Existence of Forward Simulations for Particular Data Structures

January 23, 2017

1 Preliminaries

Systems we consider are labeled transition systems (LTS):

Definition 1. *An LTS is defined over four-tuples $A = (Q, \Sigma, q_0, \delta)$ where Q is the set of states, Σ is the set of transition labels, $q_0 \in Q$ is the initial state and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.*

Executions generated by this system are alternating sequence of states and transition labels $\rho = s_0, e_0, s_1, ...s_k, e_k, ...$ where each $s_i \in Q$, each $e_i \in \Sigma$, $s_0 = q_0$ and each $(s_i, e_i s_{i+1}) \in \delta$. The projection of the sequence ρ over the set Π is denoted by $\rho|\Pi$, and it is the maximum subsequence of ρ consisting of elements of Π. Traces of the LTS are obtained from executions by projecting them over Σ. For the rest of the paper and in all of the proofs, we consider only finite executions (denoted as $E(A)$) and/or traces (denoted as $Tr(A)$ of the LTSs in focus.

Libraries are LTSs that provide methods. Let $\mathcal{M}$ be the set of method names and $\mathcal{D}$ be the domain of values as input/output parameters for the methods. Then, this library contains transition labels of the form $inv(m, d, i)$ representing the invocation of method $m \in \mathcal{M}$ with input value $d \in \mathcal{D}$. The third field is the operation identifier for differentiating the different calls of the same method from the set $\mathcal{O}$. For simplicity, we take $\mathcal{O} = \mathbb{N}$ for the rest of the paper. We also assume that methods could have at most one input parameter. If they do not have any input arguments (like pop method of a stack), we can omit the second field from the action. They also provide actions of the form $ret(m, d, i)$ representing the return of method $m \in M$ with value $d \in D$ which has been invoked previously with action $inv(m, d', i)$. Again, we assume that the methods can return at most one parameter and we may omit the second field from the action if they have none (like enqueue method of a queue). Before starting to reason about any set of libraries, we first fix the sets $\mathcal{M}$ and $\mathcal{D}$ and libraries in our focus agree on this sets. For any transition label $e = inv(m, d, i)$ or $e = ret(m, d, i)$, we have the function $oid(e) = i$.

Since libraries are LTSs, they produce traces. A trace $e = e_1, e_2, ..., e_n$ of library L is *well-formed* iff (i) every return matches an earlier invocation:

$e_j = ret(m, d, k)$ implies that there exists $i < j$ such that $e_i = inv(m, d', k)$ and (ii) every operation identifier is used at most one invocation/return pair: $oid(e_i) = oid(e_j) = k$ and $i < j$ implies $e_i = inv(m, d, k)$ and $e_j = ret(m, d', k)$. From now on, we assume that libraries produce well-formed traces. Let $f : \mathbb{N} \to (N)$ be a bijection. Then, traces e and e' are equivalent if e' is obtained from e by replacing every action $inv(m, d, k)$ with $inv(m, d, f(k))$ and every action $ret(m, d, k)$ with $ret(m, d, f(k))$.

Based on these definitions on the traces of the libraries, we can define refinement between libraries:

Definition 2. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. We define the set $A\Sigma = \{inv(m, d, i) | m \in \mathcal{M} \wedge d \in \mathcal{D} \wedge i \in \mathbb{N}\} \cup \{ret(m, d, i) | m \in \mathcal{M} \wedge d \in \mathcal{D} \wedge i \in \mathbb{N}\}$ as abstract transition labels. Note that $A\Sigma \subseteq \Sigma_{L_1}$ and $A\Sigma \subseteq \Sigma_{L_2}$. Then, we say L_1 refines L_2 iff for every trace $e \in Tr(L_1)$, there exists a trace $e' \in Tr(L_2)$ such that $e|A\Sigma$ is equivalent to $e'|A\Sigma$.*

Linearizability is also a relation between two libraries and it is stricter than refinement. It requires e' in Definition 2 to be a sequential one. A trace e is sequential iff following two conditions hold for its projection to abstract transition labels $e|A\Sigma = e_1, ...e_n$: (i) $e_1 = inv(m, d, k)$ for some $m \in \mathcal{M}, d \in \mathcal{D} and k \in \mathbb{N}$ and (ii) for all $i \in [1, n)$, either $e_i = inv(m, d, k)$ and $e_{i+1} = ret(m, d', k)$ or $e_i = ret(m, d, k)$ and $e_{i+1} = inv(m', d', k')$ for some $m, m' \in \mathcal{M}$, $d, d' \in \mathcal{D}$ and $k, k' \in \mathbb{N}$.

We can extend the relations between libraries by introducing simulation relations. We will later show that simulation relations imply refinement.

Definition 3. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. Then, the relation $fs \subseteq Q_{L_1} \times Q_{L_2}$ is called a forward simulation iff the following holds:*

(i) $fs[q_{0_{L_1}}] = \{q_{0_{L_2}}\}$

(ii-a) *If $(s, inv(m, d, k), s') \in \delta_{L_1}$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{a} u'$ where $a = a_1, a_2, ..., a_n$ such that $a_1 = inv(m, d, k)$ and for all $i \in [2, n]$, $a_i \in \Sigma_{L_2} \backslash A\Sigma$. The expression $u \xrightarrow{a} u'$ means that there exists a sequence of states $u_1, u_2, ..., u_{n+1}$ such that $u_1 = u$, $u_{n+1} = u'$ and for all $i \in [1, n]$, $(u_i, a_i, u_{i+1}) \in \delta_{L_2}$.*

(ii-b) *If $(s, ret(m, d, k), s') \in \delta_{L_1}$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{a} u'$ where $a = a_1, a_2, ..., a_n$ such that $a_n = ret(m, d, k)$ and for all $i \in [1, n-1]$, $a_i \in \Sigma_{L_2} \backslash A\Sigma$.*

(ii-c) *If $(s, t, s') \in \delta_{L_1}$ for some $t \in \Sigma_{L_1} \backslash A\Sigma$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{a} u'$ where $a = a_1, a_2, ..., a_n$ such that for all $i \in [1, n]$, $a_i \in \Sigma_{L_2} \backslash A\Sigma$. Moreover, a could be the empty sequence.*

If $fs[s]$ is a unique state for all $s \in Q_{L_1}$ then it is called a refinement mapping/function. A dual notion of forward simulation is the backward simulation:

Definition 4. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. Then, the relation $bs \subseteq Q_{L_1} \times Q_{L_2}$ is called a backward simulation iff the following holds:*

(i) $bs[q_{0_{L_1}}] = \{q_{0_{L_2}}\}$

(ii-a) If $(s, inv(m,d,k), s') \in \delta_{L_1}$ *and* $u' \in bs[s']$*, then there exists* $u \in bs[s]$ *such that* $u \xrightarrow{a} u'$ *where* $a = a_1, a_2, ..., a_n$ *such that* $a_1 = inv(m,d,k)$ *and for all* $i \in [2,n]$*,* $a_i \in \Sigma_{L_2} \backslash A\Sigma$*.*

(ii-b) If $(s, ret(m,d,k), s') \in \delta_{L_1}$ *and* $u' \in bs[s']$*, then there exists* $u \in bs[s]$ *such that* $u \xrightarrow{a} u'$ *where* $a = a_1, a_2, ..., a_n$ *such that* $a_n = ret(m,d,k)$ *and for all* $i \in [1, n-1]$*,* $a_i \in \Sigma_{L_2} \backslash A\Sigma$*.*

(ii-c) If $(s, t, s') \in \delta_{L_1}$ *for some* $t \in \Sigma_{L_1} \backslash A\Sigma$ *and* $u' \in bs[s']$*, then there exists* $u \in bs[s]$ *such that* $u \xrightarrow{a} u'$ *where* $a = a_1, a_2, ..., a_n$ *such that for all* $i \in [1,n]$*,* $a_i \in \Sigma_{L_2} \backslash A\Sigma$*. Moreover, a could be the empty sequence.*

Simulation relations are used to prove refinement relations among libraries. Following lemmas show their soundness:

Lemma 1. *Let L_1 and L_2 be two libraries agreeing on $\mathcal{M}$ and $\mathcal{D}$ sets. If fs (bs) is a forward (backward) simulation relating L_1 to L_2, then L_1 refines L_2.*

Proof. Looks trivial and follows Lynch paper. Can be completed later. □

2 Existence of Forward Simulations for Queue Implementations that have Fixed Dequeue Linearization Points

In this section, we will show that for any concurrent queue implementation library L_C for which we know the linearization points of the dequeue operation and that is linearizable with respect to the reference implementation library L_A, there exists a forward simulation fs relating L_C to L_I where L_I is an intermediate library equivalent to L_A i.e. L_I refines L_A and vice versa.

For all the queue libraries, we fix $\mathcal{M} = \{enq, deq\}$ and $\mathcal{D} = \mathbb{N} \cup \{\texttt{EMPTY}\}$. Since we know the linearization point of dequeues, we extend the definitions of the previous sections adding this information. First, we extend the set $A\Sigma$ introduced in Definition 2 for queues in our focus as $AQ\Sigma = A\Sigma \cup \{lin(deq,d,k) | d \in \mathcal{D}, k \in \mathbb{N}\}$. For any library L we consider in this section, $AQ\Sigma \subseteq \Sigma_L$. Then, we define q-refinement by replacing $A\Sigma$ with $AQ\Sigma$ in Definition 2. We also define q-linearizability, by enforcing $lin(deq,d,k)$ to appear immediately after $inv(deq,k)$ and immediately before $ret(deq,d,k)$ in a sequential execution. We also change definition of forward and backward simulation relations by replacing $A\Sigma$ with $AQ\Sigma$ in the original definitions and adding a new condition $(ii-d)$ to each of them:

Forward Simulation: (ii-d) If $(s, lin(deq, d, k), s') \in \delta_{L_1}$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $(u, lin(deq, d, k), u') \in \delta_{L_2}$

Backward Simulation: (ii-d) If $(s, lin(deq, d, k), s') \in \delta_{L_1}$ and $u' \in bs[s']$, then there exists $u \in bs[s]$ such that $(u, lin(deq, d, k), u') \in \delta_{L_2}$

Lemma 1 of the previous section still holds if we replace refinement with q-refinement and use new definitions of backward and forward simulations.

