A Mechanized Refinement Proof of the Chase-Lev Deque using a Proof System

Suha Orhun Mutluergil and Serdar Tasiran

Koc University, Istanbul, TURKEY

Abstract. We present a linearizability proof for the Chase-Lev work-stealing queue (WSQ) on sequentially consistent (SC) memory. We used the CIVL proof system for verifying refinement of concurrent programs. The lowest-level description of the WSQ is the data structure code described in terms of fine-grained actions whose atomicity is guaranteed by hardware. Higher level descriptions consist of increasingly coarser action blocks obtained using a combination of Owicki-Gries (OG) annotations and reduction and abstraction. We believe that the OG annotations (location invariants) we provided to carry out the refinement proofs at each level provide insight into the correctness of the algorithm. The top-level description for the WSQ consists of a single atomic action for each data structure operation, where the specification of the action is tight enough to show that the WSQ data structure is linearizable.

Keywords: Chase-Lev deque, Owicki-Gries method, reduction, abstraction, refinement, linearizability, static verification

1 Introduction

Work stealing is a widely accepted and applied method for scheduling tasks used by many programming languages, run-time systems and frameworks that support distribution of computation into tasks in shared-memory parallel programs. Work stealing queue data structures constitute the core of this method. The queue keeps a pool of tasks to be executed and provides methods to threads for putting and taking tasks from the pool. The WSQ algorithm must provide certain guarantees such as the same task cannot be scheduled twice or given sufficient number of requests, all the tasks in the pool are scheduled. These guarantees are vital for the correct functioning of the system.

In this study, we verify the Chase-Lev WSQ algorithm [2], a widely-used non-blocking algorithm, by providing a linearizability proof for its sequentially consistent (SC) executions. Starting with fine-grained concurrent method bodies, we obtain atomic method abstractions. Those abstractions are tight enough to show that the WSQ algorithm satisfies the desired properties.

The proof is performed using the CIVL proof system [5] and it has four layers[1]. At the bottom layer, method bodies consist of fine-grained atomic statements supported by most hardware and programming languages. In the following

[1] CIVL proof files can be obtained from: http://msrc.ku.edu.tr/projects/chase-lev-wsq/.

layers, atomic blocks inside the method bodies grow using abstraction, reduction and location annotations until we obtain the desired abstract atomic bodies of the methods at the fourth layer (depicted in the top rows of Figures 2 and 3).

This study has the following contributions and results:

– We present the first mechanized linearizability proof of the Lev-Chase work stealing queue algorithm for its SC executions using a proof system.
– Obtaining correct location and mover annotations for the fine-grained method bodies require reasoning about all possible interleavings of the program. We believe that the proof annotations at lower layers provide insight about the behavior of SC executions of this program.
– Our proof is based on two important techniques: Owicki-Gries [11] and Lipton's reduction/abstraction [8] method. We show that the combined use of these two techniques is powerful, and each is best suited to carry out certain parts of the reasoning.

Section 2 gives an overview of the Chase-Lev work stealing queue algorithm. In Section 3, we give a brief information about the proof techniques we utilized. Details of the mechanized proof are presented in Section 4. We compare our work with related studies in Section 5 and finish with closing remarks and future work in Section 6. Some observations on the WSQ algorithm that will be useful for our proofs are put on Appendix A. Initial abstractions/simplifications on the algorithm applied before the mechanized proof are explained in Appendix B.

2 The Chase-Lev Work Stealing Queue Algorithm

Operations effecting one of the worker thread's queue in the Chase-Lev WSQ algorihtm is presented in Figure 1 using the programming language CIVL.

Shared variables H, T and *items* represent the current head (top), tail (bottom) and the task pool, respectively. Tasks are assumed to be of type *int*. The *items* has an infinite domain. Hence, it is never required to resize it due to an overflow and we do not need to think of it as a circular array.

The `put` and `take` methods are executed exclusively by the worker thread (called `ptTid` in short from now on). The `put` method adds one more element to the tail of the queue.

The `take` method first reserves the last element by decrementing T by one (Line 2). If it observes an empty queue, then it increments T back and returns an `EMPTY` task (if block at Line 4). If it observes more than one elements in the queue, it returns the element at index T (if block at Line 9). If there is a single element in the queue, the `ptTid` tries to take it by a `CAS` operation (Line 11).

The `steal` method is executed by a stealer thread. If it sees the queue empty, it returns `EMPTY` task (if block at Line 4). Otherwise, it iteratively tries to steal an element by incrementing H by one via a `CAS` statement. If `CAS` is successful, then `steal` returns successfully with the element at index H (If block at Line 9). If `CAS` is not successful, then the current element at index h is stolen or taken. Hence, `steal` tries to steal another element in a new iteration.

```
H:int;
T:int;
items:[int]int;

put(task:int)
{
  var t:int;

t := T;
items[t] := task;
T := t+1;
return;
}
```

```
take():(task:int)
{
  var h,t:int;
  var chk:bool;

  t := T-1;
  T := t;
  h := H;
  if(t<h)
  {
   T := h;
   task := EMPTY;
   return;
  }
  task := items[t];
  if(h<t)
  {
   return;
  }
  [if(h==H)
  {
   H := h+1;
   chk := true;
  }
  else
  {
   chk := false;
  }]
  if(!chk)
  {
   task := EMPTY;
  }
  T := t+1;
  return;
}
```

```
steal():(task:int)
{
  var h,t:int;
  var chk:bool;

  while(true)
  {
   h := H;
   t := T;
   if(h>=t)
   {
    task := EMPTY;
    return;
   }
   task := items[h];
   [if (h==H)
    {
     H := h+1;
     chk := true;
    }
    else
    {
     chk := false;
    }]
   if(chk)
   {
    return;
   }
  }
  return;
}
```

