∈ ir ⟹ is_pop a ⟹ is_push b"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma ir_push_and_pop4 : "(a, b) ∈ ir ⟹ is_pop b ⟹ is_push a"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma insI : "is_push a ⟹ is_push b ⟹ (a, c) ∈ pre ⟹ (c, b) ∈ ir
⟹ (a, b) ∈ ins"
by (metis ins_def ins_orderI2 isPush_def lem_push_then_pop_exists)

lemma insD : "(a, b) ∈ ins ⟹ is_push a ∧ is_push b ∧ ((a, b) ∈ pre
∨(∃ c . is_pop c ∧ (a, c) ∈ pre ∧ (c, b) ∈ ir))"
by (metis (hide_lams, no_types) ins_def ins_orderD ir_push_and_pop2
isPush_def lem_val_then_push_and_pop)

lemma ipr_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈
(ins_order pre val ir) ⟹ (ib, ra) ∈ ir ⟹ (ra, rb) ∉ pre"
apply (drule_tac ib="ib" and rb="rb" in rem_ooo)
apply (assumption)+
apply (rule notI)
by (metis ir_push_and_pop1 isPop_def is_popD lem_val_then_push_and_pop
rem_orderI1)

lemma ipr_ooo_ir : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈
(ins_order pre val ir) ⟹ (ib, ra) ∈ ir ⟹ ¬((ra, ic) ∈ ir ∧ (ic, rb) ∈
ir)"
apply (drule_tac ib="ib" and rb="rb" in rem_ooo)
apply (assumption)+
apply (rule notI)
```

```
by (metis (hide_lams, no_types) alternating_def def_value ir_alternating
isPop_def rem_orderI2 val_order_def)

lemma lem_val_then_ir : "(a, b) ∈ val ⟹ (a, b) ∈ ir"
apply (frule val_then_pr)
apply (frule lem_val_then_push_and_pop)
apply (erule conjE)
apply (frule_tac a="a" and b="b" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (assumption)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

lemma lem_push_pr_pop_then_ir : " is_push a ⟹ is_pop b ⟹ (a, b) ∈ pre
⟹ (a, b) ∈ ir"
apply (frule_tac a="a" and b="b" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (assumption)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

lemma not_ir_push_pop_ir : "(a, b) ∉ ir ⟹ is_push a ⟹ is_pop b ⟹
(b, a) ∈ ir"
by (metis push_pop_ordered)

lemma not_ir_pop_push_ir : "(a, b) ∉ ir ⟹ is_pop a ⟹ is_push b ⟹
(b, a) ∈ ir"
by (metis push_pop_ordered)

lemma ir_transitive : "(a, b) ∈ ir ⟹ (b, c) ∈ ir ⟹ (c, d) ∈ ir ⟹
(a, d) ∈ ir"
apply (case_tac "(a, d) ∈ ir")
apply (assumption)
apply (frule ir_push_and_pop)
apply (erule disjE)
apply (erule conjE)
apply (drule not_ir_push_pop_ir)
apply (assumption)
apply (metis ir_push_and_pop1 ir_push_and_pop3)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(b, c)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(c, d)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(d, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="c" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
apply (frule_tac a="c" and b="d" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
```

```
apply (assumption)
apply (metis acyclic_def pr_u_ir_acyclic transE trans_trancl)
apply (drule not_ir_pop_push_ir)
apply (erule conjE)
apply (assumption)
apply (metis ir_push_and_pop1 ir_push_and_pop3)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(b, c)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(c, d)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(d, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="c" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
apply (frule_tac a="c" and b="d" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
by (metis acyclic_def pr_u_ir_acyclic transE trans_trancl)

lemma lem_pop_pr_push_then_ir : " is_pop a ⟹ is_push b ⟹ (a, b) ∈ pre
⟹ (a, b) ∈ ir"
apply (frule_tac a="b" and b="a" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

lemma lem_ins_transitive : "trans ins"
apply (rule transI)
apply (drule insD)+
apply (erule conjE)+
apply (erule disjE)+
apply (metis pre_transitive lem_pr_then_ins transE)
apply (metis pre_transitive insI transE)
apply (erule disjE)
apply (smt abcdD ins_def insI ipr_ooo_ir lem_pop_then_push_exists
lem_pr_then_ins lem_push_pr_pop_then_ir lem_push_then_pop_exists pr_abcd
val_then_pr)
by (metis insI ir_transitive lem_push_pr_pop_then_ir)

lemma lem_ins_abcd : "abcd ins"
apply (rule abcdI)
apply (drule insD)+
apply (erule conjE)+
apply (erule disjE)+
apply (metis abcdD lem_pr_then_ins pr_abcd)
apply (metis abcdD insI lem_pr_then_ins pr_abcd)
by (metis abcdD insI lem_pr_then_ins pr_abcd)

lemma lem_ins_irreflexive : "irrefl ins"
apply (unfold irrefl_def)
apply (rule allI)
apply (rename_tac x)
apply (rule notI)
apply (drule insD)
apply (erule conjE)+
apply (erule disjE)
apply (metis pre_irreflexive lem_irreflexive_then_not_equal)
apply (erule exE)
apply (erule conjE)+
apply (frule_tac c="(x, c)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(c, x)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="x" and b="c" and c="x" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
by (metis acyclic_def pr_u_ir_acyclic)


definition lin :: "'a rel"
  where
    "lin = {(ra, rb) . (∃ ia ib . (ia, ra) ∈ val ∧ (ib, rb) ∈ val ∧ (ib,
ia) ∈ ins ∧ (ia, rb) ∈ ir)}"

lemma linD : "(ra, rb) ∈ lin ⟹ (∃ ia ib . (ia, ra) ∈ val ∧ (ib, rb) ∈
val ∧ (ib, ia) ∈ ins ∧ (ia, rb) ∈ ir)"
apply (unfold lin_def)
by (simp add: set_eq_iff)

lemma linI : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, rb) ∈ ir ⟹
(ib, ia) ∈ ins ⟹ (ra, rb) ∈ lin"
apply (unfold lin_def)
apply (simp add: set_eq_iff)
apply (rule_tac x="ia" in exI)
apply (rule conjI)
apply (assumption)
apply (rule_tac x="ib" in exI)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
by (assumption)+

definition pop_ir :: "'a rel"
  where
    "pop_ir = {(ra, rb) . is_pop ra ∧ is_pop rb ∧ (∃ ic . (ra, ic) ∈ ir ∧
(ic, rb) ∈ ir)}"

lemma pop_irD : "(a, b) ∈ pop_ir ⟹ is_pop a ∧ is_pop b ∧ (∃ ic . (a,
ic) ∈ ir ∧ (ic, b) ∈ ir)"
apply (unfold pop_ir_def)
by (simp add: set_eq_iff)

lemma pop_irI : "is_pop a ⟹ is_pop b ⟹ (a, c) ∈ ir ⟹ (c, b) ∈ ir
⟹ (a, b) ∈ pop_ir"
apply (unfold pop_ir_def)
apply (simp add: set_eq_iff)
by metis

lemma lem_pop_ir_transitive : "trans pop_ir"
apply (rule transI)
apply (drule pop_irD)+
apply (erule conjE)+
apply (erule exE)+
apply (erule conjE)+
apply (case_tac "(x, ica) ∈ ir")
apply (metis pop_irI)
apply (drule not_ir_pop_push_ir)
apply (assumption)
apply (metis ir_push_and_pop3)
apply (frule_tac c="(ica, x)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(x, ic)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(ic, y)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(y, ica)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="x" and b="ic" and c="y" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
apply (frule_tac a="y" and b="ica" and c="x" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
by (metis acyclic_def pr_u_ir_acyclic transD trans_trancl)

lemma lem_pop_ir_irreflexive : "irrefl pop_ir"
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (rename_tac x)
apply (drule pop_irD)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)
apply (frule_tac c="(ic, x)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(x, ic)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="x" and b="ic" and c="x" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
by (metis acyclic_def pr_u_ir_acyclic)

lemma lem_pop_ir_abcd : "abcd pop_ir"
apply (rule abcdI)
apply (drule pop_irD)+
apply (erule conjE)+
apply (erule exE)+
apply (erule conjE)+
by (metis (hide_lams, no_types) ir_push_and_pop3 lem_pop_ir_transitive
pop_irI push_pop_ordered trans_def)

lemma lem_lin_irreflexive : "irrefl lin"
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (unfold lin_def)
apply (simp add: set_eq_iff)
apply (erule exE)
apply (erule conjE)
apply (erule exE)
apply (erule conjE)+
by (metis def_push_once lem_ins_irreflexive lem_irreflexive_then_not_equal
push_once_def)

lemma lem_lin_transitive : "trans lin"
apply (rule transI)
apply (rename_tac rc rb ra)
apply (unfold lin_def)
apply (simp add: set_eq_iff)
apply (erule exE)+
apply (rename_tac ic ib)
apply (erule conjE)+
apply (erule exE)+
apply (rename_tac ibb ia)
apply (erule conjE)+
apply (rule_tac x="ic" in exI)
apply (rule conjI)
apply (assumption)
apply (rule_tac x="ia" in exI)
by (smt ins_def def_push_once ipr_ooo_ir lem_ins_transitive
lem_val_then_push_and_pop not_ir_push_pop_ir push_once_def transE)

lemma lem_prUlin_irreflexive: "irrefl (pre Un lin)"
by (metis pre_irreflexive Un_iff irrefl_def lem_lin_irreflexive)

lemma lem_lin_u_pop_ir_transitive : "trans (lin Un pop_ir)"
apply (rule transI)
apply (erule UnE)+
apply (metis UnI1 lem_lin_transitive transD)
apply (drule pop_irD)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)
apply (case_tac "(x, ic) ∈ ir")
apply (metis UnCI ir_push_and_pop2 ir_push_and_pop3 pop_irI)
apply (smt ins_def ipr_ooo_ir ir_push_and_pop3 is_pop_def linD
not_ir_pop_push_ir)
apply (erule UnE)
apply (drule pop_irD)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)
apply (case_tac "(ic, z) ∈ ir")
apply (metis UnCI ir_push_and_pop1 ir_push_and_pop3 pop_irI)
apply (smt ins_def ipr_ooo_ir ir_push_and_pop3 is_pop_def linD
not_ir_pop_push_ir)
by (metis UnI2 lem_pop_ir_transitive transD)

lemma lem_lin_u_pop_ir_irreflexive : "irrefl (lin Un pop_ir)"
by (metis UnE irrefl_def lem_lin_irreflexive lem_pop_ir_irreflexive)

lemma prUlinD : "(a, e) ∈ trancl (pre ∪ lin) ⟹(a, e) ∈ pre ∨ (a, e) ∈
lin ∨ (∃ b .(a, b) ∈ lin ∧ (b, e) ∈ pre) ∨ (∃ b. (a, b) ∈ pre ∧ (b, e)
∈ lin) ∨ (∃ b c .(a, b) ∈ lin ∧ (b, c) ∈ pre ∧ (c, e) ∈ lin)"
apply (erule trancl.induct)
apply (metis UnE)
apply (erule disjE)
apply (erule UnE)
apply (metis pre_transitive transD)
apply metis
apply (erule UnE)
apply (erule disjE)
apply metis
apply (erule disjE)
apply (metis pre_transitive transD)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (insert pr_abcd)
apply (frule_tac a="a" and b="ba" and c="b" and d="c" and r="pre" in
abcdD)
apply (assumption)+
apply (erule disjE)
apply (rule disjI1)
apply (assumption)

apply (metis ins_def ipr_ooo linD)
apply (metis (full_types) ins_def abcdD ipr_ooo linD)
by (metis lem_lin_transitive transD)

lemma lem_prUlin_acyclic: "acyclic (pre ∪ lin)"
apply (unfold acyclic_def)
apply (rule allI)
apply (rule notI)
apply (drule prUlinD)
apply (erule disjE)+
apply (metis pre_irreflexive lem_irreflexive_then_not_equal)
apply (erule disjE)+
apply (metis lem_irreflexive_then_not_equal lem_lin_irreflexive)
apply (erule disjE)+
apply (metis ins_def ipr_ooo linD)
apply (erule disjE)+
apply (metis ins_def ipr_ooo linD)
apply (erule exE)+
apply (insert lem_lin_transitive)
apply (erule conjE)+
apply (drule_tac r="lin" and x="c" and y="x" and z="b" in transD)
apply (assumption)+
by (metis ins_def ipr_ooo linD)

lemma lem_pre_then_not_pop_ir : "(a, b) ∈ pre ⟹ (b, a) ∈ pop_ir ⟹
False"
apply (unfold pop_ir_def)
apply (simp add: set_eq_iff)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, ic)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(ic, a)" and A="pre" and B="ir" in UnI2)
by (metis Un_commute acyclic_def pr_u_ir_acyclic r_into_trancl'
r_r_into_trancl trancl_trans)

lemma prUirUlinD : "(a, b) ∈ trancl (pre ∪ (lin ∪ pop_ir)) ⟹ (a, b) ∈
(pre ∪ (lin ∪ pop_ir)) ∨
                                                          (∃ c . (a, c)
∈ pre ∧ (c, b) ∈ (lin ∪ pop_ir)) ∨
                                                          (∃ c . (a, c)
∈ (lin ∪ pop_ir) ∧ (c, b) ∈ pre) ∨
                                                          (∃ c d . (a,
c) ∈ (lin ∪ pop_ir) ∧ (c, d) ∈ pre ∧ (d, b) ∈ (lin ∪ pop_ir))"
apply (erule trancl.induct)
apply (rule disjI1)
apply (assumption)
apply (erule disjE)+
apply (erule UnE)+
apply (metis pre_transitive UnI1 transD)
apply metis
apply (erule UnE)+
apply (metis UnI1)
apply (metis UnI2)
apply (metis UnCI lem_lin_u_pop_ir_transitive sup_commute transD)
apply (erule UnE)
apply (erule disjE)+
apply (erule exE)
apply (erule conjE)
apply (insert pr_abcd)
apply (drule_tac r="pre" and a="b" and b="c" and c="a" and d="ca" in
abcdD)
apply (assumption)+
apply (erule disjE)
apply (erule UnE)
apply (metis ins_def ipr_ooo linD)
apply (frule_tac c="(b, ca)" and A="pre" and B="pop_ir" in UnI1)
apply (frule_tac c="(ca, b)" and A="pre" and B="pop_ir" in UnI2)
apply (metis lem_pre_then_not_pop_ir)
apply (metis UnI1)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (metis pre_transitive transE)
apply (erule exE)+
apply (erule conjE)+
apply (drule_tac r="pre" and a="b" and b="c" and c="ca" and d="d" in
abcdD)
apply (assumption)+
apply (erule disjE)
apply (erule_tac c="(d, b)" in UnE)
apply (metis ins_def ipr_ooo linD)
apply (frule_tac c="(b, d)" and A="pre" and B="pop_ir" in UnI1)
apply (frule_tac c="(d, b)" and A="pre" and B="pop_ir" in UnI2)
apply (metis lem_pre_then_not_pop_ir)
apply metis
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (metis lem_lin_u_pop_ir_transitive transE)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply metis
apply (erule exE)+
apply (erule conjE)+
by (metis lem_lin_u_pop_ir_transitive transE)

