Verifying a Very Racy Stack

Matt Windsor and Mike Dodds

University of York
{mbw500, mike.dodds}@york.ac.uk

Abstract. We discuss a work-in-progress on verifying the *SP pool*, a concurrent stack, using separation logic. We report on a first draft of a proof of several properties we then use to prove memory safety, moving towards functional correctness. We comment on the advantages and disadvantages of our chosen logic, *iCAP*, and lessons we can take away for applications of separation logic to low-level concurrent software.

1 Introduction

Concurrent programming is more necessary now than ever before. Modern CPU architectures, instead of building up with higher clock speeds and longer pipelines, are growing out with multi-core designs. To use every core effectively, we must spread our computations across them, moving from one sequential process to multiple concurrent threads that may operate in parallel.

For concurrent programs, our choice of data structures is key. Concurrent structures must tolerate interference: as multiple threads can access the structure at one time, we must ensure that no action is interrupted by other actions in a way that disrupts its own work. We want them to be fast, but optimisations tend to raise the possibility of unwanted interference. Data-races, the simultaneous writing and reading (or multiple-writing) of a shared variable, are a concern.

Proving these structures correct is hard—correctness for concurrent programs is more complex than in the sequential world. There are many more issues to worry about: race conditions, liveness, and the existence of more than one valid scheduling in most cases. Various correctness conditions define concurrent correctness: a common one is linearisability [4], the idea that, given a sequential specification of an algorithm, every operation takes effect atomically and in a schedule consistent with the program orders of each sequential method.

We examine the *SP pool*, a single-producer multi-consumer stack. Part of the multi-producer *time-stamped stack* of Dodds *et al.* [3], it uses low-level optimisations which make reasoning challenging. Building up to proving it correct, we discuss our approach to proving well-formedness conditions on the pool, using the separation logic *iCAP* [7]. Such logics extend Hoare logic to deal with disjoint resources, such as areas of memory, that can be separated and combined. With these, we can work on part of a large resource while preserving its context.

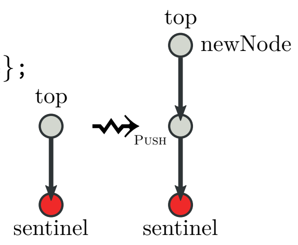
Paper structure. In Section 2, we outline the SP pool. We then discuss what it means for the pool to be well-formed (but not necessarily correct). We outline our proof method in Section 4, and evaluate it in Section 5. We end with an overview of what we have learned, and proposed future work.

2 Outline of the SP Pool

The pool consists of several methods, each acting on some top node `top`. We outline each in pseudo-code, with a diagram of its key atomic action; circles represent nodes of interest (shaded red if taken, or 'dead' to the algorithm). Here, and later on, we use **boldface** for method names, SMALL CAPS for atomic action names, `typewriter` for code, and *`italic typewriter`* for atomic code.

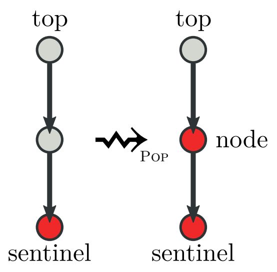
We create pools with **init**. New pools have one node, the sentinel, which marks the end of the pool. The sentinel is taken, has no element, and points to itself. We do not discuss **init** further: we assume it occurs outside shared state, and yields well-formed pools. The **push**[1] method creates a new node, points it to the current `top`, then atomically makes said new node `top`.

```
Node push(Element el) {
   Node newNode = {element: el, taken: false};
   newNode.next = top;
   top = newNode; // atomic action 'PUSH'
   fwdComp(newNode); // defined later
   return newNode;
}
```



The **pop**[2] method finds the newest non-taken node, if one exists. It does so via a simple non-atomic while loop from `top`. It then tries to mark that node as taken, through atomic compare-and-swap[3]. If it fails, another consumer took the node first, and the caller should **pop** again. Else, it returns the taken node's element.

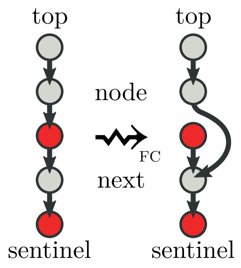
```
<Element, Status> pop() {
   Node oldTop = top;
   Node node = oldTop;
   while (node.taken) {
     if (node.next == node)
       return <NULL, EMPTY>; // Sentinel
     node = node.next;
   }
   if (CAS(node.taken, false, true)) { // atomic action 'POP'
     backComp(oldTop, node); // defined later
     fwdComp(node);
     return <node.element, SUCCESS>;
   }
   return <NULL, CONTENDED>;
}
```



[1] **insert** in [3].
[2] **getYoungest** and **remove** in [3].
[3] For our needs, this is *`CAS(var, old, new)`*, which sets `var = new` if `var == old`.