Fig. 1. Chase-Lev Work Stealing Queue Algorithm

The behavior of the methods explained above is easily provable if they execute sequentially. However, we assume SC setting such that execution of methods could be interleaved with operations of other threads but operations of the same thread appear in the sequence of program order to itself and other threads. SC is one of the strongest guarantees that can be given for a concurrent program. Yet it is still more difficult to reason about program correctness in SC then in sequential setting since one needs to consider all possible thread interleavings. We present more detailed observations about the SC executions of the WSQ algorithm in Appendix A.

Our linearizability proof begins with a slightly modified version of the WSQ algorithm presented in Figure 1 based on valid abstractions/simplifications explained in Appendix B.

3 Overview of Proof Methodology

In this section, we provide some important techniques we utilize in our proof and supported by the proof system CIVL. We only give high-level definitions and explain how we utilized them. Formal definitions of the concepts described here can be found in [3] and [5].

The language CIVL is specially developed for verification purposes. It allows usual constructs existing in many imperative programming languages and some additional constructs for verification purposes. A CIVL program consists of method bodies and atomic actions. Method bodies contain usual imperative constructs like assignments, sequencing, conditional statements, loops, method calls, thread creation and atomic action calls. Atomic actions consist of single-state location annotations and two-state transition relations.

The method bodies are partitioned into steps. CIVL allows programmers to decide on the granularity of a step in a method body. Hence, a step may contain multiple statements. Atomic actions are single step. We denote steps inside brackets in this paper. A program executes by picking a thread non-deterministically and executing a non-deterministic number of next steps of this thread. An execution is obtained using the SC memory model by interleaving steps from different threads. If the location annotation of an atomic action does not hold in a state of an execution just before executing this action, the program fails. The program is safe if no execution fails the location annotation of an atomic action.

To check safety of a program, CIVL utilizes Owicki-Gries (OG) reasoning [11]. The OG checks two things: i) a location annotation holds after a thread takes a step (sequential correctness) and ii) the location annotation is preserved by concurrent threads (non-interference). In addition to the location annotations, CIVL allows programmers to write method pre- and post-conditions. They are also checked via OG reasoning. Moreover, programmers can write conditions to be satisfied inside the atomic blocks as assert statements. Correctness of these statements are checked again by OG reasoning without the non-interference part.

CIVL enables programmers to grow atomic steps inside the procedure bodies using a technique called reduction. To achieve this, each atomic action is annotated with R, L, N or B tags standing for right-, left-, non- and both-movers. A sequence of steps that begins with a sub-sequence of right- or both-movers, followed by an optional non-mover, followed by a sub-sequence of left-movers or both-movers could form a single atomic step. CIVL also performs a check to validate that the atomic action conforms to its mover type. An action A is a right-mover if executing A first and then an action B from another thread in all executions can be simulated by first executing B and then A. A dual definition applies for left-movers. An action is a both-mover (non-mover) if it is both (neither) right-mover and (nor) left-mover.

A layer in CIVL is a program and CIVL performs refinement proofs in a sequence of layers. While moving from one layer to the next layer, CIVL allows programmers to abstract atomic actions so that they have more behavior in the next layer. Hence, they have more relaxed location annotations and they can make actions from other threads mover and grow the steps of method bodies in the next layer. For instance, *havoc* is a keyword in CIVL, used for assigning non-deterministic values to variables. *havoc* x action abstracts $x := t$ action since the former statement allows variable x to have a range of values including the latter value t.

Another option that a programmer can benefit from between layers is the method abstraction. It allows programmers to replace a method body with a single atomic action. This method enables programmers to increase granularity of the program by replacing fine-grained method bodies with a single coarser action block. CIVL also performs a check between layers to validate method abstraction. An atomic action A abstracts a method body B if and only if A abstracts a single step of B in all possible execution paths of B and all the other steps of B in this execution path refines *skip*. Moreover, the return variable must not be modified after the step that refines A. From now on, we call the step that refines A as the action block.

CIVL validates that the program at the bottom layer refines the program at the top-layer if the OG and mover checks pass for all layers and action abstraction and method abstraction checks pass between all layers.

CIVL allows the programmers to use primitive types like boolean or integer and allow users to define their own types. Moreover, it supports linear variables. Difference of a linear variable from a regular variable is that value inside a linear variable cannot be duplicated. Since thread identifiers are unique, we use linear thread identifiers. We modify methods such that each thread gets a linear variable *tid* as input. For `take` and `put` methods, we know that *tid* must have the value `ptTid` and for `steal` method, *tid* must be different than `ptTid`. This additional information makes CIVL know that `put` and `take` methods cannot be concurrent whereas `steal` can be concurrent with other methods and itself.