lemma lem_pr_ir_lin_acyclic : "acyclic (pre ∪ (lin ∪ pop_ir))"
apply (unfold acyclic_def)
apply (rule allI)
apply (rule notI)
apply (drule prUirUlinD)
apply (erule disjE)
apply (metis UnE lem_irreflexive_then_not_equal
lem_lin_u_pop_ir_irreflexive pre_irreflexive)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)+
apply (erule UnE)
apply (metis ins_def ipr_ooo linD)
apply (frule_tac c="(x, c)" and A="pre" and B="pop_ir" in UnI1)
apply (frule_tac c="(c, x)" and A="pre" and B="pop_ir" in UnI2)
apply (metis lem_pre_then_not_pop_ir)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis ins_def ipr_ooo linD)
apply (frule_tac c="(c, x)" and A="pre" and B="pop_ir" in UnI1)
apply (frule_tac c="(x, c)" and A="pre" and B="pop_ir" in UnI2)
apply (metis lem_pre_then_not_pop_ir)
apply (erule exE)+
apply (erule conjE)+
apply (insert lem_lin_u_pop_ir_transitive)
apply (drule_tac r="(lin Un pop_ir)" and x="d" and y="x" and z="c" in
transD)
apply (assumption)+
apply (erule_tac c="(d, c)" in UnE)
apply (metis ins_def ipr_ooo linD)
apply (frule_tac c="(c, d)" and A="pre" and B="pop_ir" in UnI1)
apply (frule_tac c="(d, c)" and A="pre" and B="pop_ir" in UnI2)
by (metis lem_pre_then_not_pop_ir)

definition w1 :: "'a rel"
where
  "w1 = trancl (pre ∪ (lin ∪ pop_ir))"

definition wins1 :: "'a rel"
where
  "wins1 = {(a, b) . is_push a ∧ is_push b ∧ ((a, b) ∈ w1 ∨ (∃ c . is_pop
c ∧ (a, c) ∈ w1 ∧ (c, b) ∈ ir))}"

lemma lem_trancl_w1D : "(x, y) ∈ trancl w1 ⟹ (x, y) ∈ w1"
by (metis trancl_id trans_trancl w1_def)

lemma lem_w1_transitive : "trans w1"
by (metis trans_trancl w1_def)

lemma lem_w1_irreflexive : "irrefl w1"
by (metis acyclic_irrefl lem_pr_ir_lin_acyclic w1_def)

lemma lem_w1_acyclic : "acyclic w1"
by (metis lem_transitive_irreflexive_then_acyclic lem_w1_irreflexive
lem_w1_transitive)

lemma w1D : "(a, b) ∈ w1 ⟹ (a, b) ∈ (pre ∪ (lin ∪ pop_ir)) ∨
                                                          (∃ c . (a, c)
∈ pre ∧ (c, b) ∈ (lin ∪ pop_ir)) ∨
                                                          (∃ c . (a, c)
∈ (lin ∪ pop_ir) ∧ (c, b) ∈ pre) ∨
```

```
                                                      (∃ c d . (a,
c) ∈ (lin ∪ pop_ir) ∧ (c, d) ∈ pre ∧ (d, b) ∈ (lin ∪ pop_ir))"
by (metis prUirUlinD w1_def)

lemma lem_w1_then_push_or_pop1 : "(a, b) ∈ w1 ⟹ is_push a ∨ is_pop a"
apply (drule w1D)
apply (erule disjE)
apply (metis Un_iff lem_push_or_pop1 lem_val_then_push_and_pop linD
pop_irD)
apply (erule disjE)
apply (metis lem_push_or_pop1)
apply (erule disjE)
apply (metis Un_iff lem_val_then_push_and_pop linD pop_irD)
by (metis Un_iff lem_val_then_push_and_pop linD pop_irD)

lemma lem_w1_then_push_or_pop2 : "(a, b) ∈ w1 ⟹ is_push b ∨ is_pop b"
apply (drule w1D)
apply (erule disjE)
apply (metis Un_commute Un_iff is_pop_def lem_push_or_pop2 linD pop_irD)
apply (erule disjE)
apply (metis Un_iff is_pop_def linD pop_irD)
apply (erule disjE)
apply (metis lem_push_or_pop2)
by (metis Un_iff lem_val_then_push_and_pop linD pop_irD)

lemma lem_w1_push_then_pr : "(a, b) ∈ w1 ⟹ is_push a ⟹ is_push b ⟹
(a, b) ∈ pre"
apply (drule w1D)
apply (erule disjE)
apply (erule UnE)
apply (assumption)
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis irrefl_def lem_pop_ir_irreflexive pop_irD pop_irI
push_pop_ordered)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis irrefl_def lem_pop_ir_irreflexive pop_irD pop_irI
push_pop_ordered)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis def_value is_push_def linD val_order_def)
apply (metis irrefl_def lem_pop_ir_irreflexive pop_irD pop_irI
push_pop_ordered)
apply (erule exE)+
apply (erule conjE)+
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
by (metis lem_irreflexive_then_not_equal lem_pop_ir_irreflexive pop_irD
pop_irI push_pop_ordered)

lemma lem_wins1_then_ins : "(a, b) ∈ wins1 ⟹ (a, b) ∈ ins"
apply (unfold wins1_def)
apply (simp add: set_eq_iff)
apply (erule conjE)+
apply (erule disjE)
apply (drule w1D)
apply (erule disjE)
apply (metis lem_pr_then_ins lem_w1_push_then_pr r_into_trancl' w1_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis def_value is_pop_def lem_push_then_pop_exists pop_irD
val_order_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis def_value is_push_def lem_pop_then_push_exists pop_irD
val_order_def)
apply (erule exE)+
apply (erule conjE)+
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis def_value is_pop_def lem_push_then_pop_exists pop_irD
val_order_def)
apply (erule exE)
apply (erule conjE)+
apply (drule w1D)
apply (erule disjE)
apply (erule UnE)
apply (metis insI)
apply (erule UnE)
apply (metis def_value is_push_def linD val_order_def)
apply (metis def_value is_push_def lem_pop_then_push_exists pop_irD
val_order_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)

apply (smt insI ins_def ipr_ooo_ir lem_val_then_push_and_pop linD
not_ir_push_pop_ir)
apply (metis insI ir_transitive lem_irreflexive_then_not_equal
lem_pop_ir_irreflexive not_ir_push_pop_ir pop_irD pop_irI)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis def_value is_pop_def lem_push_then_pop_exists pop_irD
val_order_def)
apply (erule exE)+
apply (erule conjE)+
apply (erule UnE)+
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (erule UnE)
apply (metis lem_irreflexive_then_not_equal lem_pop_ir_irreflexive pop_irD
pop_irI push_pop_ordered)
by (metis def_value is_pop_def lem_push_then_pop_exists pop_irD
val_order_def)

lemma lem_w1_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈
wins1 ⟹ (ib, ra) ∈ ir ⟹ (rb, ra) ∈ w1"
apply (frule lem_wins1_then_ins)
apply (drule_tac ia="ib" and ib="ia" and ra="rb" and rb="ra" in linI)
apply (assumption)+
by (metis UnCI r_into_trancl' w1_def)

lemma lem_pr_ir_then_ir : "is_push a ⟹ is_push b ⟹ is_pop c ⟹ (a,
b) ∈ pre ⟹ (b, c) ∈ ir ⟹ (a, c) ∈ ir"
apply (case_tac "(a, c) ∈ ir")
apply (assumption)
apply (drule not_ir_push_pop_ir)
apply (assumption)+
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, c)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="c" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)
by (metis Un_iff acyclic_def pr_u_ir_acyclic trancl_into_trancl2)

lemma lem_w1Uir_then_push_or_pop1 : "(a, b) ∈ w1 Un ir ⟹ is_push a ∨
is_pop a"
by (metis UnE ir_push_and_pop lem_w1_then_push_or_pop1)

lemma lem_w1Uir_then_push_or_pop2 : "(a, b) ∈ w1 Un ir ⟹ is_push b ∨
is_pop b"
by (metis UnE ir_push_and_pop lem_w1_then_push_or_pop2)

lemma lem_trancl_w1UnirD : "(a, b) ∈ trancl(w1 Un ir) ⟹
  (is_push a ∧ is_push b ∧ ((a, b) ∈ pre ∨ (∃ c. (a, c)∈ ir ∧ (c, b)∈
ir))) ∨
  (is_push a ∧ is_pop b ∧ (a, b) ∈ ir) ∨
  (is_pop a ∧ is_push b ∧ (a, b) ∈ ir) ∨
  (is_pop a ∧ is_pop b ∧ (a, b) ∈ w1)"

apply (erule trancl.induct)
apply (frule lem_w1Uir_then_push_or_pop1)
apply (frule lem_w1Uir_then_push_or_pop2)
apply (erule disjE)+
apply (metis Un_iff ir_push_and_pop1 lem_w1_push_then_pr)

apply (rule disjI2)
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (erule UnE)
apply (drule w1D)
apply (erule disjE)
apply (metis (lifting) UnE def_value lem_pop_then_push_exists
lem_push_pr_pop_then_ir lem_push_then_pop_exists linD pop_irD val_order_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (smt ins_def ipr_ooo_ir lem_push_pr_pop_then_ir
lem_val_then_push_and_pop linD push_pop_ordered)
apply (metis ir_transitive lem_push_pr_pop_then_ir pop_irD)
apply (erule disjE)
apply (metis Un_iff def_value is_pop_def is_push_def linD pop_irD
val_order_def)
apply (metis UnE Un_commute def_value lem_pop_then_push_exists
lem_push_then_pop_exists linD pop_irD val_order_def)
apply (assumption)

apply (erule disjE)
apply (rule disjI2)
apply (rule disjI2)
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (erule UnE)
apply (drule w1D)
apply (erule disjE)
apply (metis (hide_lams, no_types) Un_iff def_value is_pop_def is_push_def
lem_pop_pr_push_then_ir linD pop_irD val_order_def)
apply (erule disjE)
apply (metis Un_iff def_value is_pop_def is_push_def linD pop_irD
val_order_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (smt ins_def ipr_ooo_ir lem_pop_pr_push_then_ir
lem_val_then_push_and_pop linD push_pop_ordered)
apply (metis ir_transitive lem_pop_pr_push_then_ir pop_irD)
apply (metis UnE Un_commute def_value lem_pop_then_push_exists
lem_push_then_pop_exists linD pop_irD val_order_def)
apply (assumption)
apply (rule disjI2)+
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (metis UnE def_value ir_push_and_pop3 lem_pop_then_push_exists
lem_push_then_pop_exists val_order_def)
apply (frule lem_w1Uir_then_push_or_pop2)
apply (erule disjE)+
apply (erule conjE)+
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (erule disjE)
apply (metis pre_transitive Un_iff ir_push_and_pop2 lem_push_pr_pop_then_ir
lem_w1_push_then_pr transD)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (metis (hide_lams, no_types) Un_iff ir_push_and_pop1
lem_irreflexive_then_not_equal lem_lin_u_pop_ir_irreflexive
lem_pr_ir_then_ir lem_w1_push_then_pr not_ir_push_pop_ir pop_irI)
apply (metis ir_push_and_pop2 ir_transitive push_pop_ordered)

apply (erule conjE)+
apply (rule disjI2)
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (erule UnE)
apply (erule disjE)
apply (smt lem_ins_irreflexive lem_ins_transitive
lem_irreflexive_then_not_equal lem_pr_then_ins lem_wins1_then_ins
mem_Collect_eq push_pop_ordered split_conv trans_def wins1_def)
apply (erule exE)
apply (erule conjE)+

apply (smt ir_transitive irrefl_def lem_ins_irreflexive lem_wins1_then_ins
mem_Collect_eq case_prodI push_pop_ordered wins1_def)

apply (erule disjE)
apply (metis lem_pr_ir_then_ir)