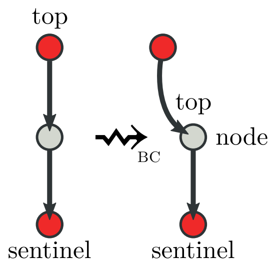
If we ignore **fwdComp** and **backComp**, our code implements all of the functionality of the pool. In practice, this is not sufficient: **push** adds new nodes to the pool, yet **pop** does not dispose of nodes: it marks them taken. This causes time and space leaks: taken nodes remain in memory, and must be traversed, yet are 'dead' to the algorithm. We must remove taken nodes from the chain of nodes between top and sentinel—the spine—, so that garbage collection (or similar) can dispose of them. To do so, we first define **fwdComp**:

```
void fwdComp(Node node) {
   next = node.next;
   while (next.next != next && next.taken)
     next = next.next;
   node.next = next; // atomic action 'FC'
}
```



We call this forwards compression: we sweep from a target node downwards, non-atomically, to find the next non-taken node. Then, we atomically re-point the target's `next` to that non-taken node. We now define backwards compression:

```
void backComp(Node oldTop, Node node) {
   CAS(top, oldTop, node); // atomic 'BC'
   if (oldTop != node)
     oldTop.next = node; // atomic 'FC'
}
```



From earlier, we know that everything between `oldTop` and `node` is taken. We can use this in two ways. Firstly, if `oldTop` is still `top`, we can move the latter to point to `node`: this is the `CAS` shown above. Secondly, if doing so does not create a cycle, we can perform atomic action FC between `oldTop` and `top`.

Unusual observations on the SP pool. The pool, though simple in implementation and intuitively a perfectly reasonable stack, violates many common-sense assumptions about correct stacks. For example, the top node is not always the most recently inserted node (and nodes are not totally ordered by insertion); the pool is not a linked list; and nodes compressed out of the spine can still be traversed and even re-appear later. All of these surprising facts arise from compression: it races with itself and other methods, and alters the pool shape in ways unusual for a stack. However, in terms of adhering to the specification of a concurrent stack, compression has no visible effect on the pool.

As both forwards compression and popping use non-atomic traversals through next-pointers, they can observe a node at the moment a forwards compression moves it out of the spine. This is why we cannot assume the pool is a linked list; while the spine forms one, and all non-taken nodes are within it, there may exist several active branches where a traversal is still sweeping a series of already-compressed nodes. Our SP pool, then, is a reversed tree, with all branches eventually converging onto the spine.

Non-atomic traversals also pose problems. In figure 1, we have a spine containing some taken nodes, and non-taken nodes `A` and `B` in that order.

Step	Thread 1	Thread 2	Thread 3
1	**push** inserts `node`		
2	**push** begins FC on `node`		
3	**fc** loop stops with `next=A`		
4		**pop** marks `A` taken	
5			**pop** marks `node` taken
6			**pop** performs BC on `node`
7			**pop** begins FC on `node`
8			**fc** loop stops with `next=B`
9			**fc** changes `node.next=B`
10	**fc** changes `node.next=A`		

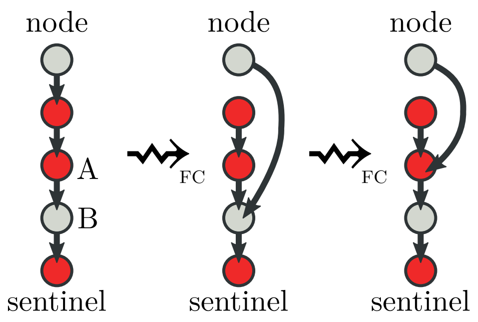


Fig. 1. Sample schedule for data-race between two forwards compressions, and diagram of steps 9 and 10.

Thread 1 replaces Thread 3's compression `node`→`B` (step 9) with the smaller `node`→`A` (step 10), undoing part of 3's work. This is harmless: the only difference is the number of taken nodes on the spine, and this cannot affect behaviour.

A similar effect occurs between backwards compression and insertion. When the new node is created in local state, it takes the current top pointer as `next`. While no other insert can change the top pointer in the meantime, backwards compression can. This is also harmless, as it can only moves a top pointer down past taken nodes, which will be re-introduced by the insertion.

3 Well-Formedness of the SP Pool

We intend to prove well-formedness properties for the pool. To do so, we now examine what we mean by 'well-formedness', first in informal overview and later in a more precise mathematical sense.