Next, we define the abstract library L_A as follows:

- A queue state consists of a finite sequence of natural numbers representing the queue content and a program counter for each operation. More formally: $Q_A \subseteq \mathbb{N}^* \times (\mathbb{N} \to Lbl_A)$ where $Lbl_A = \{N, E_0, E_1, E_2, D_0, D_1, D_2\}$ is the set of transition labels of the operations. Operations that have not started yet are mapped to N. E_i (D_i) denotes particular transitions in enqueue (dequeue) operations that will be clear when we define δ_A. For a state q, we denote the contents of the queue (first field) with s_q and the function that maps operations to labels (the second field) with f_q.

- Transition labels consists of invocations, returns and linearizations of enqueue and dequeue operations: $\Sigma_A = AQ\Sigma \cup \{lin(enq, d, k) | d \in \mathcal{D}, k \in \mathbb{N}\}$. Invocation of *deq* operation does not have any input and return of *enq* operation does not have any output value. We omit second fields from these labels.

- Initial state is the empty queue: $q_{0A} = (\langle\rangle, f_{q_{0A}})$ where $f_{q_{0A}}(i) = N$ for all $i \in \mathbb{N}$.

- Each operation consists of invocation, linearization and return steps. These are reflected in the transition relation. For the below definitions, unchanged parts of the state are omitted from the formulae:
  - $(q, inv(enq, d, k), q') \in \delta_A$ iff $d \neq \texttt{EMPTY} \wedge f_q(k) = N \wedge f_{q'}(k) = E_0$,
  - $(q, lin(enq, d, k), q') \in \delta_A$ iff $f_q(k) = E_0 \wedge s_{q'} = s_q \circ \langle d \rangle \wedge f_{q'}(k) = E_1$ where $\circ$ is the operation that concatenates two finite sequences,
  - $(q, ret(enq, k), q') \in \delta_A$ iff $f_q(k) = E_1 \wedge f_{q'}(k) = E_2$,
  - $(q, inv(deq, k), q') \in \delta_A$ iff $f_q(k) = N \wedge f_{q'}(k) = D_0$
  - $(q, lin(deq, d, k), q') \in \delta_A$ iff $f_q(k) = D_0 \wedge (d \neq \texttt{EMPTY} \wedge s_q = \langle d \rangle \circ s_{q'} \vee d = \texttt{EMPTY} \wedge s_q = s_{q'} = \langle\rangle) \wedge f_{q'}(k) = D_1$,
  - $(q, ret(deq, d, k), q') \in \delta_A$ iff $f_q(k) = D_1 \wedge f_{q'}(k) = D_2$.

We restrict traces generated by this LTS by neglecting the invalid traces. If we project a trace to some operation identifier k and obtain a sequence $\langle ..., inv(enq, d, k), lin(enq, d', k), ... \rangle$ or $\langle ..., lin(deq, d, k), ret(deq, d', k), ... \rangle$ where $d \neq d'$ we say that this trace is invalid.

We want to introduce L_I as the next step. States of L_I consists of strict orders of enqueue operations. Nodes of the strict order come from the set

$ND = \mathbb{N} \times \mathcal{D} \times \{\texttt{PENDING}, \texttt{CLOSED}\}$. Basically each node is a tuple keeping the operation ID, the value to be enqueued and if this enqueue is pending or closed. Strict order also keeps a set of directed edges $ed \subseteq ED = ND \times ND$. Then, a strict order is a tuple (nd, ed) where the set ed obeys the assumption of strict order i.e. irreflexivity, asymmetry and transitivity. Then, the LTS of L_I is defined as follows:

- A state consists of a partial order and a function keeping the program counter. More formally $Q_I \subseteq ND \times ED \times (\mathbb{N} \to Lbl_I)$ where $Lbl_I = \{N, E_0, E_1, D_0, D_1, D_2\}$ is the set of transition labels. nd_q ed_q and f_q denotes the nodes of the strict order in state q, edges of the strict order in state q and the function that maps operation to labels in state q, respectively.

- The transition label set consists of invocation and return action of both methods and linearization action of only the dequeue method: $\Sigma_I = AQ\Sigma$. Invocation of *deq* operation does not have any input and return of *enq* operation does not have any output value. We omit second fields from these labels.

- Initial state consists of an empty strict order and a function mapping every operation to N: $q_{0I} = (so_{q_{0I}}, f_{q_{0I}}$ where $so_{q_{0I}} = (\emptyset, \emptyset)$ and $f_{q_{0I}}(i) = N$ for all $i \in \mathbb{N}$.

- While defining δ_I, we again omit mentioning about the parts that has not changed:

  - $(q, inv(enq, d, k), q') \in \delta_I$ iff $f_q(k) = N \wedge f_{q'}(k) = E_0 \wedge d \neq \texttt{EMPTY} \wedge (k, \_, \_) \notin nd_q \wedge nd_{q'} = nd_q \cup \{(k, d, \texttt{PENDING}\} \wedge AddNode(ed_{q'}, ed_q, k)$ where $AddNode(ed_{q'}, ed_q, k)$ is true iff $ed_{q'}$ is obtained from ed_q by adding edges from every closed node of ed_q to the node with identifier k.
  - $(q, ret(enq, k), q') \in \delta_I$ iff $f_q(k) = E_0 \wedge f_{q'}(k) = E_1 \wedge ((k, \_, \texttt{PENDING}) \notin nd_q \wedge so_q = so_{q'} \vee UpdateNode(so_{q'}, so_q, k))$ where $UpdateNode(so_{q'}, so_q, k)$ is true iff $(k, d, \texttt{PENDING}) \in nd_q$ for some $d \in \mathcal{D}$, it is replaced with node $(k, d, \texttt{CLOSED}$ in the state q' and all the edges adjacent to $(k, d, \texttt{PENDING})$ in state q are replaced with edges adjacent to $(k, d, \texttt{CLOSED})$ in the state q'.
  - $(q, inv(deq, k), q') \in \delta_I$ iff $f_q(k) = N \wedge f_{q'}(k) = D_0$.
  - $(q, lin(deq, d, k), q') \in \delta_I$ iff $f_q(k) = D_0 \wedge f_{q'}(k) = D_1 \wedge (d = \texttt{EMPTY} \wedge ed_q = ed_{q'} = \emptyset \wedge Pending(nd_q) \wedge Pending(nd_{q'} \vee d \neq \texttt{EMPTY} \wedge RemoveNode(so_q, so_{q'}, d))$ where $RemoveNode(so_q, so_{q'}, d)$ is true iff there is a minimal node $n \in so_q$ of which data value (second field) is d and $so_{q'}$ is obtained from so_q by removing n and all the edges adjacent to it and $Pending(nd)$ iff all the nodes in nd are pending. Note that linearization of dequeue could remove a PENDING node.

– $(q, ret(deq, d, k)q') \in \delta_I$ iff $f_q(k) = D_1 \wedge f_{q'}(k) = D_2$.

Again, we omit the inconsistent traces from L_I as in L_A. Note that the library L_I is deterministic with respect to alphabe $AQ\Sigma$.

Next, we show equivalence of L_A and L_I in terms of q-refinement.

Lemma 2. *L_I is a q-refinement of L_A.*

Proof. We will provide a backward simulation relation btw L_I and L_A. Given a state $q = (so_q, f_q) \in Q_I$, $q' \in bs[q]$ iff (i) $s_{q'}$ is a linearization of so_q projected to d fields (second field). This linearization may omit the pending elements (the ones of which third field is `PENDING`), but it must contain all the `CLOSED` elements. Note that pending elements of so_q are maximal and they can appear after the closed elements, towards the end of the sequence. (ii) If $f_q(k) \in \{N, D_0, D_1, D_2\}$, then $f_{q'}(k) = f_q(k)$. If $f_q(k) = E_0$, $(k, d, \texttt{PENDING}) \in nd_q$ and this node participates in the linearization $s_{q'}$. Then $f_{q'}(k) = E_1$. If it does not participate in the linearization, then $f_{q'}(k) = E_0$. If $f_q(k) = E_0$ but there is no node with operation identifier k in nd_q, then $f_{q'} = E_1$. Lastly, if $f_q(k) = E_1$ then $f_{q'}(k) = E_2$. Next, we will show that bs is a backward simulation relation.

$\langle i \rangle$ $bs[q_{0\,I}] = \{q_{0\,A}\}$.

$\langle ii - a - enq \rangle$ Let $(q, inv(enq, d, k), q') \in \delta_I$ and $t' \in bs[q']$. We know that $(k, d, \texttt{PENDING}$ is a maximal element in $so_{q'}$ and either $s_{t'}$ does not linearize it or $s_{t'} = \rho \circ \langle d \rangle \circ \pi$ where π consists of only linearization of `PENDING` elements. For the first case $s_{t'} = s_t$ for some $t \in bs[q]$ and $(t, inv(enq, d, k), t') \in \delta_A$. For the latter case, $s_t = \rho$ for some $t \in bs[q]$ and $t \xrightarrow{a}_{L_A} t'$ where $a = inv(enq, d, k), lin(enq, d, k), lin(enq, d_1, k_1), ..., lin(enq, d_j, k_j)$ where $\pi = d_1, ...d_j$. The f_t could be obtained easily by observing the sequence a and it can be checked that such $t \in bs[q]$ exists.