apply (metis ir_transitive)
apply (erule disjE)+
apply (erule conjE)+
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (rule disjI2)
apply (rule_tac x="b" in exI)
apply (rule conjI)
apply (assumption)
apply (erule UnE)
apply (drule w1D)
apply (erule disjE)
apply (metis (lifting) Un_def def_value is_pop_def lem_pop_pr_push_then_ir
lem_push_then_pop_exists linD mem_Collect_eq pop_irD val_order_def)
apply (erule disjE)
apply (metis UnE def_value lem_pop_then_push_exists
lem_push_then_pop_exists linD pop_irD val_order_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (smt ins_def ipr_ooo_ir lem_pop_pr_push_then_ir
lem_val_then_push_and_pop linD push_pop_ordered)
apply (metis ir_transitive lem_pop_pr_push_then_ir pop_irD)
apply (erule exE)+
apply (erule conjE)+
apply (erule UnE)+
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis def_value is_pop_def lem_push_then_pop_exists pop_irD
val_order_def)
apply (erule UnE)
apply (metis def_value lem_push_then_pop_exists linD val_order_def)
apply (metis Un_iff acyclic_irrefl irrefl_def not_ir_push_pop_ir pop_irD
pr_u_ir_acyclic r_into_trancl')
apply (assumption)
apply (erule disjE)
apply (erule conjE)+
apply (metis Un_iff ir_push_and_pop2 lem_irreflexive_then_not_equal
lem_pop_ir_irreflexive lem_pr_ir_then_ir lem_w1_push_then_pr pop_irI
push_pop_ordered)
apply (erule conjE)+
apply (erule UnE)
apply (insert lem_w1_transitive)
apply (drule_tac r="w1" and x="a" and y="b" and z="c" in transD)
apply (assumption)+
apply (drule_tac a="a" and b="c" in w1D)
apply (erule disjE)
apply (metis (lifting) Un_def def_value is_pop_def lem_pop_pr_push_then_ir
lem_pop_then_push_exists lem_push_then_pop_exists linD mem_Collect_eq
pop_irD  val_order_def)
apply (erule disjE)
apply (metis (hide_lams, no_types) Un_iff lem_val_then_push_and_pop
lem_w1_transitive linD pop_irD trans_def)
apply (erule disjE)
apply (smt UnE ins_def ipr_ooo_ir ir_transitive lem_pop_pr_push_then_ir
lem_val_then_push_and_pop linD not_ir_pop_push_ir pop_irD)
apply (metis (hide_lams, no_types) Un_iff lem_val_then_push_and_pop
lem_w1_transitive linD pop_irD trans_def)
apply (drule w1D)
apply (erule disjE)
apply (erule UnE)
apply (metis lem_pre_then_not_pop_ir not_ir_push_pop_ir pop_irI)
apply (erule UnE)
apply (smt ins_def ipr_ooo_ir linD push_pop_ordered)
apply (metis ir_transitive pop_irD)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (drule linD)
apply (erule exE)+
apply (erule conjE)+
apply (metis ins_def ipr_ooo_ir is_pop_def lem_pre_then_not_pop_ir pop_irI
push_pop_ordered)
apply (metis lem_pop_ir_transitive lem_pre_then_not_pop_ir
not_ir_push_pop_ir pop_irI r_r_into_trancl trancl_id)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (erule UnE)
apply (smt ins_def ipr_ooo_ir lem_pre_then_not_pop_ir
lem_val_then_push_and_pop linD not_ir_pop_push_ir pop_irI)
apply (metis lem_pop_ir_transitive lem_pre_then_not_pop_ir pop_irI
push_pop_ordered transD)
apply (erule exE)+
apply (erule conjE)+
apply (erule UnE)+
apply (rule disjI2)
apply (rule disjI2)
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (drule linD)+
apply (erule exE)+
apply (erule conjE)+
apply (metis (hide_lams, no_types) ins_def ipr_ooo_ir is_pop_def
lem_pre_then_not_pop_ir pop_irI push_pop_ordered)
apply (rule disjI2)
apply (rule disjI2)
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (drule linD)
apply (erule exE)+
apply (erule conjE)+
apply (smt Collect_mem_eq Un_def ins_def ipr_ooo_ir lem_pop_ir_transitive
lem_pre_then_not_pop_ir lem_val_then_push_and_pop not_ir_push_pop_ir pop_irI
r_r_into_trancl sup_idem trancl_id val_then_not_pr_def)
apply (erule UnE)
apply (rule disjI2)
apply (rule disjI2)
apply (rule disjI1)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (drule linD)
apply (erule exE)+
apply (erule conjE)+
apply (smt ins_def ipr_ooo_ir lem_pop_ir_transitive lem_pre_then_not_pop_ir
lem_val_then_push_and_pop not_ir_push_pop_ir pop_irI r_r_into_trancl
trancl_id)
apply (metis lem_pop_ir_transitive lem_pre_then_not_pop_ir pop_irI
push_pop_ordered trans_def)
apply (erule disjE)
apply (metis Un_iff acyclic_def ir_transitive lem_w1_acyclic
not_ir_push_pop_ir pop_irI r_into_trancl' r_r_into_trancl w1_def)
apply (erule disjE)
apply (erule conjE)+
apply (erule UnE)
apply (rule disjI2)+
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (drule w1D)
apply (erule disjE)
apply (metis (hide_lams, no_types) Un_iff ir_push_and_pop4
lem_push_pr_pop_then_ir lem_val_then_push_and_pop lem_w1_push_then_pr linD
not_ir_push_pop_ir pop_irD pop_irI r_into_trancl' w1_def)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)+
apply (drule_tac a="b" and b="ca" in lem_push_pr_pop_then_ir)
apply (metis Un_iff is_pop_def linD pop_irD)
apply (assumption)
apply (erule UnE)
apply (case_tac "(b, c) ∈ ir")
apply (metis (hide_lams, no_types) Un_iff pop_irI r_into_trancl' w1_def)
apply (smt ins_def ipr_ooo_ir ir_push_and_pop3 linD push_pop_ordered)
apply (drule pop_irD)
apply (erule conjE)+
apply (erule exE)
apply (erule conjE)
apply (metis (hide_lams, no_types) Un_iff ir_transitive pop_irI
r_into_trancl' w1_def)
apply (erule disjE)
apply (metis UnE def_value lem_pop_then_push_exists
lem_push_then_pop_exists linD pop_irD val_order_def)
apply (metis UnE def_value lem_pop_then_push_exists
lem_push_then_pop_exists linD pop_irD val_order_def)
apply (metis (hide_lams, no_types) Un_iff pop_irI r_into_trancl' w1_def)
by (metis Un_iff def_value ir_push_and_pop3 ir_push_and_pop4
lem_push_then_pop_exists push_pop_ordered transD val_order_def)

lemma lem_w1_ir_acyclic : "acyclic (w1 Un ir)"
apply (unfold acyclic_def)
apply (safe)
apply (drule lem_trancl_w1UnirD)
apply (safe)
apply (metis pre_irreflexive irrefl_def)
apply (metis ir_push_and_pop2 lem_irreflexive_then_not_equal
lem_pop_ir_irreflexive pop_irI)
apply (metis irrefl_def lem_pop_ir_irreflexive pop_irI)
apply (metis irrefl_def lem_pop_ir_irreflexive pop_irI)
by (metis irrefl_def lem_w1_irreflexive)

lemma lem_pr_in_w1 : "pre ⊆ w1"
by (metis UnI1 r_into_trancl' subsetI w1_def)

lemma lem_pr_then_w1 : "(a, b) ∈ pre ⟹ (a, b) ∈ w1"
by (metis lem_pr_in_w1 set_rev_mp)

lemma lem_lin_then_w1 : "(a, b) ∈ lin ⟹ (a, b) ∈ w1"
by (metis Un_iff r_into_trancl' w1_def)

lemma lem_pop_ir_then_w1 : "(a, b) ∈ pop_ir ⟹ (a, b) ∈ w1"
by (metis Un_iff r_into_trancl' w1_def)

lemma lem_same_ir : "(a, b) ∉ w1 ⟹ (b, a) ∉ w1 ⟹ is_pop a ⟹ is_pop
b ⟹ (∀ e . (e, a) ∈ ir ⟶ (e, b) ∈ ir) ∧ (∀ e . (e, b) ∈ ir ⟶ (e,
a) ∈ ir)"
by (metis ir_push_and_pop4 lem_pop_ir_then_w1 not_ir_pop_push_ir pop_irI)
```

```
lemma lem_w1_then_ir : "(a, b) ∈ w1 ⟹ is_push a ⟹ is_pop b ⟹ (a, b)
∈ ir"
by (smt Un_upper2 inf_sup_aci(5) ir_push_and_pop4 lem_push_pr_pop_then_ir
lem_trancl_w1UnirD lem_w1_push_then_pr push_pop_ordered r_into_trancl'
trancl_mono)

definition w1_pop :: "'a rel"
where
  "w1_pop = {(a, b) . is_pop a ∧ is_pop b ∧ (a, b) ∈ w1}"

lemma w1_popD : "(a, b) ∈ w1_pop ⟹ is_pop a ∧ is_pop b ∧ (a, b) ∈ w1"
apply (unfold w1_pop_def)
by (simp)

lemma w1_popI : "is_pop a ⟹ is_pop b ⟹ (a, b) ∈ w1 ⟹ (a, b) ∈
w1_pop"
apply (unfold w1_pop_def)
by (simp)

lemma w1_pop_transitive : "trans w1_pop"
by (metis lem_w1_transitive transE transI w1_popD w1_popI)

lemma w1_pop_irreflexive : "irrefl w1_pop"
by (metis acyclic_def irrefl_def lem_pr_ir_lin_acyclic w1_def w1_popD)

lemma w1_pop_antisymmetric : "antisym w1_pop"
by (metis antisymI irrefl_def transD w1_pop_irreflexive w1_pop_transitive)

lemma w1_pop_acyclic : "acyclic w1_pop"
by (metis lem_transitive_irreflexive_then_acyclic w1_pop_irreflexive
w1_pop_transitive)

lemma w1_pop_then_w1 : "(a, b) ∈ w1_pop ⟹ (a, b) ∈ w1"
by (metis w1_popD)

lemma trancl_w1_pop_then_trancl_w1 : "(a, b) ∈ trancl (w1_pop Un ir) ⟹
(a, b) ∈ trancl (w1 Un ir)"
apply (erule trancl.induct)
apply (metis Un_iff r_into_trancl' w1_pop_then_w1)
by (metis Un_iff trancl.trancl_into_trancl w1_pop_then_w1)

lemma w1_pop_ir_acyclic : "acyclic (w1_pop Un ir)"
by (metis acyclic_def lem_w1_ir_acyclic trancl_w1_pop_then_trancl_w1)

definition pop_ops :: "'a set"
where
  "pop_ops = {a . is_pop a}"

lemma pop_ops_finite : "finite pop_ops"
by (metis pop_ops_def pops_finite)

lemma only_pops_in_pop_ops : "a ∈ pop_ops ⟹ is_pop a"
by (metis mem_Collect_eq pop_ops_def)

inductive finite_pop_set :: "'a set ⇒ bool"
  where
    emptyI [simp, intro!]: "finite_pop_set {}"
  | insertI [simp, intro!]: "finite_pop_set A ⟹ is_pop a ⟹
finite_pop_set (insert a A)"

lemma only_pops_in_finite_pop_set_help : "finite_pop_set r ⟹ ∀ a . a ∉
r ∨ (is_pop a)"
apply (erule finite_pop_set.induct)
apply (metis empty_iff)
by (metis insertE)

lemma only_pops_in_finite_pop_set : "finite_pop_set r ⟹ a ∈ r ⟹
is_pop a"
by (metis only_pops_in_finite_pop_set_help)

lemma pops_are_finite_pop_set_help : "finite r ⟹ (∃ a . a ∈ r ∧
¬is_pop a) ∨ finite_pop_set r"
apply (erule finite.induct)
apply (metis emptyI)
by (metis insertCI insertI)

lemma pops_are_finite_pop_set : "finite_pop_set pop_ops"
apply (insert pop_ops_finite)
apply (drule pops_are_finite_pop_set_help)
apply (erule disjE)
apply (erule exE)
apply (erule conjE)
apply (drule only_pops_in_pop_ops)
apply metis
by assumption

inductive finite_pop_rel :: "'a rel ⇒ bool"
  where
    emptyI [simp, intro!]: "finite_pop_rel {}"
  | insertI [simp, intro!]: "finite_pop_rel A ⟹ is_pop a ⟹ is_pop b ⟹
finite_pop_rel (insert (a, b) A)"

lemma only_pops_in_finite_pop_rel_help : "finite_pop_rel r ⟹ ∀ a b. (a,
b) ∉ r ∨ (is_pop a ∧ is_pop b)"
apply (erule finite_pop_rel.induct)
apply (metis empty_iff)
by (metis Pair_inject insertE)

lemma only_pops_in_finite_pop_rel : "finite_pop_rel r ⟹ (a, b) ∈ r ⟹
is_pop a ∧ is_pop b"
by (metis only_pops_in_finite_pop_rel_help)

lemma finite_pop_rel_union : "finite_pop_rel A ⟹ finite_pop_rel B ⟹
finite_pop_rel (A Un B)"
apply (induct rule: finite_pop_rel.induct)
by simp_all

lemma pair_is_finite_pop_rel : "is_pop a ⟹ is_pop b ⟹ finite_pop_rel
{(a, b)}"
by (metis finite_pop_rel.simps)

lemma finite_pop_relI_help3 : "{a} × (insert aa A) = {a} × A Un {(a, aa)}
"
by (metis Sigma_empty1 Un_insert_right insert_times_insert
sup_bot.left_neutral sup_bot.right_neutral)
```

lemma *finite__pop_relI_help2 : "finite_pop_set A ⟹ ¬is_pop a ∨ finite_pop_rel ({a} × A)"*
apply *(erule finite_pop_set.induct)*
apply *(metis Sigma_empty2 finite_pop_rel.emptyI)*
apply *(erule disjE)*
apply *(rule disjI1)*
apply *(assumption)*
apply *(rule_tac t="{a} × insert aa A"* **and** *s="{a} × A Un {(a, aa)}"* **in** *subst)*
apply *(metis finite_pop_relI_help3)*
by *(metis finite_pop_rel_union pair_is_finite_pop_rel)*

lemma *finite_relI_help1 : "insert a A × B = A × B Un {a} × B"*
by *(metis Sigma_Un_distrib1 Un_commute insert_is_Un)*

lemma *finite_relI : "finite_pop_set A ⟹ finite_pop_set B ⟹ finite_pop_rel (A × B)"*
apply *(erule finite_pop_set.induct)*
apply *(metis Sigma_empty1 finite_pop_rel.emptyI)*
apply *(rule_tac t="insert a A × B"* **and** *s="A × B Un {a} × B"* **in** *subst)*
apply *(metis finite_relI_help1)*
apply *(rule finite_pop_rel_union)*
apply *assumption*
by *(metis finite__pop_relI_help2)*

lemma *pop_ops_rel_finite : "finite_pop_rel (pop_ops × pop_ops)"*
by *(metis finite_relI pops_are_finite_pop_set)*

lemma *all_in_pop_ops_rel : "is_pop a ⟹ is_pop b ⟹ (a, b) ∈ (pop_ops × pop_ops)"*
by *(metis SigmaI mem_Collect_eq pop_ops_def)*

lemma *w1_pop_in_pop_ops_rel : "w1_pop ⊆ (pop_ops × pop_ops)"*
apply *(rule subrelI)*
by *(metis all_in_pop_ops_rel w1_popD)*

lemma *help_irreflexive : "trans r ⟹ irrefl r ⟹ antisym r ⟹ (a, b) ∉ r ⟹ (b, a) ∉ r ⟹ a ≠ b ⟹ irrefl (trancl (insert (a, b) r))"*
by *(metis acyclic_insert acyclic_irrefl rtranclD trancl_id)*

lemma *help_antisymmetric : "trans r ⟹ irrefl r ⟹ antisym r ⟹ (a, b) ∉ r ⟹ (b, a) ∉ r ⟹ a ≠ b ⟹ antisym (trancl (insert (a, b) r))"*
by *(metis acyclic_def acyclic_insert antisym_def lem_transitive_irreflexive_then_acyclic rtrancl_eq_or_trancl trancl_id trancl_trans)*

lemma *is_pop_preserved : "is_pop a ⟹ is_pop b ⟹ ∀ c d . (c, d) ∈ r ⟶ is_pop c ∧ is_pop d ⟹ (aa, ba) ∈ (insert (a, b) r) ⟹ is_pop aa ∧ is_pop ba"*
by *(metis insert_iff prod.inject)*

lemma *r_is_pop_then_trancl_is_pop : "∀ a b . (a, b) ∈ r ⟶ is_pop a ∧ is_pop b ⟹ (c, d) ∈ trancl r ⟹ is_pop c ∧ is_pop d"*
by *(metis converse_tranclE trancl.cases)*

definition *w2 :: "'a rel ⇒ bool"*