We heavily utilize the techniques above in our proof. In all layers, we provide location annotations that show the relation between the global variables H and T. Those annotations play a crucial role during the method abstractions between layers and mover checks inside the layers. We start with a relatively complicated relation between those global variables in fine-grained method bodies of lower layers. As the methods get coarser at later layers, we establish $H \leq T$ as a global location annotation.

At the end of Layer 0, we abstract some of the atomic actions so that steps inside the method bodies of `put` and sub-methods of `take` could grow bigger via reduction. Consequently, method bodies of `put` and sub-methods of `take` contain single steps at the end of Layer 1. Then, we can turn them into atomic actions in Layer 2 by applying method abstraction. Since all of the sub-methods of `take` turn into atomic actions at Layer 2, we can use method abstraction at the end of Layer 2 to obtain the desired atomic action of `take` at Layer 3. Moreover, $H \leq T$ becomes a global location invariant at Layer 2 and it enables us to use method abstraction at the end of Layer 2 on the `steal` method to obtain desired atomic action of `steal` at Layer 3.

4 Mechanized Proof Steps

In this section, we present the mechanized proof of the WSQ algorithm. A schematic of the proof is given in Figures 2 and 3. Before diving into details of the proof, we provide a brief explanation about the programs in these figures. Numbers inside the method bodies correspond to steps or control points of those methods. Atomic actions are written inside the brackets. We may omit the brackets if the atomic action consists of a single statement. Location annotations of atomic actions are given between $\langle$ and $\rangle$ symbols. If a location annotation has

Layer	steal(linear tid:Tid):(task:int)	put(linear tid:tid, task:int)
3	⟨stTid ∧ $H \leq T$ ∧ !tics⟩	
	[goto 1A, 1B;	
	1A: assume H<T;	
	task := items[H];	⟨ptInv()⟩
	H :=H+1;	[items[T] := task;
	return;	T := T+1;]
	1B: assume H<=T;	
	task :=EMPTY;	
	return;]	
2	pre: ⟨stTid(tid) ∧ ticsCond() ∧ !tics⟩	
	post: ⟨stTid(tid) ∧ ticsCond() ∧ !tics⟩	
	{	
	1 ⟨... ∧ !tics⟩	
	h := H;	
	2 ⟨... ∧ !tics⟩	
	[if(h<T)	
	assume h==H ==> H<T;	
	else	
	assume h>=y;]	
	3 if(h >= t)	
	{	
	4 task := EMPTY;	⟨ptInv()⟩
	5 ⟨... ∧ !tics⟩ return;	[items[T] := task;
	}	T := T+1;]
	6 ⟨... ∧ !tics⟩	
	[assert !tics && h==H ==> H<T;	
	if(h==H)	
	task := items[h];	
	else	
	havoc(task);]	
	7 ⟨... ∧ !tics ∧	
	(H=h → task = items[h])⟩	
	[assume H == h; H := h+1;]	
	8 ⟨... ∧ !tics⟩ return;	
	}	
1	pre: ⟨stTid(tid) ∧ ticsCond()⟩	
	post: ⟨stTid(tid) ∧ ticsCond()⟩	
	{	
	1 ⟨...⟩[N] h := H;	
	2 ⟨...⟩ [N]	
	[if(h<T)	
	assume h==H ==> H<T;	
	else	
	assume h>=y;]	pre: ⟨ptInv()⟩
	3 if(h >= t)	post: ⟨ptInv()⟩
	{	{
	4 [B] task := EMPTY;	⟨ptInv()⟩
	5 ⟨...⟩ return;	1 [R] t := T;
	}	2 [R] [assert t==T && !tics;
	6 ⟨...⟩[N]	items[t] := task;]
	[assert !tics && h==H ==> H<T;	3 [N] T := t+1;
	if(h==H)	4 ⟨ptInv()⟩ return;
	task := items[h];	}
	else	
	havoc task;]	
	7 ⟨...⟩[N]	
	[assume H == h; H := h+1;]	
	8 ⟨...⟩ return;	
	}	
0	pre: ⟨stTid(tid) ∧ ticsCond()⟩	
	post: ⟨stTid(tid) ∧ ticsCond()⟩	
	{	
	1 ⟨ticsCond()⟩ h := H;	
	2 ⟨H≥h ∧ ticsCond()⟩ t := T;	
	3 if(h >= t)	pre: ⟨ptInv()⟩
	{	post: ⟨ptInv()⟩
	4 ⟨ticsCond()⟩ task := EMPTY;	{
	5 return;	1 ⟨ptInv()⟩
	}	t := T;
	6 ⟨ H≥h ∧ ticsCond()∧ ticsCond2(h)⟩	2 ⟨ptInv()⟩
	task := items[h];	items[t] := task;
	7 ⟨H≥h ∧ ticsCond() ∧ ticsCond2(h)⟩	3 ⟨ptInv()⟩
	[assume H == h; H := h+1;]	T := t+1;
	8 ⟨ticsCond()⟩ return	4 ⟨ptInv()⟩ return;
	}	}
	ticsCond():	ptInv():
	(tics ⇒ H ≤ T+1) ∧ (!tics ⇒ H ≤ T)	tid = ptTid ∧ !tics ∧ H ≤ T
	ticsCond2(h:int):	
	(tics ∧ h=H ⇒ H ≤ T) ∧	
	(!tics ∧ h=H ⇒ H < T)	
	stTid(tid:Tid):	
	tid ≠ NULL ∧ tid ≠ ptTid	