$\langle ii - a - deq \rangle$ Let $(q, inv(deq, k), q') \in \delta_I$ and $t' \in bs[q']$. We know that $f_{t'}(k) = D_0$. We know that there is a $t \in bs[q]$ such that $s_t = s_{t'}$ (since $so_q = so_{q'}$), $f_t(k) = N$ and $(t, inv(deq, k), t') \in \delta_A$.

$\langle ii - d \rangle$ Let $(q, lin(deq, d, k), q') \in \delta_I$ and $t' \in bs[q']$. First consider the case $d \neq$ `EMPTY`. We have two cases: Either the node with operation id k in the so_q was `PENDING` or `CLOSED`. For both of the cases, we can find $t \in bs[q]$ such that $s_t = \langle d \rangle \circ s_{t'}$ and $f_t(k) = D_0$. Hence $(t, lin(deq, d, k), t') \in \delta_I$. Next, consider the case $d =$ `EMPTY`. Then, we know that $s_{t'} = \langle\rangle$ and there exists $t \in bs[q]$ such that $s_t = \langle\rangle$ and $f_t = D_0$. Hence $(t, lin(deq, d, k), t') \in \delta_A$.

$\langle ii - b - enq \rangle$ Let $(q, ret(enq, k), q') \in \delta_I$ and $t' \in bs[q']$. There are two possible cases: There exists a node $(k, d, \texttt{PENDING}) \in nd_q$ or for all nodes $n = (k, \_, \_)$, $n \notin nd_q$. For the former case, $s_{t'} = \rho \circ \langle d \rangle \circ \pi$ where ρ is linearization of closed nodes in $nd_{q'}$ and π is linearization of some open nodes in $node_{q'}$. We also know that $f_{t'}(k) = E_2$. Then, there exists a node $t \in bs[q]$ such that $s_t = s_{t'}$ (since nodes that generate ρ are closed in q and that generate π are open in q) and $f_t(k) = E_1$. Hence, $(t, ret(deq, k), t') \in \delta_A$. For the

latter case, we know that $so_q = so_{q'}$. Therefore, there exists $t \in bs[q]$ such that $s_t = s_{t'}$. Moreover $f_t(k) = E_1$. Hence, $(t, ret(enq, k), t') \in \delta_A$.

$\langle ii - b - deq \rangle$ Let $(q, ret(deq, d, k), q') \in \delta_I$ and $t' \in bs[q']$. Then, we know that $f_{t'}(k) = D_2$ and there exists a $t \in bs[q]$ such that $s_t = s_{t'}$ (since $so_q = so_{q'}$) and $f_t(k) = D_1$. Then, $(t, ret(deq, d, k) \in \delta_A$.

□

Lemma 3. *L_A is a q-refinement of L_I.*

Proof. Since L_I is deterministic with respect to the alphabet $AQ\Sigma$, we should be able to find a forward simulation between L_A and L_I if our claim is correct. We propose a forward simulation by adding some auxiliary varibles to the state of L_A. In the new augmented state of L_A, the sequence does not only keep the elements of the queue but also the operation identifiers that enqueued them. Hence, the sequence consists of pairs of the form $s_q(i) = (k, d)$ where $q \in Q_A$, $i \in \mathbb{N}$ is an index, $d \in \mathcal{D}\backslash\{\texttt{EMPTY}\}$ is a value and $k \in \mathbb{N}$ is an operation identifier. We also add a set $InvEnq_q$ component to state that keeps the enqueue operations at the point E_0 i.e. enqueues that are invoked bu have not been linearized yet. Elements of $InvEnq_q$ are pairs of the form (d, k) where $d \in \mathcal{D}\backslash\{\texttt{EMPTY}\}$ is a value and $k \in \mathbb{N}$ is an operation identifier. Then our forward simulation $fs \subseteq Q_A \times Q_I$ relates the state $q = (s_q, f_q) \in Q_A$ to a state $q' = (so_{q'}, f_{q'}) \in Q_I$ iff (i) $f_{q'}(k) = f_q(k)$ for all $f_q(k) \in \{N, D_0, D_1, D_2\}$, (ii) $f_q(k) = E_0$ or $f_q(k) = E_1$ implies $f_{q'}(k) = E_0$, (iii) $f_q(k) = E_2$ implies $f_{q'}(k) = E_1$, (iv) If $(d, k) \in InvEnq_q$ or there exists an index $i \in \mathbb{N}$ such that $s_q(i) = (k, d)$ and $f_q(k) = E_1$, then $(k, d, \texttt{PENDING})$ is a node in $so_{q'}$, (v) if there exists an index $i \in \mathbb{N}$ such that $s_q(i) = (k, d)$ and $f_q(k) = E_2$ then $(k, d, \texttt{CLOSED})$ is a node in $so_{q'}$, (vi) open nodes in $so_{q'}$ are maximal and (vii) there exists a linearization (total order) $lo_{q'}$ of $so_{q'}$ that may omit some open nodes of $so_{q'}$ such that if we project nodes of $lo_{q'}$ to the first two fields, this linear order is equal to the sequence s_q.

Next, we will show that fs is a forward simulation relation:

$\langle i \rangle$ $fs[(q_0)_A] = \{(q_0)_I\}$

$\langle ii - a - enq \rangle$ Let $(q, inv(enq, d, k), q') \in \delta_A$ and $t \in fs[q]$. We can obtain t' such that $(t, inv(enq, d, k), t') \in \delta_I$. We show that $t' \in fs[q']$ by checking seven conditions of our fs relation. We omit the node (k, d) while linearizing $so_{t'}$ and obtain $s_{q'}$.

$\langle ii - a - deq \rangle$ Let $(q, inv(deq, k), q') \in \delta_A$ and $t \in fs[q]$. We obtain t' such that $(t, inv(deq, k), t') \in \delta_I$. The only difference between t' and t is that $f_{t'}(k) = D_0$. We can again see that $t' \in fs[q']$.

$\langle ii - c \rangle$ Let $(q, lin(enq, d, k), q') \in \delta_A$ and $t \in fs[q]$. Pick $t' = t$. We can see that $t \in fs[q']$. While obtaining linearization of $so_{t'}$, we do not neglect the node $(k, d, \texttt{PENDING})$ this time, although we neglect it in the linearization of so_t.

$\langle ii-d \rangle$ Let $(q, lin(deq, d, k), q') \in \delta_A$ and $t \in fs[q]$. Obtain t' such that $(t, lin(deq, d, k), t') \in \delta_I$. In s_q, (k, d) must be the minimum element. Hence, $(k, d, \_)$ is a minimal node in so_t and $lin(deq, d, k)$ is an enabled action in L_I. This action removes the node $(k, d, \_)$ from so_t and changes $f_t(k) = D_0$ to $f_{t'}(k) = D_1$. We can see that $t' \in fs[q']$ by checking the seven conditions.

$\langle ii-b-enq \rangle$ Let $(q, ret(enq, k), q') \in \delta_A$ and $t \in fs[q]$. There are two cases: Either there exists an index $i \in \mathbb{N}$ such that $s_q(i) = (k, d)$ or there is no such i. For the former case, there is no node of the form $(k, d, \_)$ in so_t. We obtain t' such that $(t, ret(enq, k), t') \in \delta_I$. Then, there is no node of the form $(k, d, \_)$ in $so_{t'}$ neither and $so_{t'} = so_t$. since $s_q = s_{q'}$ also holds for this case, $t' \in fs[q']$ holds. For the latter case, we again pick t' such that $(t, ret(enq, k), t') \in \delta_I$. Again, $so_t = so_{t'}$ and $s_q = s_{q'}$ holds and $t' \in fs[q']$ can be observed.

$\langle ii-b-deq \rangle$ Let $(q, ret(deq, d, k), q') \in \delta_A$ and $t \in fs[q]$. We pick t' such that $(t, ret(deq, d, k), t') \in \delta_I$. We know that $so_t = so_{t'}$ and only change is $f_{t'}(k) = D_2$. Since we know that $s_q = s_{q'}$ and $f_{q'}(k) = D_2$, $t' \in fs[q']$ holds.

□

2.1 Herlihy & Wing Queue Linearizability Proof

We will begin with formal description of HW Queue library L_C:

- States of the queue are forms of the tuple $(O_e, O_d, bck, itms, i, x, rng, cp)$ where $O_e, O_d \subseteq \mathcal{O}$ are disjoint sets of operation identifiers of enqueue and dequeue operations, respectively, $bck \in \mathbb{N}$ and $itms : \mathbb{N} \to \mathcal{D}$ are global variables of HW queue, $i : (O_e \cup O_d) \to \mathbb{N}$ and $(O_e \cup O_d) \to \mathcal{D}$ are local variables common to both enqueue and dequeue operations, although they are used for different purposes, $rng : O_d \to \mathbb{N}$ is a local variable for dequeue operations and $cp : \mathcal{O} \to \mathbb{N}$.

- Transition labels alphabet consists of the common $AQ\Sigma$ actions. In addition $enq1$, $enq2$ actions for enqueue and $deq1$, $deq2$, $deq3$ and $deq4$ operations for dequeue operations are added to these transition labels. Semantics of the new actions are derived from the algorithm directly and described in Figure 1.