```
where
  "w2 r ⟷ w1_pop ⊆ r ∧ (ALL a b . (a, b) ∈ pop_ops × pop_ops ⟶ a=b ∨
(a, b) ∈ r ∨ (b, a) ∈ r) ∧ trans r ∧ irrefl r ∧ antisym r ∧ (∀ a b . (a,
b) ∈ r ⟶ is_pop a ∧ is_pop b)"

lemma lem_exists_total_closure : "(∃ r . w2 r)"
apply (unfold w2_def)
apply (insert pop_ops_rel_finite)
apply (erule finite_pop_rel.induct)
apply (metis empty_iff equalityE w1_popD w1_pop_antisymmetric
w1_pop_irreflexive w1_pop_transitive)
apply (erule exE)
apply (case_tac "a=b")
apply (rule_tac x="r" in exI)
apply (erule conjE)+
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (rule allI)+
apply (rule impI)
apply (unfold insert_def)
apply (erule UnE)
apply (metis (mono_tags) mem_Collect_eq prod.inject)
apply metis
apply metis
apply (case_tac "(a, b) ∈ r")
apply (rule_tac x="r" in exI)
apply (rule conjI)
apply metis
apply (rule conjI)
apply (rule allI)+
apply (rule impI)
apply (erule UnE)
apply (metis (full_types) mem_Collect_eq)
apply metis
apply metis
apply (case_tac "(b, a) ∈ r")
apply (rule_tac x="r" in exI)
apply (rule conjI)
apply metis
apply (rule conjI)
apply (rule allI)+
apply (rule impI)
apply (erule UnE)

apply (metis (mono_tags) mem_Collect_eq prod.inject)
apply metis
apply metis
apply (rule_tac x="trancl (insert (a, b) r)" in exI)
apply (rule conjI)
apply (metis r_into_trancl' subsetI subset_insertI2 subset_trans)
apply (erule conjE)+
apply (rule conjI)
apply (rule allI)+
apply (rule impI)
apply (erule UnE)
apply (metis (lifting) insert_compr mem_Collect_eq r_into_trancl')
apply (metis insert_iff r_into_trancl')
apply (rule conjI)
apply (metis trans_trancl)
apply (rule conjI)
apply (rule help_irreflexive)
apply (assumption)+
apply (rule conjI)
apply (rule help_antisymmetric)
apply (assumption)+
apply (rule allI)+
apply (rule impI)
apply (rule_tac r="insert (a, b) r" and c="aa" and d="ba" in
r_is_pop_then_trancl_is_pop)
apply (metis is_pop_preserved)
by assumption

lemma lem_w2_then_pop : "w2 r ⟹ (a, b) ∈ r ⟹ is_pop a ∧ is_pop b"
by (metis w2_def)

lemma lem_w2_pop_total : "w2 r ⟹ is_pop a ⟹ is_pop b ⟹ a ≠ b ⟹
(a, b) ∈ r ∨ (b, a) ∈ r"
by (metis all_in_pop_ops_rel w2_def)

lemma lem_w2_antisymmetric : "w2 r ⟹ antisym r"
by (metis w2_def)

lemma lem_w2_acyclic : "w2 r ⟹ acyclic r"
by (metis lem_transitive_irreflexive_then_acyclic w2_def)

lemma lem_w2_transitive : "w2 r ⟹ trans r"
by (metis w2_def)

lemma lem_preUw2_then_push_or_pop1 : "w2 r ⟹ (a, b) ∈ pre Un r ⟹
is_push a ∨ is_pop a"
by (metis Un_iff lem_push_or_pop1 lem_w2_then_pop)

lemma lem_preUw2_then_push_or_pop2 : "w2 r ⟹ (a, b) ∈ pre Un r ⟹
is_push b ∨ is_pop b"
by (metis Un_iff lem_push_or_pop2 lem_w2_then_pop)

lemma lem_pre_w2D_help : "w2 r ⟹ (a, b) ∈ trancl (pre Un r) ⟹
    is_push a ∨ (is_pop b ∧ (a, b) ∈ r) ∨ (a, b) ∈ pre ∨ (∃ c. is_pop c
∧ (a, c) ∈ r ∧ (c, b) ∈ pre)"
apply (erule trancl.induct)
apply (metis Un_iff lem_w2_then_pop)
apply (erule disjE)
apply metis
apply (frule_tac a="b" and b="c" in lem_preUw2_then_push_or_pop2)
apply (assumption)
apply (erule disjE)
apply (erule conjE)
apply (erule disjE)
apply (erule UnE)
apply metis
apply (metis ir_push_and_pop3 ir_push_and_pop4 lem_w2_then_pop
push_pop_ordered)
apply (metis Un_iff lem_w2_transitive trans_def)
apply (erule disjE)
apply (metis pre_irreflexive UnE abcd_def def_value
lem_irreflexive_then_not_equal lem_pop_then_push_exists
lem_push_then_pop_exists lem_w2_then_pop pr_abcd val_order_def)
apply (erule disjE)
apply (frule lem_push_or_pop1)
apply (erule disjE)
apply metis
apply (frule lem_push_or_pop2)
apply (erule disjE)
apply (metis pre_irreflexive UnE abcdD ir_push_and_pop4
lem_irreflexive_then_not_equal lem_pop_pr_push_then_ir lem_w2_then_pop
pr_abcd)
apply (smt UnE lem_pr_then_w1 lem_w2_transitive subsetD transD w1_popI
w2_def)
by (smt pre_transitive UnE lem_pr_then_w1 lem_w2_transitive subsetD transD
w1_popI w2_def)


lemma lem_pre_w2D : "w2 r ⟹ (a, b) ∈ trancl (pre Un r) ⟹ is_pop a ⟹
is_push b ⟹ (a, b) ∈ pre ∨ (∃ c. is_pop c ∧ (a, c) ∈ r ∧ (c, b) ∈
pre)"
by (metis lem_irreflexive_then_not_equal lem_pop_ir_irreflexive
lem_pre_w2D_help not_ir_pop_push_ir pop_irI)

lemma lem_pre_w2D2_help : "w2 r ⟹ (a, b) ∈ trancl (pre Un r) ⟹
  is_pop a  ∨ (a, b) ∈ pre ∨ (is_push a ∧ (∃ c . is_pop c  ∧ (a, c) ∈ pre
∧ (c, b) ∈ trancl (pre Un r)))"
apply (erule trancl.induct)
apply (metis Un_iff lem_w2_then_pop)
by (metis Un_iff pre_transitive lem_push_or_pop1 lem_w2_then_pop
trancl.r_into_trancl trancl.trancl_into_trancl trans_def)

lemma lem_pre_w2D2 : "w2 r ⟹ (a, b) ∈ trancl (pre Un r) ⟹ is_push a
⟹ (a, b) ∈ pre ∨ (∃ c. is_pop c ∧ (a, c) ∈ pre ∧ (c, b) ∈ trancl (pre
Un r))"
by (metis (hide_lams, no_types) def_value lem_pop_then_push_exists
lem_pre_w2D2_help lem_push_then_pop_exists val_order_def)


lemma lem_tranclpreUw2_then_push_or_pop1 : "w2 r ⟹ (a, b) ∈ trancl (pre
Un r) ⟹ is_push a ∨ is_pop a"
by (metis lem_pre_w2D2_help lem_push_or_pop1)

lemma lem_tranclpreUw2_then_push_or_pop2 : "w2 r ⟹ (a, b) ∈ trancl (pre
Un r) ⟹ is_push b ∨ is_pop b"
by (metis (hide_lams, no_types) lem_preUw2_then_push_or_pop2 tranclE)


lemma lem_trancl_pre_w2D : "w2 r ⟹ (a, b) ∈ trancl (pre Un r) ⟹
is_push a ∨ is_push b ∨ (a, b) ∈ r"
apply (erule trancl.induct)
apply (erule UnE)
apply (frule lem_push_or_pop1)
apply (frule lem_push_or_pop2)
apply (erule disjE)+
apply metis
apply metis
apply (erule disjE)
apply metis
apply (metis lem_pr_then_w1 set_rev_mp w1_popI w2_def)
apply metis
apply (erule disjE)
apply metis
apply (erule disjE)
apply (frule_tac a="a" and b="b" in lem_tranclpreUw2_then_push_or_pop1)
apply (assumption)+
apply (erule disjE)
apply metis
apply (frule_tac a="a" and b="b" in lem_pre_w2D)
apply (assumption)+
apply (erule disjE)
apply (erule UnE)
apply (frule_tac a="b" and b="c" in lem_push_or_pop2)
apply (erule disjE)
apply metis
apply (metis (hide_lams, no_types) pre_irreflexive abcd_def irrefl_def
lem_pr_then_w1 pr_abcd subsetD w1_popI w2_def)
apply (frule_tac a="b" and b="c" in lem_w2_then_pop)
apply (assumption)
apply (metis ir_push_and_pop4 lem_pop_pr_push_then_ir)
apply (frule_tac a="b" and b="c" in lem_preUw2_then_push_or_pop2)
apply (assumption)
apply (erule disjE)
apply metis
apply (smt pre_transitive UnE lem_pr_then_w1 lem_w2_transitive
r_r_into_trancl subsetD trancl_id w1_popI w2_def)
apply (frule_tac a="b" and b="c" in lem_preUw2_then_push_or_pop2)
apply (assumption)
apply (erule disjE)
apply metis
by (smt UnE lem_pr_in_w1 lem_w2_transitive r_r_into_trancl subsetD
trancl_id w1_popI w2_def)

lemma lem_w1_in_w2 : "w2 r ⟹ (a, b) ∈ w1 ⟹ is_pop a ⟹ is_pop b ⟹
(a, b) ∈ r"
by (metis set_rev_mp w1_popI w2_def)

lemma lem_not_w1_then_same_ir : "w2 r ⟹ is_pop a ⟹ is_pop b ⟹ (a,
b) ∈ r ⟹ (a, b) ∉ w1 ⟹ (e, a) ∈ ir ⟹ (e, b) ∈ ir"
apply (rotate_tac -2)
apply (erule contrapos_np)
apply (frule ir_push_and_pop4)
apply (assumption)
apply (drule not_ir_push_pop_ir)
apply (assumption)+
apply (drule_tac a="b" and b="a" and c="e" in pop_irI)
apply (assumption)+
apply (drule lem_pop_ir_then_w1)
by (metis acyclic_def ir_push_and_pop2 lem_w1_in_w2 lem_w2_acyclic
r_r_into_trancl)

lemma lem_not_w1_then_same_ir2 : "w2 r ⟹ is_pop a ⟹ is_pop b ⟹ (a,
b) ∈ r ⟹ (a, b) ∉ w1 ⟹ (a, e) ∈ ir ⟹ (b, e) ∈ ir"
by (metis ir_push_and_pop3 lem_pop_ir_then_w1 not_ir_pop_push_ir pop_irI)
```

lemma *lem_not_w1_then_same_ir3 : "w2 r ⟹ is_pop a ⟹ is_pop b ⟹ (a, b) ∈ r ⟹ (a, b) ∉ w1 ⟹ (b, e) ∈ ir ⟹ (a, e) ∈ ir"*
by *(metis ir_push_and_pop3 lem_irreflexive_then_not_equal lem_not_w1_then_same_ir lem_pop_ir_irreflexive pop_irI push_pop_ordered)*

lemma *lem_pr_in_w2 : "w2 r ⟹ is_pop a ⟹ is_pop b ⟹ (a, b) ∈ pre ⟹ (a, b) ∈ r"*
by *(metis lem_w1_in_w2 lem_pr_then_w1)*

lemma *help_me : "w2 r ⟹ (a, b) ∈ r ⟹ (a, b) ∈ w1 ∨ ((c, a) ∈ ir ⟶ (c, b) ∈ ir)"*
by *(metis lem_not_w1_then_same_ir lem_w2_then_pop)*

lemma *lem_prUw2_then_w1Uir : "w2 r ⟹ (a, b) ∈ trancl (pre Un r) ⟹ (is_pop a ∧ is_pop b ∧ (a, b) ∈ r) ∨ (a, b) ∈ trancl (w1 Un ir)"*
apply *(erule trancl.induct)*
apply *(frule_tac a="a"* **and** *b="b"* **in** *lem_preUw2_then_push_or_pop1)*
apply *(assumption)*
apply *(frule_tac a="a"* **and** *b="b"* **in** *lem_preUw2_then_push_or_pop2)*
apply *(assumption)*
apply *(erule disjE)+*
apply *(metis UnE inf_sup_ord(3) lem_pr_then_w1 r_into_trancl' trancl_mono w2_def)*
apply *(metis UnE Un_upper2 lem_push_pr_pop_then_ir r_into_trancl' trancl_mono w2_def)*
apply *(erule disjE)*
apply *(metis Un_iff lem_pop_pr_push_then_ir r_into_trancl' w2_def)*
apply *(metis UnE lem_pr_in_w2)*
apply *(erule disjE)+*
apply *(erule conjE)*
apply *(frule_tac a="b"* **and** *b="c"* **in** *lem_preUw2_then_push_or_pop2)*
apply *(assumption)*
apply *(erule disjE)*
apply *(erule UnE)*
apply *(frule_tac a="c"* **and** *b="a"* **in** *push_pop_ordered)*
apply *(assumption)+*
apply *(erule disjE)*
apply *(erule conjE)*
apply *(frule_tac a="b"* **and** *b="c"* **in** *lem_pop_pr_push_then_ir)*
apply *(assumption)+*
apply *(case_tac "(a, b) ∈ w1")*
apply *(metis Un_iff r_r_into_trancl)*
apply *(metis (hide_lams, no_types) ins_def insI ipr_ooo_ir is_pop_def is_push_def lem_not_w1_then_same_ir lem_val_then_ir val_then_pr)*
apply *(metis Un_iff r_into_trancl')*
apply *(erule conjE)*
apply *(metis lem_pop_ir_then_w1 lem_pre_then_not_pop_ir lem_w1_push_then_pr lem_w2_then_pop pop_irI push_pop_ordered)*
apply *(metis (lifting) def_value lem_pop_then_push_exists lem_push_then_pop_exists lem_trancl_pre_w2D lem_w2_transitive r_r_into_trancl trancl.r_into_trancl trancl_id val_order_def)*
apply *(erule UnE)*
apply *(metis (hide_lams, no_types) UnCI lem_pr_in_w1 set_rev_mp trancl.trancl_into_trancl)*
apply *(frule_tac a="a"* **and** *b="b"* **in** *lem_tranclpreUw2_then_push_or_pop1)*
apply *(assumption)*
apply *(erule disjE)*
apply *(frule_tac a="b"* **and** *b="c"* **in** *lem_w2_then_pop)*