Orders on the SP pool. Our concept of well-formedness is heavily based on the observation and constraint of various orders between pool nodes. Before we define well-formedness, we first give each an informal definition:

Insertion predecessor. The relation of node pairs (a, b) where a was inserted directly after b;

Insertion order. The reflexive transitive closure of the above ($a > b$ if a was inserted any time after b), and is total due to the single-producer nature of the pool;

Physical predecessor. The relation of node pairs (a, b) where a`.next` is b;

Physical order. The reflexive transitive closure of the above ($a > b$ if a reaches b by one or more next pointers); this is the reverse of the intuitive order, but is defined as such to make relating it with insertion order easier.

We now define well-formedness as a collection of invariants, mostly on the above orders, each preventing various possible pool failures. This resembles the invariants given by O'Hearn *et al.* [6] for a similar data structure, and several of these invariants are necessary for proving that the methods behave correctly.

Order-subset. Physical order subsets insertion order; actions cannot cause cycles or non-'last-in-first-out' ordering, but physical order may be partial;

Single-predecessor. Each node has exactly one physical predecessor;

Sentinel-predecessor. Only the sentinel can point to itself, preventing cycles;

Sentinel-reach. Each node reaches the sentinel through a path of `next`-pointers; all nodes eventually have a common successor and form the same pool;

Spine. The spine is the only path on which non-taken nodes can lie; compression cannot sweep non-taken nodes off it;

Taken–non-taken disjoint. No node can be both taken and non-taken;

Element-injection. Every non-taken node has exactly one element, and every element is mapped to at most one node.

Stability. As the pool is concurrent, consumers can be scheduled in parallel with each other, and with the single producer. Thus, it is constantly changing under our atomic actions. This poses a threat to well-formedness: we have to show that once we've established it, no atomic action at any point can violate it. This idea of stability is a key component of our proof.

State tuples. In our proof, a pool state is captured by the 8-tuple

$$\begin{array}{llr}
\langle N & \subseteq \mathit{Node}, & \text{(non-taken nodes)} \\
T & \subseteq \mathit{Node}, & \text{(taken nodes)} \\
t & \in \mathit{Node}, & \text{(top node)} \\
s & \in \mathit{Node}, & \text{(sentinel node)} \\
\mathcal{P} & \subseteq \mathit{Node} \times \mathit{Node}, & \text{(physical predecessors)} \\
\mathcal{T} & \subseteq \mathit{Node} \times \mathit{Node}, & \text{(insertion predecessors)} \\
E\rangle & \subseteq \mathit{Node} \times \mathit{Element} & \text{(element map)}
\end{array}$$

We achieve the physical order, which we call $\geq_{\mathcal{P}}$, by taking the reflexive transitive closure of $\mathcal{P}$; insertion order $\geq_{\mathcal{T}}$ is achieved similarly for $\mathcal{T}$.

Abstract atomic actions. Each state-modifying action in the pool is atomic: any other thread either sees a state before the action, or a state afterwards. We can characterise all changes to the pool state in terms of these actions.

The pool has four atomic actions—PUSH, POP, FC, and BC—, which correspond to the italicised statements in our pseudo-code. We begin by giving sketches of these in the form of functions on state tuples. These sketches are initially too weak, as they permit ill-formed pools, but we later constrain them. First, we define the atomic actions of pushing and popping some node n:

$$\begin{aligned}\textsc{Push}\ n &\approx \langle N, T, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle \\ &\quad\rightsquigarrow \langle N \uplus \{n\}, T, n, s, \mathcal{P} \uplus \{(n, m)\}, \mathcal{T} \uplus \{(n, i)\}, E \uplus \{(n, e)\}, n\rangle \\ \textsc{Pop}\ n &\approx \langle N \uplus \{n\}, T, t, s, \mathcal{P}, \mathcal{T}, E \uplus \{(n, x)\}, i\rangle \rightsquigarrow \langle N, T \uplus \{n\}, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle\end{aligned}$$

Forwards-compressing a node a to replace its next-pointer b with c, if all nodes moved between branches as a result are taken, corresponds to the atomic action

$$\text{FC}\ a\ b\ c \approx \langle N, T, t, s, \mathcal{P} \uplus \{(a, b)\}, \mathcal{T}, E, i\rangle \rightsquigarrow \langle N, T, t, s, \mathcal{P} \uplus \{(a, c)\}, \mathcal{T}, E, i\rangle$$

It should now be clear why $\mathcal{P}$ and $\mathcal{T}$ are distinct: FC 'abbreviates' the former to remove unnecessary paths through taken nodes. We must show that all edits to $\mathcal{P}$ are consistent with $\mathcal{T}$, only differing by unlinking taken nodes.