- Initial state is the configuration $(\emptyset, \emptyset, 0, \emptyset, \emptyset, \emptyset, \emptyset)$ where $\emptyset$ maps every element in the domain to 0 or `EMPTY`, depending on the range or the domain is the empty set for functions.

- Figure 1 describes the derivation rules for L_C based on the HW queue algorithm.

We extract the derivation rules from the algorithm.
Then, we proceed to prove linearizability of L_C.

Lemma 4. *L_C is a q-refinement of L_I.*

Proof. Since L_I is deterministic w.r.t. alphabet $AQ\Sigma$, there exists a forward simulation relation form L_C to L_I if L_C is a refinement of L_I. We will define the relation fs and show that it is a forward simulation relation.

The relation fs relates $q = (O_e, O_d, bck, itms, i, x, rng, cp) \in Q_C$ to the state $q' = (O, <, \ell, rval, cp') \in Q_I$ iff:

1. For all $k \in O_e$, if $cp(k) \in \{1, 2, 3\}$, then $cp'(k) = 1$; if $cp(k) = 4$ then $cp'(k) = 2$. For all $k \in O_d$, if $cp(k) \in \{1, 2, 5\}$, then $cp'(k) = 1$; if $cp(k) = 3$, then $cp'(k) = 2$ and if $cp(k) = 4$, then $cp'(k) = 3$.
2. $k \in O$ iff $k \in O_e$; and $cp(k) \in \{1, 2\}$ or $cp(k) \in \{3, 4\} \wedge itms(i(k)) \neq \texttt{EMPTY}$.
3. $\ell(k) = (d, \texttt{PEND})$ iff $cp(k) \in \{1, 2\}$ or $cp(k) = 3 \wedge itms(i(k)) \neq \texttt{EMPTY}$; and $x(k) = d$ and $k \in O_e$. $\ell(k) = (d, \texttt{COMP}$ iff $k \in O_e$, $cp(k) = 4$, $itms(i(k)) \neq \texttt{EMPTY}$ and $x(k) = d$.
4. $rval(k) = d$ if $k \in O_d$, $cp[k] = 3$ and $x(k) = d$.
5. $<$ is a strict partial order and if second field of $\ell(k)$ is $\texttt{PEND}$ for some $k \in O$, then k is maximal in $<$.
6. For two nodes $m, n \in O$ such that $cp(m), cp(n) > 1$, if $i(m) > i(n)$ then $m \not< n$.
7. Let $m, n \in O$ be two nodes such that $i(m) < i(n)$ and $cp(m), cp(n) > 1$. If there exists a dequeue operation identifier $k \in O_d$ such that $cp(k) \in \{2, 5\}$ such that $i(m) < i(k) \leq i(n)$ and $i(n) < rng(k)$, then $m \not< n$.

Next, we show that fs is a forward simulation relation:

$\langle i \rangle$ $fs[q_{0C}] = \{q_{0I}\}$

$\langle ii - a - enq \rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying CALL-ENQ rule of L_C and $t \in fs[q]$. Premise of CALL-ENQ rule of L_I holds for t due to premise of CALL-ENQ rule of L_C and item 1 in definition of fs. Let t' be the state obtained by applying CALL-ENQ to t. One can observe that $t' \in fs[q']$ by checking the conditions of the fs stated above. Condition 5 holds because the newly added element k which is $\texttt{PEND}$ does not occur on the left side of $<_{t'}$ relation and rules of strict partial order is preserved on $<_{q'}$. Conditions to re-check: 1, 2, 3 and 5.

$\langle ii - a - deq \rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying CALL-DEQ rule of L_C and $t \in fs[q]$. Premise of CALL-DEQ rule of L_I holds for t due to premise of CALL-DEQ rule of L_C and item 1 in definition of fs. Let t' be the state obtained by applying CALL-DEQ to t. One can observe that $t' \in fs[q']$ by checking the conditions of the fs stated above. Conditions to re-check: 1.

$\langle ii-c-enq1\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying ENQ-1 rule of L_C and $t \in fs[q]$. Pick $t' = t$. We still have $t' \in fs[q']$. Conditions 1-5 are trivial to check. Conditions 6 and 7 still hold because $i_{q'}(k) = bck_{q'}$. Conditions to re-check: 1, 2, 3, 6 and 7.

$\langle ii-c-enq2\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying ENQ-2 rule of L_C and $t \in fs[q]$. Pick $t' = t$. One can see that $t' = t \in fs[q']$ by checking conditions 1-7. Conditions to re-check: 1, 2, and 3.

$\langle ii-c-deq1\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying DEQ-1 rule of L_C and $t \in fs[q]$. Pick $t' = t$. One can see that $t' = t \in fs[q']$ by checking conditions 1-7. Nontrivial case is condition 7 and it still holds since one can see that $i_{q'}(k) = 0$, although $rng_{q'}(k)$ is assigned to a new higher value. Conditions to re-check: 1 and 7.

$\langle ii-c-deq2\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying DEQ-2 rule of L_C and $t \in fs[q]$. Pick $t' = t$. One can see that $t' = t \in fs[q']$ by checking conditions 1-7. Conditions to re-check: 1.

$\langle ii-c-deq3\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying DEQ-3 rule of L_C and $t \in fs[q]$. Pick $t' = t$. One can see that $t' = t \in fs[q']$ by checking conditions 1-7. Only non-trivial case is condition 7. Since $i_{q'}(k) = 0$, it cannot impose an extra restriction and it continues to hold. Conditions to re-check: 1 and 7.

$\langle ii-c-deq4\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying DEQ-4 rule of L_C and $t \in fs[q]$. Pick $t' = t$. We still have $t' \in fs[q']$. Only non-trivial case to check is condition 7. Let $m, n \in O$ such that $i_{t'}(m) < i_{t'}(k) \leq i_{t'}(n)$ and $m <_{t'} n$ which violates condition 7. Since condition 7 is not violated while constructing t and applying DEQ-4 increments i by one, the only possible case is $i_{t'}(m) = i_{t'}(k) - 1$ and $m <_t n$ holds. One can see that this information implies $itms_t(i_t(m)) \neq \texttt{EMPTY}$ and $i_t(m) = i_t(k)$. We also know that at the previous step of dequeue with identifier k, $itms(i(k))$ was EMPTY since the operation k moved to control point 5. Hence, the ENQ-2 and RET-ENQ derivations of enqueue operation m must be between the DEQ-2 and current DEQ-5 derivations of k. Then, we can also observe that $i_t(n) = i_{t'}(n) \geq rng_t(n) = rng_{t'} = n$ which contradicts our initial assumption. Conditions to re-check: 1 and 7.

$\langle ii-d\rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying LIN-DEQ rule of L_C and $t \in fs[q]$. Premise of the rule LIN-DEQ1 of L_I holds for t because of the premise of the rule LIN-DEQ of L_A and conditions on L_C. More specifically, one can show that there exists an operation $k' \in O_t$ such that $i_q(k') = i_q(k)$. Conditions 6 and 7 imply that there is no enqueue operation $k'' \in O_t$ such that $cp_q(k'') \in \{2, 3, 4\}$ and $k'' <_t k'$. Condition 3 and 5 imply that there is no enqueue operation $k'' \in O_t$ such that $cp_q(k') = 1$ and $k'' <_t k'$. Last two statements imply that k' is minimal. Let t' be the state obtained by appyling LIN-ENQ1 to t. One can see that

$t' \in fs[q']$ by checking the conditions of fs. Conditions to re-check: 1, 2, 3, 4 and 5.

$\langle ii - b - enq \rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying RET-ENQ rule of L_C and $t \in fs[q]$. We consider two cases while constructing t'. If $k \in O_t$, then premises of RET-ENQ1 rule of L_I holds due to premises of RET-ENQ and conditions of fs and we obtain t' by applying this rule to t. One can see that $t' \in fs[q']$ still holds by checking the conditions. Conditions to re-check:1, 2 and 3. The latter case is $k \notin O_t$. The premises of RET-ENQ2 are still satisfied by t since q satisfies premises of RET-ENQ of L_C and $t \in fs$. We obtain t' by applying RET-ENQ2 to t. One can observe that $t' \in fs[q']$ by checking the conditions on fs. Conditions to re-check: 1.

$\langle ii - b - deq \rangle$ Let $q' \in Q_C$ obtained from $q \in Q_C$ by applying RET-DEQ rule of L_C and $t \in fs[q]$. State t satisfies premises of the rule RET-DEQ of L_I since q satisfies premises of the rule RET-DEQ of L_C and $t \in fs[q]$. We apply RET-DEQ rule of L_I to t to obtain t'. One can see that $t' \in fs[q']$ by checking the conditions on fs. Conditions to re-check: 1.

□

3 Existence of Forward Simulations for Stack Implementations that have Fixed Pop Linearization Points

4 Relaxation for the Data Structures Without Fixed Remove Linearization Points

We have observed that some implementations (like time-stamped stack) do not have fixed remove (pop) linearization points that will correspond to $lin(pop, e, k)$ where e could be a data value or `EMPTY`. However, we observe that, these implementations contain some points in their pop methods that logically removes the element from the pool. We call them commit points. If a method of a library has a fixed linearization point, it is also a commit point. In this sense, commit points are weaker versions of fixed linearization points. Fixed linearization point of a pop preserves the following properties:

- If a *ret* comes before a linearization point in the concrete execution, this order is preserved in the linearization of this execution.
- If a *lin* pop comes before another *lin* pop in the concrete execution, this order is preserved in the linearization of this execution.
- If a *lin* pop comes before an *inv* in the concrete execution, this order is preserved in the linearization of this execution.