```
apply (assumption)+
apply (erule conjE)
apply (frule_tac a="b" and b="c" and c="a" in help_me)
apply (assumption)
apply (erule disjE)
apply (metis (hide_lams, no_types) Un_iff trancl.simps)
apply (smt UnI1 Un_commute def_value lem_pop_then_push_exists
lem_push_then_pop_exists lem_trancl_w1UnirD trancl.r_into_trancl
val_order_def)
by (metis Un_iff def_value lem_pop_then_push_exists
lem_push_then_pop_exists lem_trancl_pre_w2D trancl.trancl_into_trancl
val_order_def w2_def)



lemma lem_w3_irreflexive : "w2 r ⟹ irrefl (trancl (pre Un r))"
by (metis acyclic_def irrefl_def lem_pop_ir_irreflexive
lem_prUw2_then_w1Uir lem_trancl_pre_w2D lem_w1_ir_acyclic lem_w2_acyclic
lem_w2_transitive pop_irI push_pop_ordered trancl_id)

lemma lem_w3_antisymmetric : "w2 r ⟹ antisym (trancl (pre Un r))"
by (metis antisymI irrefl_def lem_w3_irreflexive trancl_trans)

lemma val_then_not_w3 : "w2 r ⟹ (ia, ra) ∈ val ⟹ (ra, ia) ∉ trancl
(pre Un r)"
by (metis Un_iff lem_irreflexive_then_not_equal lem_w3_irreflexive
trancl.simps val_then_pr)

lemma lem_witness_wellformed : "w2 r ⟹ wellformed_history (trancl (pre
Un r)) val f"
apply (rule wellformed_historyI)
apply (metis trans_trancl)
apply (metis lem_w3_antisymmetric)
apply (metis lem_w3_irreflexive)
apply (metis operations_countable)
apply (metis operations_finite)
apply (metis def_push_once)
apply (metis def_pop_once)
apply (metis def_value)
apply (metis is_pop_def is_push_def lem_tranclpreUw2_then_push_or_pop1
lem_tranclpreUw2_then_push_or_pop2)
by (metis val_then_not_prI val_then_not_w3)

lemma lem_help_push_or_pop: "w2 r ⟹ (a, b) ∈ ((pre ∪ r)^+ ∪ ir)^+ ⟹
is_push a ∨ is_pop a"
apply (erule trancl.induct)
apply (metis UnE ir_push_and_pop lem_witness_wellformed
wellformed_history.lem_push_or_pop1)
by metis

lemma lem_w3_acyclic_help : "w2 r ⟹ (a, b) ∈ trancl ( (trancl (pre Un
r)) Un ir) ⟹ (is_pop a ∧ is_pop b ∧ (a, b) ∈ r) ∨ (a, b) ∈ trancl (w1
Un ir)"
apply (erule trancl.induct)
apply (metis Un_iff lem_irreflexive_then_not_equal lem_pop_ir_irreflexive
lem_prUw2_then_w1Uir lem_trancl_pre_w2D pop_irI push_pop_ordered
trancl.r_into_trancl)
apply (erule disjE)
apply (erule conjE)+
apply (erule UnE)
apply (frule_tac c="(a, b)" and A="pre" and B="r" in UnI2)
apply (smt lem_prUw2_then_w1Uir lem_trancl_pre_w2D trancl_into_trancl2
w2_def)
apply (case_tac "(a, b) ∈ w1")
apply (metis Un_iff r_r_into_trancl)
apply (frule_tac r="r" and a="a" and b="b" and e="c" in
lem_not_w1_then_same_ir3)
apply (assumption)+
apply (metis UnCI r_into_trancl')
apply (erule UnE)
apply (frule_tac a="b" and b="c" and r="r" in lem_prUw2_then_w1Uir)
apply (assumption)
apply (erule disjE)
apply (erule conjE)+
apply (frule_tac a="a" and b="b" in lem_help_push_or_pop)
apply (assumption)
apply (erule disjE)+
apply (frule_tac a="a" and b="b" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (case_tac "(b, c) ∈ w1")
apply (metis Un_iff trancl.trancl_into_trancl)
apply (frule_tac r="r" and a="b" and b="c" and e="a" in
lem_not_w1_then_same_ir)
apply (assumption)+
apply (metis Un_iff trancl.simps)
apply (smt insI ins_def ipr_ooo_ir ir_transitive lem_pop_then_push_exists
lem_push_pr_pop_then_ir lem_push_then_pop_exists lem_trancl_w1UnirD
lem_w1_then_ir val_then_pr)
apply (smt UnCI help_me lem_prUw2_then_w1Uir lem_trancl_w1UnirD
lem_w1_in_w2 not_ir_pop_push_ir trancl.r_into_trancl trancl_into_trancl2
trancl_trans)
apply (metis trancl_trans)
by (metis Un_iff trancl.trancl_into_trancl)

lemma lem_w3_acyclic : "w2 r ⟹ acyclic ((trancl (pre Un r)) ∪ ir)"
apply (unfold acyclic_def)
apply (rule allI)
apply (rule notI)
by (metis acyclic_def irrefl_trancl_rD lem_w1_ir_acyclic lem_w2_acyclic
lem_w3_acyclic_help)

lemma lem_w3_pop_ordered : "w2 r ⟹ is_pop a ⟹ is_pop b ⟹ a ≠ b ⟹
(a, b) ∈ trancl (pre Un r) ∨ (b, a) ∈ trancl (pre Un r)"
by (metis Un_iff lem_w2_pop_total trancl.simps)



lemma lem_w3_then_ins : "w2 r ⟹ is_push a ⟹ is_push b ⟹ (a, b) ∈
trancl (pre Un r) ⟹ (a, b) ∈ ins"
by (smt insI ins_def ipr_ooo ir_push_and_pop3 ir_push_and_pop4
lem_prUw2_then_w1Uir lem_pr_then_ins lem_pre_w2D2 lem_push_then_pop_exists
lem_trancl_w1UnirD push_pop_ordered val_then_pr)

lemma lem_w3ins_then_ins : "w2 r ⟹ is_push a ⟹ is_push b ⟹ is_pop c
⟹ (a, c) ∈ trancl (pre Un r) ⟹ (c, b) ∈ ir ⟹ (a, b) ∈ ins"
```

```
apply (frule_tac a="a" and b="c" in lem_pre_w2D2)
apply (assumption)+
apply (erule disjE)
apply (metis insI)
apply (erule exE)
apply (erule conjE)+
apply (frule_tac a="ca" and b="c" in lem_trancl_pre_w2D)
apply (assumption)
apply (erule disjE)
apply (metis lem_irreflexive_then_not_equal lem_pop_ir_irreflexive
not_ir_pop_push_ir pop_irI)
apply (erule disjE)
apply (metis def_value is_pop_def is_push_def val_order_def)
apply (case_tac "(ca, c) ∈ w1")
apply (frule_tac a="b" and b="ca" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (frule_tac a="c" and b="ca" and c="b" in pop_irI)
apply (assumption)+
apply (metis acyclic_def lem_pop_ir_then_w1 lem_w1_in_w2 lem_w2_acyclic
r_r_into_trancl)
apply (metis insI)
by (metis insI lem_not_w1_then_same_ir3)

lemma lem_wins_then_ins : "w2 r ⟹ (a, b) ∈ (ins_order (trancl (pre Un
r)) val ir) ⟹ (a, b) ∈ ins"
apply (unfold ins_order_def)
apply (simp add:set_eq_iff)
apply (erule conjE)+
apply (erule disjE)
apply (metis isPush_def is_pushI lem_w3_then_ins)
by (metis ir_push_and_pop2 isPush_def is_pushI lem_w3ins_then_ins)

lemma lem_w3_ooo : "w2 r ⟹
          (ia, ra) ∈ val ⟹
          (ib, rb) ∈ val ⟹
          (ia, ib) ∈ ins_order ((pre ∪ r)⁺) val ir ⟹ (ib, ra) ∈ ir
⟹ (rb, ra) ∈ (pre ∪ r)⁺"
apply (frule_tac a="ia" and b="ib" in lem_wins_then_ins)
apply (assumption)
apply (frule_tac ia="ib" and ib="ia" and ra="rb" and rb="ra" in linI)
apply (assumption)+
apply (frule lem_lin_then_w1)
apply (frule_tac a="rb" and b="ra" in lem_w1_in_w2)
apply (assumption)
apply (metis is_pop_def)
apply (metis is_pop_def)
by (metis Un_iff r_into_trancl')

lemma lem_pr_in_w3 : "w2 r ⟹ (a, b) ∈ pre ⟹ (a, b) ∈ trancl (pre Un
r)"
by (metis Un_iff r_into_trancl')

theorem remove_relaxed_stack_linearizable : "∃ prt .( pre ⊆ prt ∧
insert_pr_relaxed_stack prt val f ir)"
apply (insert lem_exists_total_closure)
apply (erule exE)
apply (rule_tac x="trancl (pre Un r)" in exI)
apply (rule conjI)
apply (metis Un_iff r_into_trancl' subsetI)
apply (unfold insert_pr_relaxed_stack_def)
apply (rule conjI)
apply (metis lem_witness_wellformed)
apply (unfold insert_pr_relaxed_stack_axioms_def)
apply (rule conjI)+
apply (metis lem_pr_in_w3 val_then_pr)
apply (metis lem_w3_pop_ordered)
apply (rule conjI)
apply (rule ir_alternating)
apply (rule conjI)
apply (metis lem_w3_acyclic)
apply (rule allI)+
by (metis lem_w3_ooo)

end

theorem remove_relaxed_stack_linearizable : "remove_relaxed_stack pre val f
ir ⟹ linearizable_non_empty_stack pre val"
apply (frule remove_relaxed_stack.remove_relaxed_stack_linearizable)
apply (erule exE)
apply (erule conjE)
apply (drule insert_pr_relaxed_stack.insert_pr_relaxed_stack_linearizable)
apply (erule exE)
apply (erule conjE)
apply (drule insert_relaxed_stack.insert_relaxed_stack_linearizable)
apply (erule exE)
apply (erule conjE)
apply (rule_tac prt="prtb" and f="f" in linearizable_non_empty_stackI)
apply (drule remove_relaxed_stackD)
apply (erule conjE)+
apply (drule wellformed_historyD)
apply (metis conc_historyI)
apply (metis subset_trans)
by (assumption)

end
theory insert_pr_relaxed_stack
imports Main insert_relaxed_stack
begin

locale insert_pr_relaxed_stack = wellformed_history +
  fixes ir :: "'a rel"
  assumes val_then_pr: "(a, b) ∈ val ⟹ (a, b) ∈ pre"
  assumes pop_ordered: "is_pop a ⟹ is_pop b ⟹a≠b⟹(a, b) ∈ pre ∨
(b,a)∈pre"
  assumes ir_alternating: "alternating val ir"
  assumes pr_u_ir_acyclic: "acyclic (pre ∪ ir)"
  assumes ipr_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈
(ins_order pre val ir) ⟹ (ib, ra) ∈ ir ⟹ (rb, ra) ∈ pre"

context insert_pr_relaxed_stack
begin

lemma push_pop_ordered : "is_push a ⟹is_pop b ⟹ (a, b) : ir ∨ (b,a) ∈
ir"
```

```
by (metis alternating_def ir_alternating isPop_def isPush_def is_popD
is_pushD)

lemma ir_push_and_pop : "(a, b) ∈ ir ⟹ (is_push a ∧ is_pop b) ∨ (is_pop
a ∧ is_push b)"
by (metis alternating_def ir_alternating isPop_def isPush_def is_pop_def
is_push_def)

definition ins :: "'a rel"
where
  "ins = ins_order pre val ir"

lemma lem_pr_then_ins : " is_push a ⟹ is_push b ⟹ (a, b) ∈ pre ⟹
(a, b) ∈ ins"
apply (unfold ins_def)
apply (unfold ins_order_def)
apply (simp add: set_eq_iff)
by (metis isPush_def is_pushD)

lemma insI : "is_push a ⟹ is_push b ⟹ (a, c) ∈ pre ⟹ (c, b) ∈ ir
⟹ (a, b) ∈ ins"
by (metis ins_def ins_orderI2 isPush_def is_pushD)

lemma ir_push_and_pop1 : "(a, b) ∈ ir ⟹ is_push a ⟹ is_pop b"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma ir_push_and_pop2 : "(a, b) ∈ ir ⟹ is_push b ⟹ is_pop a"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma ir_push_and_pop3 : "(a, b) ∈ ir ⟹ is_pop a ⟹ is_push b"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma ir_push_and_pop4 : "(a, b) ∈ ir ⟹ is_pop b ⟹ is_push a"
by (metis def_value ir_push_and_pop is_push_def lem_pop_then_push_exists
val_order_def)