Fig. 2. Proof layers for the mechanized proof of steal and put methods

Layer	`take(linear tid:Tid):(task:int)`		
3	⟨ptInv()⟩ [goto 1A, 1B, 1C; 1A: assume H==T; task := EMPTY; return; 1B: assume H==T−1; task := items[T−1]; H := H+1; return; 1C: assume H<T−1; T := T−1; task := items[T]; return;]		
	take1(lnr tid:Tid): (task:int)	take2(lnr tid:Tid): (task:int)	take3(lnr tid:Tid): (task:int)
2	⟨ptInv()⟩ [assume H==T; task := EMPTY;]	⟨ptInv()⟩ [assume H < T−1; T := T−1; task := items[T];]	⟨ptInv()⟩ [goto 1A,1B; 1A: assume H==T; task := EMPTY; return; 1B: assume H==T−1; task := items[T−1]; H := H+1; return;]
1	pre: ⟨ptInv()⟩ post: ⟨ptInv()⟩ { ⟨ptInv()⟩ 1 [R] t := T−1; 2 [R] [T := t; tics := true;] 3 [R] assume h <= H && t<h; 4 if(t<h) { 5 [L] [T :=h; tics := false;] 6 [B] task := EMPTY; 7 ⟨ptInv()⟩ return; } ... }	pre: ⟨ptInv()⟩ post: ⟨ptInv()⟩ { ⟨ptInv()⟩ 1 [R] t := T−1; 2 [R] [T := t; tics := true;] 3 [N] [h := H; assume h<t; tics := false;] 4 if(t<h) ... 8 [B] task := items[t]; 9 if(h<t) { 10 ⟨ptInv()⟩ return; } ... }	pre: ⟨ptInv()⟩ post: ⟨ptInv()⟩ { ⟨ptInv()⟩ 1 [R] t := T−1; 2 [R] [T := t; tics := true;] 3 [R] assume h==t && h<=H; 4 if(t<h) ... 8 [B] task := items[t]; 9 if(h<t) ... 11 [N] [if(h==H) H :=h+1; chk := true; else chk := false;] 12 if(!chk) { 13 [B] task := EMPTY; } 14 [L] [T := t+1; tics := false;] 15 ⟨ptInv()⟩ return; }
0	pre: ⟨ptInv()⟩ post: ⟨ptInv()⟩ { 1 ⟨ptInv()⟩ t := T−1; 2 ⟨ptInv() ∧ t=T-1 ⟩ [T := t; tics := true;] 3 ⟨t=T ∧ H≤T+1 ∧ tics⟩ [h := H; assume t<h;] 4 if(t<h) { 5 ⟨t=T ∧ h≤H ∧ H=T+1 ∧ tics⟩ [T :=h; tics := false;] 6 ⟨ H≤T ∧ !tics⟩ task := EMPTY; 7 ⟨ H≤T ∧ !tics⟩ return; } ... }	pre: ⟨ptInv()⟩ post: ⟨ptInv()⟩ { 1 ⟨ptInv()⟩ t := T−1; 2 ⟨ptInv()∧ t=T-1⟩ [T := t; tics := true;] 3 ⟨t=T ∧ H≤T+1 ∧ tics⟩ [h := H; assume h<t; tics := false;] 4 if(t<h) ... 8 ⟨t=T ∧ H≤T ∧ h<t ∧ !tics⟩ task := items[t]; 9 if(h<t) { 10 ⟨ H≤T ∧ !tics⟩ return; } ... }	pre: ⟨ptInv()⟩ post: ⟨ptInv()⟩ { 1 ⟨ptInv()⟩ t := T−1; 2 ⟨ptInv()∧ t=T-1⟩ [T := t; tics := true;] 3 ⟨t=T ∧ H≤T+1 ∧ tics⟩ [h := H; assume h==t;] 4 if(t<h) ... 8 ⟨t=T ∧ h≤H ∧ H≤T+1 ∧ h=t ∧ tics⟩ task := items[t]; 9 if(h<t) ... 11 ⟨t=T ∧ h≤H ∧ H≤T+1 ∧ h=t ∧ tics⟩ [if(h==H) H :=h+1; chk := true; else chk := false;] 12 if(!chk) { 13 ⟨t=T ∧ H=T+1 ∧ tics⟩ task := EMPTY; } 14 ⟨t=T ∧ H=T+1 ∧ tics⟩ [T := t+1; tics := false;] 15 ⟨ptInv()⟩ return; }

Fig. 3. Proof layers for the mechanized proof of take methods

not changed since the previous layer, we denote this as $\langle ... \rangle$ or if it is tightened by adding a new constraint ϕ, we denote this as $\langle ... \wedge \phi \rangle$.

Mover annotations of the atomic actions are also present in brackets before atomic actions. R, L, B and N denote right-, left-, both- and none-mover, respectively. We may omit the mover tag if the atomic action is labeled as non-mover. Note that we may have labeled an action non-mover although it is a mover, if it is not necessary for our proof.