A commit point is weaker than a fixed linearization point in the sense that it does not need to satisfy the first and the second conditions.

To our intuition, if a pop (remove) method has multiple finite linearization points, commit point is the latest element in this set. We have never observed an implementation in which a pop is linearized after it logically removes the element from the data structure.

From now on, we will restrict ourselves to stacks, since our example implementation that we will show linearizability of is the time-stamped (TS) stack. However, the notions and the machines we will introduce can be extended to queues easily.

We fix $\mathcal{M} = \{push, pop\}$ and $\mathcal{D} = \mathbb{N} \cup \{\texttt{EMPTY}\}$. We extend the alphabet $A\Sigma$ for stacks with commit points as $ACS\Sigma = A\Sigma \cup \{com(pop, d, k) | d \in \mathcal{D}, k \in \mathcal{M}\}$. We define cs-refinement and cs-linearizability as we defined q-refinement, q-linearizability in the previous sections. We also change definitions of backward and forward simulation relations for stacks with commit points as we do in the previous sections by replacing linearization points with commit points in this extensions. Lemma 1 of the first section still holds with new simulation relation definitions and cs-linearizability and cs-linearizability implies the original linearizability definition.

Our road map for this section is as follows: We will first introduce an intermediate stack machine L_I that will be deterministic with respect to the alphabet $ACS\Sigma$. We will show that L_I is equivalent to the standard abstract stack L_A defined in the previous section with respect to the language $A\Sigma$. We show this by first showing that L_I is a refinement of L_A wrt alphabet $A\Sigma$ by finding a backward simulation relation between them and then L_A is a refinement of L_I with respect to alphabet $A\Sigma$ by finding a forward simulation relation between them. Since L_I is deterministic wrt $ACS\Sigma$, if we have an implementation L_C that is a cs-refinement of L_I, we can find a forward simulation between them. As an example, we will pick L_C as time-stamped stack and establish a forward simulation relation between it and our L_I machine.

Let us continue with defining L_I first:

- A state of L_I again consists of a partial strict order and a program counter: $Q_I \subseteq ND \times ED \times (\mathbb{N} \to Lbl_I)$ where $Lbl_I = \{N, A_0, A_1, R_0, R_1, R_2\}$ is the set of transition labels for the operations. Different from the fixed linearization point case, this time nodes are not triples but 5-tuples of the form (k, d, st, mc, con). First three fields are the same as previous intermediate machines: $k \in \mathbb{N}$ is operation identifier of a push operation, $d \in \mathcal{D}$ is the data value of that push and $st \in \{\texttt{PENDING}, \texttt{CLOSED}\}$ is the current status of the push operation. The fourth field $mc \subset \mathbb{N}$ keeps the operation identifiers of pop operations such that this push was maximally closed when the pop began. A node n is maximally closed in a state s iff $n.st = \texttt{CLOSED}$ and if there is an edge $n \to n' \in ed_s$, then $n'.st = \texttt{OPEN}$. The fifth field $con \subset \mathbb{N}$ is the set of operation identifiers of the pop methods that are concurrent with this push i.e. the either this node was open when the pop started or this node is created when the pop was pending.

- The transition labels consist of invocation and return actions of both methods and commit action for only pop method. Hence $\Sigma_I = ACS\Sigma$. Number of parameters for all actions common in previous intermediate stack machine and this intermediate stack machine are the same. Commit actions contain the second data field. They are of the form: $com(pop, d, k)$ where $d \in \mathcal{D}$ and $k \in \mathbb{N}$.

- Initial state consists of an empty strict partial order and a function mapping ever operation to N: $q_{0I} = (\emptyset, \emptyset, f_{q_{0I}})$ where $f_{q_{0I}}(i) = N$ for all $i \in \mathbb{N}$.

- We define δ_I less formally, by giving verbal explanations to the transitions, omitting the obvious updates on the f part and not mentioning about the parts of the nodes that does not change:

  - $(q, inv(push, d, k), q') \in \delta_I$ iff a new node $n = (k, d, \texttt{PENDING}, \emptyset, con_n)$ is added to $nd_{q'}$, where $con_n = \{i \in \mathbb{N} | f_q(i) = R_0\}$; $n' \to n$ will be added to $ed_{q'}$ if n' is a closed node at the state q.
  - $(q, ret(push, k), q') \in \delta_I$ iff either there is a `PENDING` node n in state q and this node becomes `CLOSED` in state q' or there is no node with identifier k in q and nothing else than f field changes in q'.
  - $(q, inv(pop, k), q') \in \delta_I$ iff for every open node $n \in nd_q$, k is added to $n.con$ in q' and for every maximally closed node $m \in nd_q$, k is added to $m.mc$ in state q'.
  - $(q, com(pop, k, d), q') \in \delta_I$ iff there exists a node n in state q such that $n.d = d$ and either $k \in n.con$ or $k \in n.mc$, this node n and all the edges adjacent to it are removed in the state q', k is removed from all con and mc fields of all nodes in state q' and for all other pop operations k' in $n.mc$ or in $n.con$ and for all states $n' \in nd_q$ such that $n' \to n \in ed_q$ and for all $n'' \in nd_q$, $n' \to n'' \in ed_q$ implies $k' \in n''.con$ (implies $k \notin n''.mc$ but it is stronger than this condition: if there are three closed states p, q, r s.t. $p \to q$, $q \to r$, $p \to r$, k' is only in $r.mc$ and we delete r, former condition only allows $k' \in q.mc$ whereas the latter one allows $k' \in p.mc$ in addition) we have $k' \in n'.mc$ in the state q'. Note that we need to assume data independence to make this action deterministic.
  - $(q, ret(pop, k, d), q') \in \delta_I$ iff $q = q'$ ignoring the f fields.

L_A is the same machine defined in the previous section. The common alphabet between L_A and L_I is $A\Sigma$. We will show that they are equivalent in terms of this alphabet.

Lemma 5. *L_I is a refinement of L_A.*

Proof. We will provide a backward simulation relation bs between states of L_I and states of L_A. Our relation $bs \subseteq Q_I \times Q_A$ relates state $q = (so_q, f_q)$ to $q' = (s_{q'}, f_{q'}$ iff (i) for all operation identifiers $k \in \mathbb{N}$, if $f_q(k) \in \{N, R_1, R_2\}$, then

$f_q(k) = f_{q'}(k)$; if $f_q(k) = A_0$, then $f_{q'}(k) = A_0$ and the data value d associated with this add operation is not inserted to $s_{q'}$ or $f_{q'}(k) = A_1$; if $f_q(k) = A_1$, then $f_{q'}(k) = A_2$; if $f_q(k) = R_0$, then $f_{q'}(k) = R_0$ or $f_{q'}(k) = R_1$; (ii) Let us call a pop operation pending if $f_q(k) = R_0$ and PP_q be set of pending pops. There there exists a function $g : PP_q \to ND_q \cup \{\texttt{NONE}\}$ such that $k \in g(k).mc$ or $k \in g(k).con$ for all pop operation identifiers k such that $g(k) \neq \texttt{NONE}$ and g is one-to-one if we neglect NONE; $s_{q'}$ is obtained by extending so_q to a total order in which pending nodes may not take place and nodes (open or closed) n such that there exists a pop operation with identifier k so that $g(k) = n$ surely do not take place, (iii) $g(k) = \texttt{NONE}$ implies $f_{q'}(k) = R_0$ and $g(k) \neq \texttt{NONE}$ implies $f_{q'}(k) = R_1$, (iv) if $g_{q'}(k) = n$ and n is a pending node, then $f_{q'}(k') = A_1$ where k' is the operation identifier part of n, (v) if there is a pending node with identifier k and it takes part in the linearization, then $f_{q'}(k) = A_1$.

Now, we will show that bs is a backward simulation relation:

$\langle i \rangle$ $bs[q_{0I}] = \{q_{0A}\}$

$\langle ii - a - push \rangle$ Let $(q, inv(push, d, k), q') \in \delta_I$ and $t' \in bs[q']$. We consider two cases: Either the newly added node in q' takes place in the linearization and there exists an index i such that $s_{t'}(i) = d$ or this new node does not exist in the linearization. For the former case construct $s_t = \langle s_{t'}(1), s_{t'}(2), ..., s_{t'}(i-1)\rangle$ and $f_t = f_{t'}$ for all the operations except the ones of which data values are linearized as $s_{t'}(j)$, for $j \geq i$. For those nodes, $f_{t'}(k') = A_1$ whereas we assign $f_t(k') = A_0$ for $j > i$ and $f_t(k) = N$. Let operation identifiers of these nodes be k_j for $j > i$. Then, $t \xrightarrow{\alpha} t'$ holds where $\alpha = inv(push, d, k), lin(push, d, k), lin(push, d_{i+1}, k_{i+1}), ...$ Moreover, $t \in bs[q]$ since s_t is a valid linearization of so_q using the same g function and omitting more open nodes and f_t obeys the conditions. For the latter case, we pick $s_t = s_{t'}$. We have two subcases: There exists a pending pop with identifier k' such that $g(k')$ is the new node with identifier k or not. For the first subcase, we pick $f_t = f_{t'}$ except $f_t(k) = N$ and $f_t(k') = R_0$. Then $txrightarrowinv(push, d, k), lin(pop, d, k')t'$ is a path in L_A and $t \in bs[q]$. For the second subcase, we just pick $f_t = f_{t'}$ except $f_t(k) = N$. Then, it is easy to see that $t \xrightarrow{inv(push,d,k)} t'$ holds and $t \in bs[q]$.