lemma lem_val_then_ir : "(a, b) ∈ val ⟹ (a, b) ∈ ir"
apply (frule val_then_pr)
apply (frule lem_val_then_push_and_pop)
apply (erule conjE)
apply (frule_tac a="a" and b="b" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (assumption)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

lemma lem_push_pr_pop_then_ir : " is_push a ⟹ is_pop b ⟹ (a, b) ∈ pre
⟹ (a, b) ∈ ir"
apply (frule_tac a="a" and b="b" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (assumption)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

lemma not_ir_push_pop_ir : "(a, b) ∉ ir ⟹ is_push a ⟹ is_pop b ⟹
(b, a) ∈ ir"
by (metis push_pop_ordered)

lemma not_ir_pop_push_ir : "(a, b) ∉ ir ⟹ is_pop a ⟹ is_push b ⟹
(b, a) ∈ ir"
by (metis push_pop_ordered)

lemma lem_pop_pr_push_then_ir : " is_pop a ⟹ is_push b ⟹ (a, b) ∈ pre
⟹ (a, b) ∈ ir"
apply (frule_tac a="b" and b="a" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI1)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

inductive_set wir :: "'a rel"
where
  wir_intro   : "(a, b) ∈ pre ⟹ is_push a ⟹ is_pop b ⟹ (a, b) ∈ wir"

| wir_step: "(∃ix rx ra. (ix, rx) ∈ val ∧
                         (a, ra) ∈ val ∧
                         is_pop b ∧
                         (ix, b) ∈ wir ∧
                         (rx, ra) ∈ pre ∧
                         (b, rx) ∈ pre ∧
                         (a, rx) ∈ wir) ⟹ (a, b) ∈ wir"


lemma lem_wir_pr_then_wir_help : "(a, b) ∈ wir ⟹
    is_push c ∨ (b, c) ∉ pre ∨
   (is_pop c ∧ (b, c) ∈ pre ∧ (a, c) ∈ wir)"
apply (insert pre_transitive)
apply (erule wir.induct)
apply (metis lem_push_or_pop2 transE wir.wir_intro)
apply (erule exE)+
apply (erule conjE)+
apply (erule disjE)
apply (simp)
apply (erule disjE)
apply metis
by (metis lem_val_then_push_and_pop pop_ordered wir.wir_step)

lemma lem_wir_pr_then_wir : "(a, b) ∈ wir ⟹ is_pop c ⟹ (b, c) ∈ pre
⟹ (a, c) ∈ wir"
```

```
by (metis def_value lem_wir_pr_then_wir_help lem_pop_then_push_exists
lem_push_then_pop_exists val_order_def)


lemma wir_right_rec : "(ix, rx) ∈ val ⟹ (a, ra) ∈ val ⟹ is_pop b ⟹

             (ix, b) ∈ pre ⟹ (rx, ra) ∈ pre ⟹ (a, rx) ∈ wir ⟹ (a,
b) ∈ wir"
by (smt lem_val_then_push_and_pop lem_wir_pr_then_wir pop_ordered
wir.wir_intro wir.wir_step)

lemma wir_left_rec : "(ix, rx) ∈ val ⟹ (a, ra) ∈ val ⟹ is_pop b ⟹
                     (ix, b) ∈ wir ⟹ (rx, ra) ∈ pre ⟹ (a, rx) ∈ pre
⟹ (a, b) ∈ wir"
by (smt def_pop_once ir_push_and_pop1 is_push_def lem_val_then_ir
lem_val_then_push_and_pop lem_wir_pr_then_wir pop_once_def pop_ordered
val_then_pr wir.simps wir.wir_intro wir.wir_step wir_right_rec)

lemma wir_left_right_rec : "(ix, rx) ∈ val ⟹ (a, ra) ∈ val ⟹ is_pop b
⟹ (ix, b) ∈ wir ⟹ (rx, ra) ∈ pre ⟹ (a, rx) ∈ wir ⟹ (a, b) ∈ wir"
by (metis (lifting, no_types) lem_val_then_push_and_pop lem_wir_pr_then_wir
pop_ordered wir.wir_step)


lemma wirD : "(ia, rb) ∈ wir ⟹ (ia, rb) ∈ pre ∨
                (∃ix rx ra. (ix, rx) ∈ val ∧
                            (ia, ra) ∈ val ∧
                            is_pop rb ∧
                            (ix, rb) ∈ wir ∧
                            (rx, ra) ∈ pre ∧
                            (rb, rx) ∈ pre ∧
                            (ia, rx) ∈ wir)"
by (metis wir.simps)

inductive_set lwir :: "'a rel"
where
  lwir_intro   : "(a, b) ∈ pre ⟹ is_push a ⟹ is_pop b ⟹ (a, b) ∈
lwir"
| lwir_step: "(∃ix rx ra. (ix, rx) ∈ val ∧
                          (a, ra) ∈ val ∧
                          is_pop b ∧
                          (ix, b) ∈ lwir ∧
                          (rx, ra) ∈ pre ∧
                          (b, rx) ∈ pre ∧
                          (a, rx) ∈ pre) ⟹ (a, b) ∈ lwir"

lemma lem_lwir_then_wir : "(a, b) ∈ lwir ⟹ (a, b) ∈ wir"
apply (erule lwir.induct)
apply (metis wir.wir_intro)
by (metis wir_left_rec)

lemma lem_lwir_left_grow : "(ib, rc) ∈ lwir ⟹ (∃ rb ic.
                            (ia, ra) ∈ val ∧ (ib, rb) ∈ val ∧ (ic, rc) ∈
val ∧
                            (rc, rb) ∈ pre ∧ (rb, ra) ∈ pre ∧
                            (ia, rb) ∈ pre) ⟶  (ia, rc) ∈ lwir"
by (metis lem_val_then_push_and_pop lwir.lwir_step)
```

```
lemma lem_lwir_trans : "(ia, rb) ∈ lwir ⟹ (∃ ra ib . (ic, rc) ∈ val ∧
   (ia, ra) ∈ val ∧ (ib, rb) ∈ val ∧ (ib, rc) ∈ lwir ∧ (rc, rb) ∈ pre ∧
   (rb, ra) ∈ pre) ⟶ (ia, rc) ∈ lwir"
apply (erule lwir.induct)
apply (smt lem_lwir_left_grow)
by (smt pre_transitive def_pop_once lem_lwir_left_grow pop_once_def transE)

lemma lem_wir_then_lwir : "(a, b) ∈ wir ⟹ (a, b) ∈ lwir"
apply (erule wir.induct)
apply (metis lwir.lwir_intro)
by (metis lem_pop_then_push_exists lem_lwir_trans)

inductive_set rwir :: "'a rel"
where
  rwir_intro   : "(a, b) ∈ pre ⟹ is_push a ⟹ is_pop b ⟹ (a, b) ∈
rwir"
| rwir_step: "(∃ix rx ra. (ix, rx) ∈ val ∧
                          (a, ra) ∈ val ∧
                          is_pop b ∧
                          (ix, b) ∈ pre ∧
                          (rx, ra) ∈ pre ∧
                          (b, rx) ∈ pre ∧
                          (a, rx) ∈ rwir) ⟹ (a, b) ∈ rwir"

lemma lem_rwir_then_wir : "(a, b)∈rwir ⟹ (a, b) ∈ wir"
apply (erule rwir.induct)
apply (metis wir.wir_intro)
by (metis wir_right_rec)



lemma lem_left_grow : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ic, rc) ∈
val ⟹
                        (ia, rb) ∈ wir ⟹ (ic, ra) ∈ pre ⟹ (ra, rc) ∈
pre ⟹ (ic, rb) ∈ wir"
by (metis lem_val_then_push_and_pop wir_left_rec)

lemma lem_right_grow : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ic, rc) ∈
val ⟹ (rb, ra) ∈ pre ⟹
                        (ia, rb) ∈ wir ⟹ (ib, rc) ∈ pre ⟹ (rc, rb) ∈
pre ⟹ (ia, rc) ∈ wir"
by (metis lem_val_then_push_and_pop wir_right_rec)

lemma lem_rwir_right_grow : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ic,
rc) ∈ val ⟹ (rc, rb) ∈ pre ⟹
                        (ia, rb) ∈ rwir ⟹ (ib, rc) ∈ pre ⟹ (rb, ra) ∈
pre ⟹ (ia, rc) ∈ rwir"
by (metis lem_val_then_push_and_pop rwir.rwir_step)

lemma lem_rwir_left_grow : "(ib, rc) ∈ rwir ⟹ (∃ rb ic.
                           (ia, ra) ∈ val ∧ (ib, rb) ∈ val ∧ (ic, rc) ∈
val ∧
                           (rc, rb) ∈ pre ∧ (rb, ra) ∈ pre ∧
                           (ia, rb) ∈ pre) ⟶  (ia, rc) ∈ rwir"
apply (erule rwir.induct)
apply (rename_tac ib rc)
apply (rule impI)
apply (erule exE)+
apply (erule conjE)+
apply (drule_tac a="ia" and b="rb" in rwir.rwir_intro)
apply (metis lem_val_then_push_and_pop)
apply (metis is_pop_def)
apply (metis rwir.rwir_step)
apply (rename_tac ib rc)
apply (safe)
apply (rename_tac rb rbb ic)
apply (metis def_pop_once pop_once_def)
by (smt pre_transitive def_pop_once lem_rwir_right_grow pop_once_def
transE)

lemma lem_rwir : "(ia, rb) ∈ rwir ⟹ (ia, rb) ∈ pre ∨
                  (∃ix rx ra. (ix, rx) ∈ val ∧
                          (ia, ra) ∈ val ∧
                          is_pop rb ∧
                          (ix, rb) ∈ pre ∧
                          (rx, ra) ∈ pre ∧
                          (rb, rx) ∈ pre ∧
                          (ia, rx) ∈ rwir)"
by (metis rwir.simps)

lemma lem_rwir_trans : "(ib, rc) ∈ rwir ⟹ (∃ rb ic . (ia, ra) ∈ val ∧
   (ib, rb) ∈ val ∧ (ic, rc) ∈ val ∧ (ia, rb) ∈ rwir ∧ (rb, ra) ∈ pre ∧
   (rc, rb) ∈ pre) ⟶ (ia, rc) ∈ rwir"
apply (erule rwir.induct)
apply (rename_tac ib rc)
apply (smt lem_rwir_right_grow)
by (smt pre_transitive def_pop_once lem_rwir_right_grow pop_once_def
transE)

lemma lem_wir_then_rwir : "(a, b) ∈ wir ⟹ (a, b) ∈ rwir"
apply (erule wir.induct)
apply (metis rwir.rwir_intro)
by (metis lem_pop_then_push_exists lem_rwir_trans)

lemma lem_rwir_then_push_and_pop : "(a, b) ∈ rwir ⟹ is_push a ∧ is_pop
b"
by (metis is_push_def rwir.simps)


definition wri :: "'a rel"
where
  "wri = {(a, b). is_pop a ∧ is_push b ∧ (b, a) ∉ wir}"

definition wii :: "'a rel"
where
  " wii = {(a, b). is_push a ∧ is_push b ∧ (a, b) ∈ pre}"

definition wrr :: "'a rel"
where
  "wrr = {(a, b). is_pop a ∧ is_pop b ∧ (a, b) ∈ pre}"

definition witness :: "'a rel"
where
  "witness = trancl (wir ∪ wri ∪ wii ∪ wrr)"

lemma lem_wir_then_push_and_pop : "(a, b) ∈ wir ⟹ is_push a ∧ is_pop b"
```

```
by (metis is_push_def wir.simps)

lemma lem_wri_then_pop_and_push : "(a, b) ∈ wri ⟹ is_pop a ∧ is_push b"
by (metis (lifting) internal_split_conv internal_split_def mem_Collect_eq
wri_def)

lemma lem_wir_wri_total : "is_push a ⟹ is_pop b ⟹ (a, b) ∈ wir ∨ (b,
a) ∈ wri"
apply (case_tac "(a, b) ∈ wir")
apply (rule disjI1)
apply (assumption)
apply (rule disjI2)
apply (unfold wri_def)
by (auto)

lemma lem_wrr_total : "is_pop a ⟹ is_pop b ⟹ a ≠ b ⟹ (a, b) ∈ wrr
∨ (b, a) ∈ wrr"
apply (frule_tac a="a" and b="b" in pop_ordered)
apply (assumption)+
apply (erule disjE)
apply (rule disjI1)
apply (unfold wrr_def)
by (safe)

lemma lem_wir_then_not_wri : "is_push a ⟹ is_pop b ⟹ (a, b) ∈ wir ⟹
(b, a) ∉ wri"
apply (rule notI)
apply (unfold wri_def)
by (safe)

lemma wriI : "is_push a ⟹ is_pop b ⟹ (a, b) ∉ wir ⟹ (b, a) ∈ wri"
by (metis lem_wir_wri_total)

lemma wriD : "(a, b) ∈ wri ⟹ is_pop a ∧ is_push b ∧ (b, a) ∉ wir"
by (metis lem_wir_then_not_wri lem_wri_then_pop_and_push)

lemma wrrD : "(a, b) ∈ wrr ⟹ is_pop a ∧ is_pop b ∧ (a, b)∈ pre"
by (metis (lifting) mem_Collect_eq split_conv wrr_def)

lemma wiiD : "(a, b) ∈ wii ⟹ is_push a ∧ is_push b ∧ (a, b)∈ pre"
apply (unfold wii_def)
by (simp add: set_eq_iff)

lemma lem_wri : "(rb, ia) ∈ wri ⟹ (ia, ra) ∈ val ⟹ (ix, rx) ∈ val
⟹
  ¬((ia, rb) ∈ pre ∨ ((ix, rb) ∈ pre ∧ (rx, ra) ∈ pre ∧ (ia, rx) ∈
wir))"
by (metis is_push_def lem_val_then_push_and_pop wir.wir_intro wir_right_rec
wriD)