Atomic actions or statements may contain constructs like `assume`, `assert` and `havoc` which are special for verification. Semantics of `havoc` is explained via an example in Section 3. The statements `assume e` or `assert e` cause an execution to block (all threads' next statement to execute becomes not enabled) or fail, respectively, if the boolean expression `e` evaluates to `false` in the state just before executing this statement. Otherwise, they are equivalent to `skip`.

For the `take1`, `take2` and `take3` methods some paths are unreachable due to `assume` statements at Line 3. We omit the program text for the if blocks leading to those paths in Figure 3 by representing them with three dots as in Line 4 of `take2` at Layer 0.

The programs contain a new boolean ghost variable named `tics`. A ghost variable is similar to a regular variable with only difference that it does not modify the program state i.e., its value is never assigned to a real program variable. Its sole purpose is to guide CIVL during mover and OG checks.

The name `tics` is short for "take in critical section". We know from Observations 3-7 in Appendix A that $H \leq T$ can be temporarily violated inside the `take` method. We say that `take` is in critical section if execution of `take` is in the area that $H \leq T$ invariant can be violated. The `tics` is used to write location annotations considering the current instruction of `ptTid`.

The proof consists of 4 layers. At the bottom layer (Layer 0), we start with the method bodies that we obtained at the end of Appendix B. We decorate the method bodies with location annotations to establish relation between H and T global variables. While going from Layer 0 to Layer 1, we abstract some of the actions of Layer 0 and we use reduction at Layer 1 to make bodies of the `put`, `take1`, `take2` and `take3` methods single step. Between Layer 1 and Layer 2, we use method abstraction on `put`, `take1`, `take2` and `take3` methods and abstract them to single step atomic actions. In Layer 2, $H \leq T$ begins to hold as a global location annotation since `take1`, `take2` and `take3` methods become coarse enough. Finally, we apply method abstraction on `take` and `steal` methods between Layer 2 and Layer 3 to obtain desired atomic actions for these methods. In Layer 3, all the methods of the WSQ algorithm are in the form of atomic actions.

Layer 0 We start with the program obtained after loop-peeling and path-splitting explained in Appendix B. Only difference is the addition of the boolean `tics` ghost variable. Taking Observations 3-7 into account, we set `tics` to *true* temporarily inside the bodies of `take` methods and set it back to $false$ at the point where we think the $H \leq T$ condition is restored.

We provide location annotations conforming the value of `tics`. In the methods, we annotate the locations so that when `tics` is *true*, $H \leq T+1$ holds and when `tics` is $false$, $H \leq T$ holds. This condition is expressed in the sub-formulae called `ticsCond` provided after `steal` method. The condition when the `tics` is *true*, is adjusted in precision. It is tight enough to continue proof in later layers and relaxed enough to be satisfied after non-interference checks. For instance, replacing it with $H == T + 1$ would be too tight and make it unsatisfiable.

$H \leq T$ is not a global invariant at Layer 0, but a relaxed version of it that we call `ticsCond` is a global invariant. All the location annotations in the method bodies and method pre- post-conditions imply `ticsCond`.

However, location annotations explained so far does not pass the OG check directly. We need to make location annotations stronger.

First, if $H = T \wedge !tics$ or $H = T{+}1{\wedge}tics$ holds at some state during execution of a `ptTid` method, a successful `CAS` operation of a `steal` could interfere and violate location annotation by incrementing H. We observe that this corner case is not possible in real executions. $H < T$ must hold if $!tics$ or $H < T + 1$ must hold if `tics` just before the execution of `CAS` action of `steal`. For this reason, we introduce `ticsCond2` function that reflects our condition and add it to the annotations of Lines 6 and 7 of `steal`.

Adding `ticsCond2` to location annotations of Lines 6 and 7 is still insufficient because two concurrent steals may violate the `ticsCond2`. If $h = H{-}1{\wedge}H = T{-}1{\wedge}!tics$ holds just before a stealer thread t_1 executes Line 6 and another stealer thread t_2 interferes at this point and performs a successful `CAS` and increments H, we come up with a state satisfying $h = H \wedge H = T{\wedge}!tics$ for t_2 which violates the `ticsCond2`. But we know that two successful `steal`s cannot be concurrent by Observation 2. This observation helps us to infer that $H \geq h$ holds during Lines 6 and 7 of the `steal` method (by Observation 1) which prevents the previous erroneous execution sample. By adding $H \geq h$ on Lines 6 and 7 of `steal`, we make sure that OG checks for location annotations of `steal` pass.

Second, methods of the WSQ modify global variables H and T by assigning them local values of h and t. To show that these assignments do not violate the conditions relating H and T, we need to relate local value t to value of T in `ptTid` methods and h to H in `steal`, `take1`, `take2` and `take3` methods. Since T is only modified by `ptTid`, adding $t = T$ to location annotations of `ptTid` methods is correct. By Observation 1, H is always non-decreasing. Hence $H \geq h$ holds for all methods. Adding these two conditions to the location annotations of certain lines is sufficient to show that modifications on global variables does not violate the required conditions.

We omit the mover tags for this layer, since no reduction is performed at Layer 0.