$\langle ii - a - pop \rangle$ Let $(q, inv(pop, k), q') \in \delta_A$ and $t' \in bs[q']$. We will again consider two cases: When relating t' to q', either $g(k) = \texttt{NONE}$ or $g(k)$ is a node in q'. In other words, either the newly invoked pop operation k did not linearize yet or it linearizes and removes an element inserted by a linearized push. The second case also splits into two cases: The element removed by pop k is inserted by a push k' that is still pending or the push has returned. We will look at all three cases separately. The easiest one is the first case. Construct $s_t = s_{t'}$ and $f_t = f_{t'}$ except that $f_t(k) = N$ whereas $f_{t'}(k) = R_0$. One can see that $t \xrightarrow{inv(pop,k)} t'$ is a step in L_A and $t \in bs[q]$. For the first case of the second case, we construct $s_t = s_{t'} \circ \langle d \rangle$ where d is

the data of node identifier k' and $f_t = f_{t'}$ except that $f_t(k) = N$ whereas $f_{t'}(k) = R_1$. One can see that $t \xrightarrow{inv(pop,k),lin(pop,d,k)}$ is a valid path in L_A. Moreover, $t \in bs[q]$ since s_t is a valid linearization of so_q. This is true because the node with identifier k' is a maximal node in so_q and we can add it to the end of linearization of $so_{q'}$. For the second subcase, we obtain s_t from $s_{t'}$ by the following procedure: Let n be the node with identifier k' and k'' be the node such that $k' \to k''$ is an edge in $so_{q'}$, k'' takes part in the linearization of $so_{q'}$ to $s_{t'}$ and it has the minimum index i in the $s_{t'}$ among all such nodes. Then, $s_t = \langle s_{t'}(1), s_{t'}(2), ..., s_{t'}(i-1), d\rangle$ where d is the data value of node with identifier k. Let k_j and d_j be the identifiers and data values of nodes that constitute $s_{q'}(j)$ for $j > i$. Clearly, $t \xrightarrow{\alpha} t'$ is a path in L_A where $\alpha = inv(pop, k), lin(pop, d, k), lin(push, d_i, k_i), ...$ Moreover, $t \in bs[q]$ because s_t is a valid linearization of so_q. It is true because $so_q = so_{q'}$, k' is a maximally closed node in so_q and all the nodes with identifier k_j $(j > i)$ are pending nodes.

$\langle ii - c\rangle$ Let $(q, com(pop, d, k), q') \in \delta_I$ and $t' \in bs[q']$. We will consider two cases. The first case is commit action removes a maximally closed or an open node. For these cases, we can construct t as in the second case of the previous item (invoke pop case). The same arguments apply for constructing a path between t and t' in L_A and showing that $t \in bs[q]$. The second case is that commit action removes a non-maximal closed node. This time, pick $t = t'$. Then, $t \xrightarrow{\epsilon} t'$ is the path in L_A and one can show that $t \in bs[q]$ by choosing $g(k)$ as the node that is removed.

$\langle ii - b - push\rangle$ Let $(q, ret(push, k), q') \in \delta_I$ and $t' \in bs[q']$. We consider two cases. Either there is a node n with identifier k and data value d in state q or not. For the first case, we consider two subcases. Either this node takes part in linearization or not (if there exists a pop k' such that $g(k') = n$). For the first subcase, we can pick $s_t = s_{t'}$ and $f_q = f_{q'}$ except that $f_q(k) = A_0$. Then, $t \xrightarrow{ret(push,k)} t'$ is a path in L_A. Moreover, $t \in bs[q]$ since n is a maximal node in q. For the second subcase, we pick $s_t = s_{t'} \circ \langle d\rangle$ and $f_t = f_{t'}$ except that $f_t(k) = A_1$ and $f_t(k') = R_0$. Then, $t \xrightarrow{lin(pop,d,k'),ret(push,k)} t'$ is a path L_A and $t \in bs[q]$ since n is a maximal open node in q. For the second case, we pick $s_t = s_{t'}$ and $f_t = f_{t'}$ except that $f_t(k) = A_1$. Then, $t \xrightarrow{ret(push,k)} t'$ is a path in L_A and $t \in bs[q]$.

$\langle ii - b - pop\rangle$ Let $(q, ret(pop, d, k), q') \in \delta_I$ and $t' \in bs[q']$. We pick $s_t = s_{t'}$ and $f_t = f_{t'}$ except that $f_t(k) = R_1$. One can see that $t \xrightarrow{ret(pop,d,k)} t'$ is a valid action in L_A and $t \in bs[q]$.

□

Lemma 6. *L_A is a refinement of L_I.*

Proof. We will construct a forward simulation relation fs between L_A and L_I. Our relation $fs \subseteq Q_A \times Q_I$ relates state $q = (s_q, f_q) \in Q_A$ to a state $q' = (so_{q'}, f_{q'}) \in Q_I$ iff (i) for all operation identifiers $k \in \mathbb{N}$, if $f_q(k) \in \{N, A_0, R_0, R_1, R_2\}$ then $f_{q'}(k) = f_q(k)$; if $f_q(k) = A_1$, then $f_{q'}(k) = A_0$; if $f_q(k) = A_2$, then $f_{q'}(k) = A_1$; (ii) We form nodes of $so_{q'}$ ($ND_{q'}$ as follows: If k is a push operation adding data value d and either $f_q(k) = R_0$ or $f_q(k) = R_1$ and data added by this push exists in s_q, then there is a `PENDING` node in $ND_{q'}$ with identifier k and data value d. If $f_q(k) = R_2$, then there is a `CLOSED` node in nd_q with identifier k and data value d. If there is an operation identifier k such that $f_q(k) = R_0$, we call this a pending pop and this pop takes place mc or con fields of nodes of q'. If $n \in ND_{q'}$ is a `PENDING` node, then $k \in n.con$. If n is maximally closed, then $k \in n.con$ or $n.mc$. If $k \in n.mc$ or $k \in n.con$ and there exists another node $n' \in ND_{q'}$ such that $n \to n' \in ED_{q'}$, then $k \in n'.con$. If $k \in n.mc$ and there exists another node $n' \in ND_{q'}$ such that $n' \to n \in ED_{q'}$, then neither $k \in n'.mc$ nor $k \in n'.con$. For any node $n \in ND_{q'}$, either $k \in n.mc$ or $k \in n.con$. (iii) We form edges of $so_{q'}$ ($ED_{q'}$) as follows: `PENDING` nodes are maximal. Edges obey the strict partial order conditions. (iv) We can find a linearization $so_{q'}$ that is equal to s_q. `PENDING` nodes may not participate in the linearization. Note that we do not need g function for keeping track of linearized pops unlike the previous proof.

Next, we will show that fs is a forward simulation relation.

$\langle i \rangle$ $fs[q_{0A}] = \{q_{0I}\}$

$\langle ii - a - push \rangle$ Let $(q, inv(push, d, k), q') \in \delta_A$ and $t \in fs[q]$. Pick t' such that $(t, inv(push, d, k), t') \in \delta_I$. Since $s_q = s_{q'}$ and $so_{t'}$ contains a maximal open new node as the only difference from so_t, we can linearize $so_{t'}$ so that linearization is equal to $s_{q'}$. By checking the other conditions, one can observe that $t' \in fs[q']$.

$\langle ii - a - pop \rangle$ Let $(q, inv(pop, k), q') \in \delta_A$ and $t \in fs[q]$. Pick t' such that $(t, inv(pop, k), t') \in \delta_I$. Only difference between t and t' is that maximally closed and open nodes in t' contain k in their mc or con fields. Since this new addition obeys our forward simulation definition, $t' \in fs[q']$.

$\langle ii - c - push \rangle$ Let $(q, lin(push, d, k)q') \in \delta_A$ and $t \in fs[q]$. Pick $t' = t$. $s_{q'}$ is still a linearization of so_t since the node with identifier k is a maximal node in ND_t and we can linearize it at the end. So, $t' \in fs[q']$ holds.

$\langle ii - c - pop \rangle$ Let $(q, lin(pop, d, k), q') \in \delta_A$ and $t \in fs[q]$. Pick t' such that $(t, com(pop, d, k)t') \in \delta_I$. The action $com(pop, d, k)$ is a valid action in L_I because d is the last element in s_q. Hence, the node n with identifier k and data value d is a maximal element in so_q and either $k \in n.mc$ or $k \in n.con$ by the properties of fs. Hence, the node with identifier k can be removed by a com action. In addition, $s_{q'}$ is a linearization of $so_{t'}$ because removed node is a maximal node and $s_{q'}$ is obtained from s_q by deleting the maximum node. Hence, $t' \in fs[q']$ holds.

$\langle ii-b-push\rangle$ Let $(q, ret(push, k), q') \in \delta_A$ and $t \in fs[q]$. We will consider two cases: either the element inserted by the push with identifier k is removed by a concurrent pop or the element is still in s_q. For the former case, there is no node with identifier k in state t and we can pick t' such that $(t, ret(push, k), t') \in \delta_I$. We have $so_t = so_{t'}$ for this case. Since $s_q = s_{q'}$ also holds, $s_{t'}$ is a linearization of $so_{q'}$ and $t' \in fs[q']$. For the latter case, we again pick t' such that $(t, ret(push, k), t') \in \delta_I$ holds. This time, only difference between so_t and $so_{t'}$ is that the node with identifier k is `CLOSED` in $so_{t'}$. Since the edges are the same, $s_{q'}$ is a valid linearization of $s_{t'}$ and $t' \in fs[q']$ holds.