lemma lem_rwir_then_ir : "(a, b) ∈ rwir ⟹ (a, b) ∈ ir"
apply (erule rwir.induct)
apply (metis lem_push_pr_pop_then_ir)
apply (erule exE)+
apply (erule conjE)+
apply (rotate_tac -3)
apply (erule contrapos_pp)
apply (frule_tac a="ix" and b="rx" in lem_val_then_push_and_pop)
apply (frule_tac a="a" and b="ra" in lem_val_then_push_and_pop)
apply (erule conjE)+
apply (frule_tac a="ix" and b="a" and c="b" in insI)
apply (assumption)+
apply (metis push_pop_ordered)
by (metis pre_antisymmetric pre_irreflexive ins_def antisymD ipr_ooo
lem_irreflexive_then_not_equal)

lemma lem_wir_then_ir : "(a, b) ∈ wir ⟹ (a, b) ∈ ir"
by (metis lem_rwir_then_ir lem_wir_then_rwir)

lemma lem_ir_then_wri : "(a, b) ∈ ir ⟹ is_pop a ⟹ is_push b ⟹ (a,
b)∈ wri"
apply (unfold wri_def)
apply (simp add: set_eq_iff)
apply (erule contrapos_pn)
apply (drule lem_wir_then_ir)
apply (rule notI)
apply (frule_tac c="(a, b)" and A="pre" and B="ir" in UnI2)
apply (frule_tac c="(b, a)" and A="pre" and B="ir" in UnI2)
apply (frule_tac a="a" and b="b" and c="a" and R ="pre Un ir" in
r_r_into_trancl)
apply (assumption)+
by (metis acyclic_def pr_u_ir_acyclic)

lemma lem_wir_then_not_rev_pr : "(a, b) ∈ wir ⟹ (b, a) ∉ pre"
apply (erule wir.induct)
apply (metis pre_antisymmetric pre_irreflexive antisymD
lem_irreflexive_then_not_equal)
by (metis lem_ir_then_wri lem_pop_pr_push_then_ir lem_val_then_push_and_pop
wir_left_right_rec wriD)

lemma lem_pr_then_wri : "(a, b) ∈ pre ⟹ is_pop a ⟹ is_push b ⟹ (a,
b) ∈ wri"
apply (rule wriI)
apply (assumption)+
apply (erule contrapos_pn)
by (metis lem_wir_then_not_rev_pr)

lemma lem_pr_in_witness: "pre ⊆ witness"
apply (rule subrelI)
apply (unfold witness_def)
apply (rule r_into_trancl)
apply (simp)
apply (frule lem_push_or_pop1)
apply (frule lem_push_or_pop2)
apply (erule disjE)+
— Subgoal 1: is_push x and is_push y
apply (rule disjI2)
apply (rule disjI2)
apply (rule disjI1)
apply (unfold wii_def)[1]
apply (simp)
— Subgoal 2: is_push x and is_pop y
apply (drule_tac a="x" and b="y" in wir.wir_intro)
apply (assumption)+
apply (rule disjI1)
apply (assumption)
apply (erule disjE)
— Subgoal 3: is_pop x and is_push y
apply (drule_tac a="x" and b="y" in lem_pr_then_wri)
apply (assumption)+
apply (rule disjI2)
apply (rule disjI1)
apply (assumption)
— Subgoal 4: is_pop x and is_pop y
apply (rule disjI2)+
apply (insert pre_irreflexive)
apply (drule_tac r="pre" and a="x" and b="y" in
lem_irreflexive_then_not_equal)
apply (assumption)
apply (unfold wrr_def)
by (safe)

lemma lem_wri_wir_then_pr : "(ra, ib) ∈ wri ⟹ (ib, rc) ∈ wir ⟹ (ra,
rc) ∈ pre"
apply (frule lem_wri_then_pop_and_push)
apply (frule lem_wir_then_push_and_pop)
apply (erule conjE)+
apply (case_tac "ra = rc")
apply (metis lem_wir_then_not_wri)
apply (frule_tac a="ra" and b="rc" in pop_ordered)
apply (assumption)
apply (assumption)
by (metis lem_wir_pr_then_wir lem_wir_then_not_wri)

lemma lem_pr_wri_then_wri : "(ra, rb) ∈ wrr ⟹ (rb, ic) ∈ wri ⟹ (ra,
ic) ∈ wri"
apply (rule wriI)
apply (metis lem_wri_then_pop_and_push)
apply (metis wrrD)
apply (drule wriD)
apply (erule conjE)+
apply (rotate_tac -1)
apply (erule contrapos_nn)
by (metis lem_wir_pr_then_wir wrrD)

lemma lem_ir_wir_then_pr : "(a, b) ∈ rwir ⟹ (c, a) ∈ ir ⟶ (c, b) ∈
pre"
by (metis ir_push_and_pop2 lem_ir_then_wri lem_rwir_then_wir
lem_wir_then_push_and_pop lem_wri_wir_then_pr)

lemma lem_ooo_infer_ir : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib)
∈ ins ⟹ (ra, rb) ∈ pre ⟹ (ra, ib) ∈ ir"
apply (rotate_tac -1)
apply (erule contrapos_pp)
apply (drule not_ir_pop_push_ir)
apply (metis is_pop_def)
apply (metis is_push_def)
apply (drule_tac ia="ia" and ra="ra" and ib="ib" and rb="rb" in ipr_ooo)
apply (assumption)
apply (metis ins_def)
apply (assumption)
by (metis pre_antisymmetric antisymD pre_irreflexive
lem_irreflexive_then_not_equal)
```

```
lemma lem_wir_then_pop_pr : "(a, b) ∈ wir ⟹ (a, ra) ∈ val ⟹ (a, b) ∉
pre ⟹ (b, ra) ∈ pre"
apply (drule wirD)
apply (erule disjE)
apply metis
apply (erule exE)+
apply (erule conjE)+
by (metis pre_transitive def_pop_once pop_once_def transD)

lemma lem_pr_wir_then_wir : "(a, b) ∈ wir ⟹ is_push c ⟹ (c, a) ∈ pre
⟹ (c, b) ∈ wir"
apply (frule lem_wir_then_push_and_pop)
apply (erule conjE)
apply (frule lem_push_then_pop_exists)
apply (erule exE)
apply (rename_tac rc)
apply (erule conjE)
apply (frule_tac a="a" in lem_push_then_pop_exists)
apply (erule exE)
apply (rename_tac ra)
apply (erule conjE)
apply (frule_tac a="ra" and b="rc" in pop_ordered)
apply (assumption)
apply (metis ins_def ipr_ooo lem_ir_then_wri lem_ooo_infer_ir
lem_pr_then_ins lem_val_then_ir val_then_pr wir.wir_intro wriD)
apply (erule disjE)
apply (case_tac "(a, b) ∈ pre")
apply (metis pre_transitive transE wir.wir_intro)
apply (rule wir.wir_step)
apply (rule_tac x="a" in exI)
apply (rule_tac x="ra" in exI)
apply (rule_tac x="rc" in exI)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (rule conjI)
apply (assumption)
apply (metis pre_transitive lem_wir_then_pop_pr transE val_then_pr
wir.wir_intro)
by (metis lem_ir_then_wri lem_ooo_infer_ir lem_pr_then_ins
lem_wir_pr_then_wir lem_wri_wir_then_pr val_then_pr wir.wir_intro)

lemma lem_wri_pr_then_wri : "(ra, ib)∈ wri ⟹ (ib, ic)∈ wii ⟹ (ra,
ic)∈ wri"
apply (rule wriI)
apply (metis wiiD)
apply (metis lem_wri_then_pop_and_push)
by (metis lem_pr_wir_then_wir wiiD wriD)

lemma lem_wir_wri_then_not_pr : "(ia, rb) ∈ wir ⟹ (rb, ic) ∈ wri ⟹
(ic, ia) ∉ pre"
by (metis lem_pr_wir_then_wir lem_wri_then_pop_and_push wriD)
```

```
lemma witnessD : "(a, b) ∈ witness ⟹
  (a, b) ∈ pre ∨
  (is_push a ∧ is_push b ∧ (∃c. (a, c) ∈ wir ∧ (c, b) ∈ wri)) ∨
  (is_push a ∧ is_pop b ∧ (a, b) ∈ wir) ∨
  (is_pop a ∧ is_push b ∧ (a, b) ∈ wri)"
apply (unfold witness_def)
apply (erule trancl.induct)
apply (metis UnE lem_wir_then_push_and_pop wiiD wriD wrrD)
apply (erule disjE)
apply (erule UnE)+
apply (metis lem_pr_then_wri lem_pr_wir_then_wir lem_push_or_pop1
lem_wir_then_push_and_pop lem_wri_wir_then_pr)
apply (metis lem_push_or_pop1 lem_wir_pr_then_wir wir.wir_intro wriD wriI)
apply (metis pre_transitive transE wiiD)
apply (metis pre_transitive transE wrrD)
apply (erule UnE)+
apply (metis def_value is_pop_def is_push_def lem_wir_pr_then_wir_help
lem_wir_then_push_and_pop lem_wri_wir_then_pr val_order_def)
apply (metis def_value is_pop_def lem_push_then_pop_exists
lem_wri_then_pop_and_push val_order_def)
apply (metis lem_wir_then_not_rev_pr lem_wri_pr_then_wri wiiD wriI)
by (metis def_value lem_pop_then_push_exists lem_push_then_pop_exists
lem_wir_pr_then_wir_help val_order_def wrrD)

lemma witnessI1 : "(a, b) ∈ pre ⟹ (a, b) ∈ witness"
by (metis lem_pr_in_witness set_rev_mp)

lemma witnessI2 : "(a, c) ∈ wir ⟹ (c, b) ∈ wri ⟹ (a, b) ∈ witness"
thm witness_def
apply (drule_tac c="(a, c)" and A="wir" and B="wri" in UnI1)
apply (drule_tac c="(a, c)" and A="wir ∪ wri" and B="wii" in UnI1)
apply (drule_tac c="(a, c)" and A="wir ∪ wri ∪ wii" and B="wrr" in UnI1)
apply (drule_tac c="(c, b)" and B="wri" and A="wir" in UnI2)
apply (drule_tac c="(c, b)" and A="wir ∪ wri" and B="wii" in UnI1)
apply (drule_tac c="(c, b)" and A="wir ∪ wri ∪ wii" and B="wrr" in UnI1)
by (metis r_r_into_trancl witness_def)

lemma witnessI3 : "(a, b) ∈ wir ⟹ (a, b) ∈ witness"
apply (drule_tac c="(a, b)" and A="wir" and B="wri" in UnI1)
apply (drule_tac c="(a, b)" and A="wir ∪ wri" and B="wii" in UnI1)
apply (drule_tac c="(a, b)" and A="wir ∪ wri ∪ wii" and B="wrr" in UnI1)
by (metis r_into_trancl' witness_def)

lemma witnessI4 : "(a, b) ∈ wri ⟹ (a, b) ∈ witness"
apply (drule_tac c="(a, b)" and B="wri" and A="wir" in UnI2)
apply (drule_tac c="(a, b)" and A="wir ∪ wri" and B="wii" in UnI1)
apply (drule_tac c="(a, b)" and A="wir ∪ wri ∪ wii" and B="wrr" in UnI1)
by (metis r_into_trancl' witness_def)

lemma lem_witness_irreflexive : "irrefl witness"
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (drule witnessD)
apply (erule disjE)
apply (metis irrefl_def pre_irreflexive)
by (metis def_value lem_pop_then_push_exists lem_push_then_pop_exists
lem_wir_then_not_wri lem_wri_then_pop_and_push val_order_def)
```

```
lemma lem_witness_antisymmetric : "antisym witness"
by (metis antisymI irrefl_def lem_witness_irreflexive trancl_trans
witness_def)

lemma lem_witness_wellformed : "wellformed_history witness val f"
apply (rule wellformed_historyI)
apply (metis trans_trancl witness_def)
apply (rule lem_witness_antisymmetric)
apply (rule lem_witness_irreflexive)
apply (rule operations_countable)
apply (rule operations_finite)
apply (rule def_push_once)
apply (rule def_pop_once)
apply (rule def_value)
apply (metis is_popD is_pushD lem_push_or_pop1 lem_push_or_pop2 witnessD)
by (metis def_val_then_not_pr def_value is_push_def lem_wri val_order_def
val_then_not_pr_def val_then_pr witnessD)

 lemma lem_witness_pop_ordered : "is_pop a ⟹ is_pop b ⟹ a≠b ⟹(a, b)
∈ witness ∨ (b, a) ∈ witness"
by (metis lem_pr_in_witness pop_ordered subsetD)

lemma lem_witness_push_pop_ordered : "is_push a ⟹ is_pop b ⟹ (a, b) ∈
witness ∨ (b, a) ∈ witness"
  apply (frule lem_wir_wri_total)
  apply (assumption)
  apply (erule disjE)
  apply (rule disjI1)
  apply (unfold witness_def)
  apply (rule r_into_trancl)
  apply (metis UnCI)
  apply (rule disjI2)
  apply (rule r_into_trancl)
  by (metis UnCI)

lemma lem_pr_wri_ooo: "(ia, rc) ∈ pre ⟹ (rc, ib) ∈ wri ⟹ (ia,
ra)∈val ⟹ (ib, rb) ∈ val ⟹ (ib, ra) ∈ wir ⟹ (rb, ra) ∈ pre"
apply (frule_tac a="ia" and b="ra" in lem_val_then_push_and_pop)
apply (frule_tac a="ib" and b="rb" in lem_val_then_push_and_pop)
apply (case_tac "ra=rb")
apply (metis def_push_once lem_wri push_once_def)
apply (erule conjE)+
apply (frule_tac a="ib" and b="ra" in lem_wir_wri_total)
apply (assumption)
apply (erule disjE)
apply (rotate_tac 1)
apply (frule lem_wri_then_pop_and_push)
apply (erule conjE)
apply (erule contrapos_pp)
apply (rule lem_wir_then_not_wri)
apply (assumption)+
apply (metis lem_wri pop_ordered wriI)
by (metis wriD)

lemma lem_wir_wri_ooo: "(ia, rc) ∈ wir ⟹ (rc, ib) ∈ wri ⟹ (ia,
ra)∈val ⟹ (ib, rb) ∈ val ⟹ (ib, ra) ∈ wir ⟹ (rb, ra) ∈ pre"
apply (frule_tac a="ia" and b="ra" in lem_val_then_push_and_pop)
apply (frule_tac a="ib" and b="rb" in lem_val_then_push_and_pop)
apply (case_tac "ra=rb")
apply (metis def_push_once push_once_def wriD)
apply (erule conjE)+
by (metis ir_push_and_pop1 lem_wir_then_ir lem_wir_then_not_wri pop_ordered
wir_left_right_rec)