Layer 0 → Layer 1 Our aim for Layer 1 is to grow steps of `take1`, `take2`, `take3` and `put` method bodies using reduction. For this reason, we abstract some of the atomic actions between Layer 0 and Layer 1. Lines 2 and 6 of `steal` method and Line 3 of `take1` and `take3` methods are abstracted for this purpose. Rationale behind these abstractions are explained in Layer 1 while explaining how the actions satisfy their mover annotations.

Layer 1 We assign mover tags to atomic actions of `put`, `take1`, `take2` and `take3` methods so that we can grow the steps of the action blocks of these methods as large as the ones we need.

First, let us explain how we grow the step of `put` method. Line 1 becomes right-mover without any abstraction since it reads the global variable T which is not modified by other threads. However, Line 2 of `put` method is not a right-mover without abstraction since it may be modifying an index of $items$ that is read by Line 6 of `steal`. If h value of `steal` at Line 6 is equal to t value of `put` at Line 3, they may be accessing the same index. Since action of `put` method modifies this index, mover check fails. But, we observe that the actual value of $items[h]$ is not needed if $h \neq H$ holds just before execution of Line 7 of `steal` method. Moreover, we need to read the actual value of $items[h]$ if $H = h$ at Line 7 in order to know that steal returns the correct element. Hence, we abstracted Line 6 of `steal` such that it assigns $items[h]$ to $task$ if $h = H$ at Line 6 and assigns a non-deterministic value to $task$ otherwise.

Next, we enlarge the step of `take1` method. Line 1 is a right-mover since it is same as Line 1 of `put` method and Line 5 is a both-mover since it is a local assignment. However, lines 2 and 6 are not movers in Layer 0. They do not commute with Line 2 of `steal` method since Line 2 of `steal` reads the global variable T which is modified by Line 2 (or Line 6) of `take1` method. To overcome this problem, let us explore what expect from Line 2 of `steal` method in our proof. If `steal` observes $h < t$ after execution of Line 2 and the value of H has not changed yet, then `ticsCond2` must hold if it continues execution through Line 6. We may satisfy this condition by assuming only $H < T$ holds after Line 2 if $h = H \wedge h < T$ holds just before Line 2. `steal` may continue through the if block at Line 3 if it observes $h < t$ before Line 2, since our top-level implementation of `steal` allows it to return `EMPTY` even if $H < T$. But, if `steal` observes $h \geq T$ before Line 2, it must enter the if block at Line 3 since the WSQ is empty. Obtaining only this information after Line 2 would be sufficient for abstracting `steal` on later layers.

Line 3 of `take1` is not a right-mover at Layer 0 since it does not commute with Line 7 of `steal` method. Instead of reading actual value of H at Line 3 of `take1`, we abstract to read a non-deterministic value less than or equal to H. This abstract read can commute right of Line 7 of `steal` since after moving right of Line 7 of `steal` local value h of `take1` can have more distinct values and it is tight enough to infer that $h >= t$. Consequently, we obtain a step for `take1` that spans lines through 1 to 6.

Lines 1 and 2 of `take2` are right-movers due to reasons explained above. Line 8 becomes a right-mover since `steal` does not modify the $items$ array. Consequently, lines from 1 to 8 of `take2` form a step in Layer 1.

Mover annotations for `take3` method also hold and lines from 1 to 14 become a step. Reason for their correctness can be explained with the same arguments above.

Note that location annotations of Layer 1 are same as Layer 0. Abstractions do not violate the conditions established at Layer 0.

Layer 1 → Layer 2 We apply method abstraction on `put`, `take1`, `take2` and `take3` methods between Layer 1 and Layer 2 since action blocks of those methods grow large enough at the end of Layer 1. We obtain the desired atomic action for `put` method.

Layer 2 In this layer, we tighten the location annotations in `steal` method such that they become the old condition and $!tics$. The OG checks pass for these new tighter annotations because no step of the program leaves the `tics` *true* after its execution at Layer 2.

In addition, we add a condition to location annotation of Line 7 of `steal` stating that if the value of H had not changed since `steal` read it, the return variable $task$ contains $items[H]$. OG check for this condition passes since Line 6 of `steal` is tight enough to assign correct value to $task$ if $H = h$. This extra condition on the location invariant is crucial when we apply method abstraction on `steal`.

Layer 2 → Layer 3 Between Layers 2 and 3, we apply method abstraction on `steal` and `take` methods. These methods become atomic actions.

The reason we can not apply method abstraction on `steal` so far is that location annotations of the action blocks of `steal` were not tight enough to obtain the desired atomic action for `steal`. It was possible to perform a successful `CAS` when $H = T$.

Applying method abstraction on `take` also becomes possible after Layer 2 since we obtained the atomic actions for its sub-methods at Layer 2.

Layer 3 All of the `take`, `put` and `steal` methods are atomic actions.

With these single atomic action bodies of the methods, it is easier to reason about the WSQ algorithm. For instance, one can show that a task pushed into deque cannot be taken or stolen more than once since `take` and `steal` methods' top-level actions atomically increment H or decrement T after taking the first or the last item from the queue.

5 Related Work

Due to its key importance in parallel sytems, there are various WSQ algorithms. A notable one is presented in Cilk multi-threaded language [4]. This algorithm is blocking and method bodies are protected by a global lock. Reasoning about correctness of Cilk WSQ algorithm is simpler but it is not efficient due to its blocking nature.