$langleii-b-pop\rangle$ Let $(q, ret(pop, d, k), q') \in \delta_A$ and $t \in fs[q]$. We pick t' such that $(t, ret(pop, d, k)t') \in \delta_I$. Only difference between t and t' is that $f_{t'}(k) = R_2$ whereas $f_t(k) = R_1$. Since $s_q = s_{q'}$ and $so_t = so_{t'}$, $s_{q'}$ is a valid linearization of $so_{t'}$. By checking the other conditions, we see that $t' \in fs[q']$.

□

Now, we show that TS-Stack is linearizable by showing the concrete TS-Stack implementation L_C is a cs-refinement of L_I. As L_C, we pick the simplest version that omits the `EMPTY` returns of the pop methods, does not allow unlinking and elimination.

First, we describe the TS-Stack algorithm:

```
struct Node{
  int data;
  int ts;
  Node* next;
  boolean taken;
};
Node* pools[maxThreads];
int TS = 0;

void push(int x) {
  Node* n = new Node(x,MAX_INT,
                          null,false);
  n->next = pools[myTID];
  pools[myTID] = n;
  int i = TS++;
  n->ts = i;
}
int pop() {
 boolean success = false;
 int maxTS = -1;
 Node* youngest = null;
 while ( !success ) {
   maxTS = -1; youngest = null;
```

```
    for(int i=0; i<maxThreads; i++){
      Node* n = pools[i];
      while (n->taken && n->next != n)
        n = n->next;
      if(maxTS < n->ts) {
        maxTS = n->ts; youngest = n;
      }
    }
    if (youngest != null)
      success=CAS(youngest->taken,
                          false,true);
  }
  return youngest->data;
}
```

Then, we begin to obtain the LTS (TSS) of TS Stack from the algorithm by defining control points and the actions among these control points as seen in Figure X Flow diagram is referenced here. To simplify the proof, we take the initializations of some local variables together as atomic.

States of the TS-Stack contains the global variables and local variables as fields. Global variables are just elements of their domains and local variables are maps from operation identifiers to their domains. We say $i_q(k)$ for referencing the value of local variable i of operation k in state q. There is only one special local variable called $myTID$. Its value is unique to each pending operation in a state i.e., concurrent operations cannot have the same $myTID$ value. TS-Stack states also contains sets $O_a, O_r \in \mathbb{O}$ which are operation identifier sets of push and pops respectively, and the control point function cp which is a map from operation identifiers to the control points set that are presented in the flow diagram Figure X Flow diagram is referenced here. Transition relation of the TS-STack is presented in Figure **??** (push rules) and Figure **??** (pop rules). Next, we show that the linearizability of TS Stack.

Lemma 7. *TSS is a Com(pop)- refinement of $AbsS$.*

Proof. Since $AbsS$ is deterministic with respect to $C \cup R \cup Com(pop)$, there exists a Com(pop) forward simulation from TSS to $AbsS$ iff TSS is a Com(pop)-refinement of $AbsS$. Hence, we define the relation fs and show that it is a Com(pop)-forward simulation relation.

Let us make some clarifications before defining the relation. In order not to confuse nodes in TS Stack and nodes in $AbsS$, we call nodes of $AbsS$ as vertices from now on. We also define ordering relation (called traverse order) among the operations in a state of TS. It basically reflects the traverse order of pop operations. For two push operations $m, n \in O_a$ is state s we say that $m <_s^{tr} n$ iff either $myTid(m) < myTid(n)$ or $myTid(m) = myTid(n)$ and $n_s(n)$ is reachable from $n_s(m)$ using next pointers. $\geq^{tr}$ is obtained from $<^{tr}$ in the usual way.

The relation $fs \subseteq Q_C \to Q_{AbsS}$ contains (s, t) iff the following are satisfied:

Nodes $k \in O_t$ iff k is a push operation in s ($k \in O_a$) such that either it has not inserted its node to the pool yet ($cp_s(k) = A_i$ and $i < 3$) or its node is not taken by a pop ($cp_s(k) = A_i$, $i \geq 3$ and $n_s(k) -> taken = false$).

Pend/Comp A vertex $k \in O_t$ is pending ($\ell_t(k) = (d, \texttt{PEND})$) iff k satisfies the previous condition, $x_s(k) = d$ and it is not completed in s ($cp_s(k) = A_i$ and $i < 6$). Similarly, this vertex is completed ($\ell_t(k) = (d, \texttt{COMP})$) iff k satisfies the previous condition, $x_s(k) = d$ and it is completed in s ($cp_s(k) = A_6$). Pending vertices are maximal with respect to $<_t$ i.e., if $k \in O_t$ is a pending vertex, then for all $k' \in O_t$ $k \not<_t k'$.

TSOrder If a node has a smaller timestamp than the other node in s, the operations that inserted them cannot be ordered reversely in t. More formally, let $k, k' \in O_t$ s.t. $n_s(k) -> ts \leq n_s(k') -> ts$. Then, $k' \not<_t k$.

TidOrder Order among the nodes inserted by the same threads in s must be preserved among the operations that inserted them in t. Let $k, k' \in O_t$ s.t. $myTid_s(k) = myTid_s(k')$ and $n_s(k) -> ts < n_s(k') -> ts$. Then, $k <_t k'$.

Frontiers Every maximally closed or pending vertex can be removed by a pending pop. More formally, let $k \in O_t$ such that $\ell_t(k) = (\_, \texttt{PEND})$. Then, for all pops p, $k \in ov_t(p)$. In the other case, let $k \in O_t$ such that $\ell_t(k) = (\_, \texttt{COMP})$ and for all other $k' \in O_t$ such that $k <_t k'$, we know $\ell_t(k') = (\_, \texttt{PEND})$. Then, for all pop operations p, $k \in be_t(p)$ or $k \in ov_t(p)$.

MaximalOV If a push $k \in O_t$ is a candidate to be removed by a pop p, then every other push k' invoked after k is a candidate to be removed by p since k is concurrent with p. More formally, let $k, k' \in O_t$ such that $k <_t k'$ and there exists a pop p such that $k \in be_t(p)$ or $k \in ov_t(p)$. Then, $k' \in ov_t(p)$.

MinimalBE If a push $k \in O_t$ has finished before the pop p is invoked and yet k is a candidate to be removed by p, other pushes completed before k can not be candidates to be removed by p at that state. More formally, let $k, k' \in O_t$ such that $k <_t k'$ and there exists a pop p such that $k' \in be_t(p)$. Then, neither $k \in be_t(p)$ nor $k \in ov_t(b)$.

ReverseFrontiers If a push $k \in O_t$ is concurrent with the pop p and there exists another push $k' \in O_t$ that is immediate predecessor of k, then k' is either concurrent or maximally closed with respect to p. More formally, let $k, k' \in O_t$ such that $k' \in pred_{<_t}(k)$ and $k \in ov_t(p)$ for some pop p. Then, either $k' \in ov_t(p)$ or $k' \in be_t(p)$.

TraverseBefore If a pop operation p is currently visiting node n and there is a node m coming before n in the traverse order with a greater timestamp, then the operation that inserts m must be concurrent with p. More formally, let $k \in O_t$ such that $n_s(k) <_s^{tr} n_s(p)$ and $n_s(k) -> ts \geq n_s(p) -> ts$. Then, $k \in ov_t(p)$.

TraverseAfter If a pop operation p is currently visiting node n that is not null and its youngest element m is not null and still not taken in state s, then either m is a candidate to be removed by p in t or there exists a later node m' than n such that m' is a candidate in t and it has a bigger timestamp than n. More formally, assume that there exists $k, k' \in O_t$ such that $youngest_s(p) -> taken \neq false$, $youngest_s(p) = n_s(k)$ and $n_s(k') = n_s(p)$. Then, either $k \in ov_t(p) \vee k \in be_t(p)$ or there exists $k'' \in O_t$ s.t. $n_s(k'') -> ts > n_s(k) -> ts$ and $k'' \in ov_t(p) \vee k'' \in be_t(p)$ and either $k' <_s^{tr} k''$ or $n_s(p) = n_s(k'') \wedge cp_s(p) = R_j \wedge j > 3$.

Next, we will show that fs is really a Com(pop)-forward simulation relation. Except the trivial base case, we case-split on the transition rules. We first assume $(s, \alpha s') \in \delta_C$ and $t \in fs[s]$. Then, we find corresponding transition $\alpha' \in \Sigma_{AbsS}$ obeying the Com(pop)-forward simulation relation conditions and obtain t' such that $(t, \alpha' t') \in \delta_{AbsS}$ and $t' \in fs[s']$.

We observe that if $\alpha \in C \cup R \cup Com(pop)$, then the corresponding rule in $AbsS$ is $\alpha' = \alpha$. Otherwise, $\alpha' = \epsilon$.

Let the following derivation rule of TSS be the one for describing α:

$$\frac{\psi}{s \xrightarrow{\alpha} s'}$$

and the following one be the derivation rule of $AbsS$ describing α' if $\alpha' \neq \epsilon$ (equivalently $\alpha' = \alpha$):

$$\frac{\psi'}{q \xrightarrow{\alpha'} q'}$$

For the cases $\alpha' = \alpha$, we first need to show α' is enabled in state t i.e., t satisfies ψ'. If this can not be directly obtained from s satisfies ψ and using one or two obvious conditions on fs (since $t \in fs[s]$), we show the derivation in the proof. Then, t' is obtained in a unique way since $AbsS$ is deterministic on its alphabet $\Sigma_{AbsS} = C \cup R \cup Com(pop)$. The, only other thing to show is $t' \in fs[t']$. We show this by proving that t' does not violate any of the conditions of the fs described above. We only explain why the new instantiations due to the difference between s' and s or the difference between t' and t do not violate the conditions. We skip the instances that we assumed while relating s to t.