lemma lem_witness_ooo : "(ia, ra)∈val ⟹(ib, rb)∈val ⟹ (ia,
ib)∈witness ⟹ (ib, ra)∈witness ⟹ (rb, ra)∈witness"
apply (frule_tac a="ia" and b="ra" in lem_val_then_push_and_pop)
apply (frule_tac a="ib" and b="rb" in lem_val_then_push_and_pop)
apply (erule conjE)+
apply (drule witnessD)+
apply (rule witnessI1)
apply (erule disjE)+
apply (metis ins_def ipr_ooo lem_pr_then_ins lem_push_pr_pop_then_ir)
apply (metis ins_def def_value ipr_ooo is_push_def lem_pr_then_ins
lem_wir_then_ir val_order_def)
apply (erule disjE)+

apply (metis lem_wir_wri_ooo wir.wir_intro)

apply (metis lem_irreflexive_then_not_equal lem_witness_irreflexive
witnessI3 witnessI4 wriI)
apply (erule disjE)+
apply (metis def_value is_push_def val_order_def)
apply (erule disjE)+
apply (metis lem_wir_wri_ooo)
apply (metis def_value is_push_def val_order_def)
by (metis def_value lem_pop_then_push_exists val_order_def)

theorem insert_pr_relaxed_stack_linearizable : "∃ prt .( pre ⊆ prt ∧
insert_relaxed_stack prt val f)"
apply (unfold insert_relaxed_stack_def)
apply (unfold insert_relaxed_stack_axioms_def)
apply (rule_tac x="witness" in exI)
apply (rule conjI)
apply (rule lem_pr_in_witness)
apply (rule conjI)
apply (rule lem_witness_wellformed)
apply (rule conjI)
apply (metis lem_pr_in_witness pop_ordered subsetD)
apply (rule conjI)
apply (metis lem_witness_push_pop_ordered)
by (metis lem_witness_ooo)

end
end
theory insert_relaxed_stack
imports Main concurrent_histories
begin


locale insert_relaxed_stack = wellformed_history +
  assumes pop_ordered: "is_pop a ⟹ is_pop b ⟹a≠b⟹(a, b) ∈ pre ∨
(b,a)∈pre"
  assumes push_pop_ordered: "is_push a ⟹is_pop b ⟹ (a, b) : pre ∨
(b,a)∈pre"
```

```
  assumes ir_ooo : "(ia, ra) ∈ val ⟹ (ib, rb) ∈ val ⟹ (ia, ib) ∈ pre
⟹ (ib, ra) ∈ pre ⟹ (rb, ra) ∈ pre"

context insert_relaxed_stack
begin

definition witness :: "'a rel"
where
  "witness = {(a, b). (a, b) ∈ pre ∨
           (∃ra rb. (a, ra) ∈ val ∧ (b, rb) ∈ val ∧ ((ra, b)∈pre ∨ ((a,
rb) ∈ pre ∧ (b, ra) ∈ pre ∧ (rb, ra)∈pre)))}"

lemma witnessI1 : "(a, b) ∈ pre ⟹ (a, b) ∈ witness"
apply (unfold witness_def)
by (simp add:set_eq_iff)

lemma witnessI2 : "(a, ra) ∈ val ⟹ (b, rb) ∈ val ⟹ (ra, b) ∈ pre ⟹
(a, b) ∈ witness"
apply (unfold witness_def)
apply (simp add:set_eq_iff)
by metis

lemma witnessI3 : "(a, ra) ∈ val ⟹ (b, rb) ∈ val ⟹ (a, rb) ∈ pre ⟹
(b, ra) ∈ pre ⟹ (rb, ra) ∈ pre ⟹ (a, b) ∈ witness"
apply (unfold witness_def)
apply (simp add:set_eq_iff)
by metis

lemma witnessD : "(a, b) ∈ witness ⟹ (a, b) ∈ pre ∨
 (∃ra rb. (a, ra) ∈ val ∧ (b, rb) ∈ val ∧ (ra, b)∈pre) ∨
 (∃ra rb. (a, ra) ∈ val ∧ (b, rb) ∈ val ∧ (a, rb) ∈ pre ∧ (b, ra) ∈ pre
∧ (rb, ra)∈pre)"
apply (unfold witness_def)
apply (simp add:set_eq_iff)
by metis

lemma lem_witness_irreflexive : "irrefl witness"
apply (unfold irrefl_def)
apply (rule allI)
apply (rule notI)
apply (drule witnessD)
apply (erule disjE)
apply (metis lem_irreflexive_then_not_equal pre_irreflexive)

apply (erule disjE)
apply (metis def_val_then_not_pr val_then_not_pr_def)
by (metis def_pop_once lem_irreflexive_then_not_equal pop_once_def
pre_irreflexive)

lemma lem_witness_transitive : "trans witness"
apply (insert pre_transitive)
apply (insert pre_irreflexive)
apply (insert pre_antisymmetric)
apply (rule transI)
apply (drule witnessD)
apply (drule witnessD)
apply (erule disjE)+
apply (metis transE witnessI1)
apply (erule disjE)
apply (metis def_val_then_not_pr lem_val_then_push_and_pop push_pop_ordered
transE val_then_not_pr_def witnessI1)
apply (erule exE)+
apply (erule conjE)+
apply (frule lem_push_or_pop1)
apply (erule disjE)
apply (smt ir_ooo irrefl_def is_push_def lem_val_then_push_and_pop
push_pop_ordered trans_def witnessI2 witnessI3)
apply (smt ir_ooo irrefl_def is_push_def push_pop_ordered trans_def
witnessI1)
apply (erule disjE)+
apply (metis def_val_then_not_pr lem_val_then_push_and_pop push_pop_ordered
transD val_then_not_pr_def witnessI1)
apply (frule lem_push_or_pop2)
apply (erule disjE)
apply (erule exE)+
apply (erule conjE)+
apply (frule lem_push_then_pop_exists)
apply (erule exE)
apply (erule conjE)
apply (rename_tac rx ry rz)
apply (case_tac "rx=rz")
apply (metis antisym_def def_push_once ir_ooo
lem_irreflexive_then_not_equal push_once_def)
apply (frule_tac ia="x" and ra="rx" in is_popI)
apply (frule_tac a="rx" and b="rz" in pop_ordered)
apply (assumption)+
apply (erule disjE)
apply (smt antisymD ir_ooo lem_val_then_push_and_pop push_pop_ordered
transD witnessI1)
apply (frule_tac ia="y" and ra="ry" in is_popI)
apply (case_tac "ry=rz")
apply (metis def_push_once lem_irreflexive_then_not_equal push_once_def)
apply (frule_tac a="ry" and b="rz" in pop_ordered)
apply (assumption)+
apply (erule disjE)
apply (smt push_pop_ordered transD witnessI2 witnessI3)
apply (frule_tac ia="x" and ra="rx" in is_pushI)
apply (frule_tac a="x" and b="rz" in push_pop_ordered)
apply (assumption)
apply (erule disjE)
apply (metis push_pop_ordered witnessI2 witnessI3)
apply (smt ir_ooo lem_irreflexive_then_not_equal lem_witness_irreflexive
push_pop_ordered trans_def witnessI2)
apply (smt ir_ooo is_pushI lem_irreflexive_then_not_equal push_pop_ordered
transD witnessI1)
apply (erule disjE)+
apply (metis def_val_then_not_pr lem_val_then_push_and_pop push_pop_ordered
transD val_then_not_pr_def witnessI1)
apply (smt def_val_then_not_pr ir_ooo lem_irreflexive_then_not_equal
lem_val_then_push_and_pop push_pop_ordered transD val_then_not_pr_def
witnessI1)
apply (erule disjE)
apply (metis def_pop_once pop_once_def transD witnessI1)
apply (erule exE)+
apply (erule conjE)+
apply (rename_tac rx ry ryy rz)
```

apply *(frule_tac ia="x"* **and** *ra="rx"* **in** *is_pushI)*
apply *(frule_tac a="x"* **and** *b="rz"* **in** *push_pop_ordered)*
apply *(metis is_pop_def)*
apply *(erule disjE)*
apply *(smt def_pop_once pop_once_def transE witnessI3)*
by *(smt ir_ooo lem_irreflexive_then_not_equal transD)*

lemma *lem_witness_antisymmetric : "antisym witness"*
apply *(insert pre_transitive)*
apply *(insert pre_irreflexive)*
apply *(insert pre_antisymmetric)*
apply *(rule antisymI)*
apply *(drule witnessD)+*
apply *(erule disjE)+*
apply *(metis antisymD)*
apply *(erule disjE)*
apply *(metis def_val_then_not_pr transD val_then_not_pr_def)*
apply *(metis antisymD ir_ooo lem_irreflexive_then_not_equal)*
apply *(erule disjE)+*
apply *(metis def_val_then_not_pr transD val_then_not_pr_def)*
apply *(metis antisymD ir_ooo lem_irreflexive_then_not_equal)*
apply *(erule disjE)+*
apply *(metis def_val_then_not_pr lem_val_then_push_and_pop push_pop_ordered transE val_then_not_pr_def)*
apply *(metis antisymD def_pop_once def_value pop_once_def val_order_def)*
apply *(erule disjE)*
apply *(metis antisym_def def_pop_once def_value pop_once_def val_order_def)*
by *(metis antisym_def def_pop_once irrefl_def pop_once_def)*

lemma *lem_pr_in_witness : "pre* $\subseteq$ *witness"*
by *(metis subrelI witnessI1)*

lemma *lem_witness_total_on_push : "is_push a* $\Longrightarrow$ *is_push b* $\Longrightarrow$ *a* $\neq$ *b* $\Longrightarrow$ *(a, b)* $\in$ *witness* $\vee$ *(b, a)* $\in$ *witness"*
apply *(insert pre_transitive)*
apply *(insert pre_irreflexive)*
apply *(insert pre_antisymmetric)*
apply *(frule_tac ia="a"* **in** *is_pushD)*
apply *(frule_tac ia="b"* **in** *is_pushD)*
apply *(erule exE)+*
apply *(rename_tac rb)*
apply *(frule_tac a="a"* **and** *b="rb"* **in** *push_pop_ordered)*
apply *(metis is_pop_def)*
apply *(frule_tac a="b"* **and** *b="ra"* **in** *push_pop_ordered)*
apply *(metis is_pop_def)*
apply *(erule disjE)+*
apply *(frule_tac ia="a"* **and** *ra="ra"* **in** *is_popI)*
apply *(frule_tac ia="b"* **and** *ra="rb"* **in** *is_popI)*
apply *(case_tac "ra=rb")*
apply *(metis def_push_once push_once_def)*
apply *(metis pop_ordered witnessI3)*
apply *(metis witnessI2)*
by *(metis witnessI2)*

lemma *lem_witness_total_on_pop : "is_pop a* $\Longrightarrow$ *is_pop b* $\Longrightarrow$ *a* $\neq$ *b* $\Longrightarrow$ *(a, b)* $\in$ *witness* $\vee$ *(b, a)* $\in$ *witness"*
by *(metis pop_ordered witnessI1)*

```
lemma lem_witness_total_on_push_pop : "is_push a ⟹ is_pop b ⟹ (a, b)
∈ witness ∨ (b, a) ∈ witness"
by (metis push_pop_ordered witnessI1)

lemma lem_val_then_pr: "(ia, ra)∈val ⟹ (ia, ra) ∈ pre"
by (metis def_val_then_not_pr is_pop_def is_push_def push_pop_ordered
val_then_not_pr_def)

lemma lem_val_then_not_witness : "(ia, ra) ∈ val ⟹ (ra, ia) ∉ witness"
by (metis def_val_then_not_pr def_value val_order_def val_then_not_pr_def
witnessD)

lemma lem_witness_wellformed : "wellformed_history witness val f"
apply (rule wellformed_historyI)
apply (rule lem_witness_transitive)
apply (rule lem_witness_antisymmetric)
apply (rule lem_witness_irreflexive)
apply (rule operations_countable)
apply (rule operations_finite)
apply (rule def_push_once)
apply (rule def_pop_once)
apply (rule def_value)
apply (metis def_push_or_pop witnessD)
by (metis lem_val_then_not_witness val_then_not_prI)

lemma lem_witness_sequential_history : "sequential_set witness val f"
apply (unfold sequential_set_def)
apply (rule conjI)
apply (metis lem_witness_wellformed)
apply (unfold sequential_set_axioms_def)
apply (rule conjI)
apply (metis lem_witness_total_on_push)
apply (rule conjI)
apply (metis lem_witness_total_on_pop)
by (metis lem_witness_total_on_push_pop)

lemma push_witness_pop_then_pr : "(a, b) ∈ witness ⟹ is_push a ⟹
is_pop b ⟹ (a, b) ∈ pre"
by (metis is_pushI lem_irreflexive_then_not_equal lem_witness_irreflexive
push_pop_ordered witnessD witnessI1)

lemma lem_witness_ooo : "sequential_lifo witness val"
apply (unfold sequential_lifo_def)
apply (rule allI)+
apply (rule impI)+
apply (drule_tac a="ib" and b="ra" in push_witness_pop_then_pr)
apply (metis is_pushI)
apply (metis is_pop_def)
apply (drule witnessD)
apply (erule disjE)
apply (metis ir_ooo witnessI1)
apply (erule disjE)
apply (metis pre_antisymmetric antisym_def def_pop_once def_value
pop_once_def val_order_def)
by (metis def_pop_once pop_once_def witnessI1)

lemma lem_witness_sequential_stack : "sequential_stack witness val f"
apply (unfold sequential_stack_def)
apply (rule conjI)
apply (metis lem_witness_sequential_history)
apply (unfold sequential_stack_axioms_def)
by (metis lem_witness_ooo)

theorem insert_relaxed_stack_linearizable : "∃ prt .( pre ⊆ prt ∧
sequential_stack prt val f)"
apply (rule_tac x="witness" in exI)
apply (rule conjI)
apply (metis lem_pr_in_witness)
by (metis lem_witness_sequential_stack)

end

end
```