Another WSQ algorithm introduced by Arora et al [1] is non-blocking, but it requires fixed size queues. This algorithm has been verified in [6] using a model checking approach. Model checking approach validates that the algorithm satisfies desired properties but it does not provide any insight about the behavior of the algorithm. Hence, it is difficult to reason about some side-properties and possible optimizations of the algorithm using this approach. The Chase-Lev WSQ [2] we have studied is an improvement over [1] such that size of the queue can grow without memory leaks.

Since work stealing queues are used in low-level task schedulers,the environment may provide weaker guarantees. It is known that executions of Chase-Lev WSQ under TSO semantics show more behaviour than SC executions [9]. If memory fences are inserted after Line 3 of `put` and Line 2 of `take`,non-SC behaviors are prevented [9]. In [7], a pen and pencil proof has been presented that the Chase-Lev WSQ algorithm with previously mentioned memory fences satisfy some desired specifications. A modified version of the Chase-Lev WSQ is presented in [10] such that it is correct under TSO memory-model if we know the size of store buffers.

6 Conclusions and Future Work

In this study, we have performed a linearizability proof of the Chase-Lev WSQ algorithm under SC semantics using proof tool CIVL. Lower layers of the proof provide insight for the behavior of the SC executions and the top layer single atomic block summaries of the methods are simple but tight enough to show the desired properties.

We plan to extend this work to investigate behavior of the WSQ algorithm under weak memory models like TSO by modeling the weak memory semantics explicitly in CIVL. We are particularly interested in the behavior of the executions and the properties satisfied in the absence of memory fences.

References

1. Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.
2. David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.
3. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *ACM Symposium on Principles of Programming Languages*, page 14. Association for Computing Machinery, Inc., January 2009.
4. Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.
5. Chris Hawblitzel, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. *Computer Aided Verification*, 2015.
6. Majid Khorsandi Aghai. Verification of work-stealing deque implementation. 2012.
7. Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Notices*, volume 48, pages 69–80. ACM, 2013.

8. Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
9. Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *ACM SIGPLAN Notices*, volume 47, pages 429–440. ACM, 2012.
10. Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded tso processors. *ACM SIGPLAN Notices*, 49(4):413–426, 2014.
11. Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

A Observations on the SC Executions of the Program

In this section, we first present some observations on (full or partial) executions of the WSQ algorithm with the finest-grained actions (the algorithm in Figure 1). They will be helpful for obtaining location annotations and enlarging atomic blocks in upper layers.

Our initial observations are simple and they hold for full executions.

Observation 1: H is non-decreasing throughout an execution.

Observation 2: Let us call a `steal` operation successful if it returns a value other than `EMPTY`. Then, last iterations of two successful `steal`s cannot be concurrent.

Next, we want to understand the relation between global variables H and T. For a sequential execution, one expects that $H \leq T$ invariant holds throughout the execution. This is not true for the fine-grained SC execution. We observe that H could exceed T in some special cases. However, this violations occur temporarily if `take` method follows some paths and they begin to hold again after `take` method finishes.

We examine execution portions (sub-sequences of executions) in a systematic way to obtain observations showing relation between H and T variables.

Observation 3: If an execution portion consists of only concurrent `steal` operations, we observe that $H \leq T$ is preserved throughout the execution portion.

Observation 4: If an execution portion consists of a single `put` method concurrent with `steal` methods, then $H \leq T$ holds throughout this execution portion.

Next, we consider execution portions that has `take` method concurrent with `steal` operations. The `take` method could follow three different paths by either entering the if block in Line 4 (path 1), by entering the if block in Line 9 (path 2) or by not entering those if blocks and returning by Line 15 (path 3).

Observation 5: If an execution portion consists of path 1 of take method concurrent with steals, then $H \leq T$ holds throughout this execution portion.

Observation 6: If an execution portion consists of path 2 of take method concurrent with steals, then $H \leq T$ holds before Line 2 and after Line 14 of the `take` method and $H = T \vee H = T + 1$ holds between Lines 2 and 14 of `take` method.

Observation 7: If an execution portion consists of path 3 of take method concurrent with steals, then $H \leq T$ holds before Line 2 and after Line 5 of the `take` method and $H = T + 1$ holds between Lines 2 and 5 of `take` method.

B Path Splitting and Loop Peeling

In this section we present our initial abstractions on `steal` and `take` methods. These abstractions are not performed by CIVL. Rather, they are obtained by applying some proof rules that are not currently supported by CIVL. We explain the rules and their applications in this section. Methods we obtained after initial abstractions constitute the bottom layer of our mechanized proof.

```
   steal(lnr tid:Tid):
        (task:int)
   {
    var h,t:int;
    var chk:bool;

1   while(*)
    {
2    h := H;
3    t := T;
4    assume h<t;
5    task := items[h];
6    [assume h != H;
      chk := false;]
    }
7   h := H;
8   t := T;
9   if(h>=t)
    {
10   task := EMPTY;
11   return;
    }
12  task := items[H];
13  [assume h==H;
     H == h;]
14  return;
   }
```

```
   steal(lnr tid:Tid):
        (task:int)
   {
    var h,t:int;
    var chk:bool;

1   while(*)
    {
2    havoc h,t,chk,task;
    }
3   h := H;
4   t := T;
5   if(h>=t)
    {
6    task := EMPTY;
7    return;
    }
8   task := items[H];
9   [assume h==H;
     H == h;]
10  return;
   }
```

```
  steal(lnr tid:Tid):
       (task:int)
  {
   var h,t:int;
   var chk:bool;

1  h := H;
2  t := T;
3  if(h>=t)
   {
4   task := EMPTY;
5   return;
   }
6  task := items[H];
7  [assume h==H;
    H == h;]
8  return;
  }
```