For the cases in which $\alpha' = \epsilon$, we have $t' = t$ and the only thing to show is $t \in fs[s']$. Again, we only explain why the new instantions due to the difference between s' and s do not violate the conditions.

CALL-PUSH The same derivation rule of TSS is applied to t obtain t'. The premise of the rule is satisfied by t trivially in the sense explained before. The new vertex k is added to the O_t such that k is maximal, pending and every completed vertex is ordered before k in t'. Moreover, k is overlapping with every pending pop. To see that $t' \in fs[s']$ we observe the following: *Nodes* condition is preserved because $k \in O_{t'}$. Since the newly added vertex k is

maximal and pending in t', *Pend/Comp* condition is preserved. *Frontiers* and *MaximalOV* conditions are not violated since k is added to $ov(p)$ set for every pending pop operation p.

PUSH1 We have $t' = t$ and show $t \in fs[s'].Nodes$ and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex. *TSOrder* is still preserved. Timestamp of $n_{s'}(k)$ is maximal and every other nodes of push operations with maximal timestamp in s' are pending vertices in t. Hence there can be no ordering between those pushes and k in t that can violate *TSOrder*. Moreover, k is maximal in t which means that it cannot be ordered before another push k' of which node has a lower timestamp. *TidOrder* is also satisfied. Since k is ordered after every completed push in t and every other push by the same thread is completed, ordering required by the *TidOrder* is present.

PUSH2 We have $t' = t$ and show $t \in fs[s']$. *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex. One can also see that the *TraverseBefore* condition is preserved. Let the pop p visiting node m and $n_{s'}(k) <^{tr}_{s'} m$. Since k and p are both pending in s and $t \in fs[s]$, $k \in ov_t(p)$ (by the *Frontiers* condition). Hence, *TraverseBefore* is preserved.

PUSH3 We have $t' = t$ and show $t \in fs[s']$. We consider two cases: $n_s(k) - > taken$ is `true` or it is `false`. For the former case, $k \notin O_t$. The only new instantiation we check is $k \notin O_t$ does not violate *Nodes* condition while relating s' to t. For the latter case, we have $k \in O_t$. *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex after changing s to s'.

PUSH4 We have $t' = t$ and show $t \in fs[s']$. We consider two cases: $n_s(k) - > taken$ is `true` or it is `false`. For the former case, *Nodes* condition is still satisfied since k remains to be not a vertex. For the latter case *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex. *TSOrder* condition is still not violated since if $k' <_t k$, then k' is a completed vertex in s and s'. By the premise of the rule (which can be shown to hold for every operation at control point A_4) $i_s(k') < i_s(k)$ and consequently $n_{s'}(k') - > ts < n_{s'}(k) - > ts$. Since every other push by the thread of k is completed, *TidOrder* still continues to hold for the same reasons. *TraverseAfter* condition is also preserved. Let k' be the push and p be the pop such that $n_s(k') = youngest_s(p)$, $n_s(k') \leq^{tr}_s n_s(k)$, $n_s(k') - > ts < n_s(k) - > ts$ and $k \in ov_t(p)$ or $k \in be_t(p)$. Assume $n_{s'}(k') - > ts \geq N_{s'}(k) - > ts$ after the action. Then, k' must be a pending push both in s and s' by the premise of the derivation rule and $k' \in ov_t(p)$ must be true by *Frontiers* condition and $t \in fs[s]$. Hence, the *TraverseAfter* condition is preserved.

RET-PUSH We consider two cases, $n_s(k) - > taken$ is `false` or `true`. For the former case, we obtain t' by applying RET-PUSH1 rule of *AbsS*. *Nodes* and *Pen-*

d/Comp conditions are still satisfied since k becomes a completed vertex in t'. *Frontiers* condition still holds since although k become a maximally closed vertex in t', we have $k \in ov_{t'}(p)$ for all pending nodes p (due to *Frontiers* condition, $t \in fs[s]$ and k was a pending operation in state t, $k \in ov_t(p)$).

For the latter case, we obtain t' by applying RET-PUSH2 rule of *AbsS*. *Nodes* condition is still satisfied since $k \notin O_{t'}$.

□

CALL-ENQ

$$\frac{k \notin dom(cp) \qquad d \neq \texttt{EMPTY}}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{inv(enq,d,k)} O_e \cup \{k\}, O_d, bck, itms, i[k \mapsto 0], x[k \mapsto d], rng, cp[k \mapsto 1]}$$

ENQ-1

$$\frac{k \in O_e \qquad cp(k) = 1}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{enq1(k)} O_e, O_d, bck + 1, itms, i[k \mapsto bck], x, rng, cp[k \mapsto 2]}$$

ENQ-2

$$\frac{k \in O_e \qquad cp(k) = 2}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{enq2(k)} O_e, O_d, bck, itms[i(k) \mapsto x], i, x, rng, cp[k \mapsto 3]}$$

RET-ENQ

$$\frac{k \in O_e \qquad cp(k) = 3}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{ret(enq,k)} O_e, O_d, bck, itms, i, x, rng, cp[k \mapsto 4]}$$

CALL-DEQ

$$\frac{k \notin dom(cp)}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{inv(deq,k)} O_e, O_d \cup \{k\}, bck, itms, i[k \mapsto 0], x[k \mapsto 0], rng[k \mapsto 0], cp[k \mapsto 1]}$$

DEQ-1

$$\frac{k \in O_d \qquad cp(k) = 1}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{deq1(k)} O_e, O_d, bck, itms, i, x, rng[k \mapsto bck - 1], cp[k \mapsto 2]}$$

DEQ-2

$$\frac{k \in O_d \qquad cp(k) = 2 \qquad itms[i(k)] = \texttt{EMPTY}}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{deq2(k)} O_e, O_d, bck, itms, i, x, rng, cp[k \mapsto 5]}$$

LIN-DEQ

$$\frac{k \in O_d \qquad cp(k) = 2 \qquad itms(i(k)) \neq \texttt{EMPTY}}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{lin(deq,items[i(k)],k)} O_e, O_d, bck, itms[i(k) \mapsto \texttt{EMPTY}], i, x[k \mapsto itms[i(k)]], rng, cp[k \mapsto 3]}$$

DEQ-3

$$\frac{k \in O_d \qquad cp(k) = 5 \qquad i(k) = rng(k) - 1}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{deq3(k)} O_e, O_d, bck, itms, i[k \mapsto 0], x, rng, cp[k \mapsto 1]}$$

DEQ-4

$$\frac{k \in O_d \qquad cp(k) = 5 \qquad i(k) < rng(k) - 1}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{deq4(k)} O_e, O_d, bck, itms, i[k \mapsto i(k) + 1], x, rng, cp[k \mapsto 2]}$$

RET-DEQ

$$\frac{k \in O_d \qquad cp(k) = 3}{O_e, O_d, bck, itms, i, x, rng, cp \xrightarrow{ret(deq,x,k)} O_e, O_d, bck, itms, i, x, rng, cp[k \mapsto 4]}$$

Figure 1: The transition relation of HWQ.

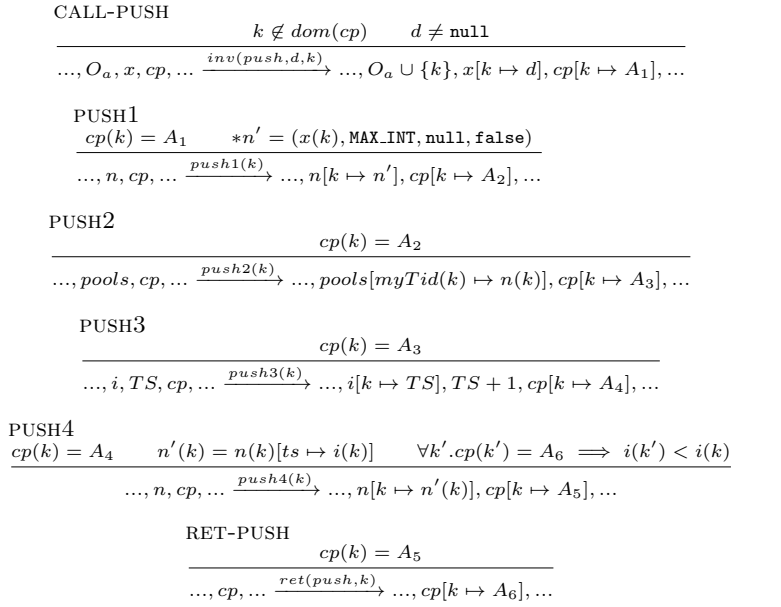

Figure 2: The push derivation rules of TSS. We only mention the state components that are modified. Unmentioned state components have the names in the algorithm in the prestate. $*n = (a, b, c, d)$ is shorthand for $n- > data = a$, $n- > ts = b$, ... $n' = n[ts \mapsto expr]$ is short for $n'- > ts = expr$ and all the other fields of n and n' are the same.

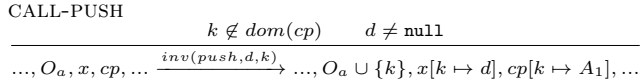

Figure 3: The pop derivation rules of TSS. We only mention the state components that are modified. Unmentioned state components have the names in the algorithm in the prestate. $*n = (a, b, c, d)$ is shorthand for $n- > data = a$, $n- > ts = b$, ... $n' = n[ts \mapsto expr]$ is short for $n'- > ts = expr$ and all the other fields of n and n' are the same.