Fig. 4. Initial abstractions on `steal` method

Our first abstraction is performed on `steal` method. If we consider the iterations of the loop at Line 1 of `steal` before the last iteration, they do not modify any global variable and value assigned to return variable is reset by the last iteration. Moreover, value assigned to local variables by reading global variables are also reread by the last iteration before using them. Hence, those iterations has no important effect on OG annotations of other methods and they will not be useful for the refinement proof of `steal`. Our aim is to abstract `steal` method so that we do not need to deal with unsuccessful previous iterations of `steal`.

On the left-side of Figure 4, we have `steal` method obtained by peeling out the last iteration of the `while` loop. All the unsuccessful iterations are captured by the loop at Line 1. They are guaranteed to be unsuccessful by `assume` statements at lines 4 and 6. The last successful iteration is modeled from Line 7 on.

Lines 2, 3, 5 and 6 could be abstracted by *havoc* $h, t, task, chk$ and line 4 can be abstracted by *skip*. With these abstractions, we obtain the method body in the middle column of Figure 4. Since h, t, chk and $task$ variables have non-deterministic values at the beginning of `steal`, the whole loop at Line 1 could be removed and we obtain the method at the right-side of Figure 4 as our basis of `steal` for the mechanized proof.

```
 take(lnr tid:Tid):
     (task:int)
 {
  var h,t:int;
  var chk:bool;

1 if(*)
  {
2   task := take1(tid);
  }
  else
  {
3  if(*)
   {
4   task := take2(tid);
   }
   else
   {
5   task := take3(tid);
   }
  }
6 return;
 }
```

```
   take1(lnr tid:Tid):
        (task:int)
   {
    var h,t:int;
    var chk:bool;

1   t := T-1;
2   T := t;
3   [h := H;
     assume t<h;]
4   if(t<h)
    {
5    T := h;
6    task := EMPTY;
7    return;
    }
8   task := items[t];
9   if(h<t)
    {
10   return;
    }
11  [if(h==H)
    {
     H := h+1;
     chk := true;
    }
    else
    {
     chk := false;
    }]
12  if(!chk)
    {
13   task := EMPTY;
    }
14 T := t+1;
15 return;
   }
```

```
   take(lnr tid:Tid):
        (task:int)
   {
    var h,t:int;
    var chk:bool;

1   t := T-1;
2   T := t;
3   [h := H;
     assume t>=h;
     assume t != h;]
4   if(t<h)
    {
5    T := h;
6    task := EMPTY;
7    return;
    }
8   task := items[t];
9   if(h<t)
    {
10   return;
    }
11  [if(h==H)
    {
     H := h+1;
     chk := true;
    }
    else
    {
     chk := false;
    }]
12  if(!chk)
    {
13   task := EMPTY;
    }
14 T := t+1;
15 return;
   }
```

```
   take(lnr tid:Tid):
        (task:int)
   {
    var h,t:int;
    var chk:bool;

1   t := T-1;
2   T := t;
3   [h := H;
     assume t>=h;
     assume t==h;]
4   if(t<h)
    {
5    T := h;
6    task := EMPTY;
7    return;
    }
8   task := items[t];
9   if(h<t)
    {
10   return;
    }
11  [if(h==H)
    {
     H := h+1;
     chk := true;
    }
    else
    {
     chk := false;
    }]
12  if(!chk)
    {
13   task := EMPTY;
    }
14 T := t+1;
15 return;
   }
```

Fig. 5. Initial abstractions on `take` method

For the `take` method, we want to separate `take` in such a way that we can reason about each possible path separately. We use the following rule for this purpose:

Rule 1: Let procedure foo has the following body:

```
{ s0; s1; s2;}
```

where `s0` and `s2` are sequence of statements and `s1` is an atomic block. Then, replacing this body with the following one is a valid abstraction of foo:

```
{ if(*){ s0; [s1;assume p;] s2; }
  else { s0;[s1;assume !p;] s2;} }
```

where `p` is a boolean expression on local variables. The $*$ denotes a non-deterministic value of `true` or `false` To obtain the desired method body in Figure 5, we apply the following steps:

1. Apply Rule 1 to `take` in Figure 1 with taking $s0$, $s1$, $s2$ as lines 1,2; 3 and 4-15 respectively. We also pick p as $h > t$.
2. Collect statements inside the if block in the method `take1` and statements in the else part in the method `take23`.
3. Apply Rule 1 to `take23` with the same line choices in step 1 but taking p as $h = t$.
4. Collect the statements inside the if block of `take23` in `take2` method and statements in the else part in `take3` method.
5. Inline call of `take23` inside the else block of `take` with its body.