We saw in section 2 that **backComp** has two atomic actions. The second is an application of FC, as seen from its signature above. The first, if the new top value is c and everything inserted between t and c is taken, is

$$\text{BC}\ c \approx \langle N, T, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle \rightsquigarrow \langle N, T, c, s, \mathcal{P}, \mathcal{T}, E, i\rangle$$

This is why t and i are distinct: BC can change the top to something other than the most recently inserted node. (However, i and t must still be connected by a run of taken nodes). We keep a record of i so that, when we insert and update $\mathcal{T}$ with the new node, we can attach it to i and preserve the totality of $\geq_{\mathcal{P}}$.

Well-formedness predicate. We now know enough to give our informal well-formedness predicates a mathematical definition. We do this in figure 2.

4 Outline of the Proof

Our chosen proof technique, *iCAP*, is a separation logic. This means that it has a concept of separating conjunction (the $*$ operator) which allows two disjoint resources to be combined into one (and a resource to be split into two disjoint resources). This gives us modular reasoning, in that a proof over one resource is still valid when the resource is combined with any sufficiently disjoint context.

At the concrete level, as per the original definition of separation logic, these resources are areas of memory, which may be split and combined during reasoning. However, *iCAP* only permits this form of reasoning during atomic actions, as these are the only times where we are allowed to modify state. Otherwise, we interact with the state via a region predicate, which gives us a view of the shared memory region as a labelled transition system. In our proof, this predicate is

$$\mathit{region}(\mathcal{S}, T_x, I_x(\texttt{top}), r)$$

If S_x is the x member of the tuple S,

$$
\begin{aligned}
wf_{os}(S) &\triangleq \geq_{S_\mathcal{P}} \subseteq \geq_{S_\mathcal{T}} && \text{(Order-subset)}\\
wf_{1p}(S) &\triangleq \forall x \in S_N \cup S_T \cdot |S_\mathcal{P}[\{x\}]| = 1 && \text{(Single-predecessor)}\\
wf_{sp}(S) &\triangleq \forall x \in S_N \cup S_T \cdot (x, S_s) \in S_\mathcal{P} \iff x = S_s && \text{(Sentinel-predecessor)}\\
wf_{sr}(S) &\triangleq \forall a \in S_N \cup S_T \cdot a \geq_{S_\mathcal{P}} S_s && \text{(Sentinel-reach)}\\
wf_{s}(S) &\triangleq \forall a \in S_N \cup S_T \cdot a \in S_N \implies S_t \geq_{S_\mathcal{P}} a && \text{(Spine)}\\
wf_{TN}(S) &\triangleq S_T \cap S_N = \emptyset && \text{(Taken–non-taken disjoint)}\\
wf_{E}(S) &\triangleq (\forall n \in S_N \cdot\ |S_E(n)| = 1) \land (\forall e \in \mathit{Elem} \cdot |\{n | (n, e) \in S_E\}| \leq 1) && \\
& && \text{(Element-injection)}\\
wf(S) &\triangleq wf_{os}(S) \land wf_{1p}(S) \land wf_{sp}(S) \land wf_{sr}(S) \land wf_{s}(S) \land wf_{TN}(S) \land wf_{E}(S)
\end{aligned}
$$

Fig. 2. Formal definitions of the well-formedness predicates.

where $\mathcal{S}$ is the set of abstract states the pool can currently take, T_x a transition relation characterising the pool's atomic actions, I_x a interpretation function mapping members of $\mathcal{S}$ to concrete heaps (here parameterised by the address of the pool top), and r a label identifying the specific shared region in question.

Filling the region predicate. Our proof method, then, first involves defining the possible values of $\mathcal{S}$, T_x, and I_x. For $\mathcal{S}$, we lift our 8-tuple representation to sets; then, in region predicates, we can encode facts about the pool. For example,

$$\mathit{pooled}(r, \texttt{top}, \texttt{n}) \triangleq \mathit{region}(\{S \mid \texttt{n} \in S_N \cup S_T\}, T_x, I_x(\texttt{top}), r)$$

has that `n` is in pool r. Predicates on the same region label are $*$-joinable resources, with the resulting $\mathcal{S}$ being the intersection of the individual sets: $\mathit{pooled}(r, \texttt{top}, \texttt{n1}) * \mathit{pooled}(r, \texttt{top}, \texttt{n2})$ tells us that pool r contains `n1` and `n2`.

Lifting our well-formedness predicate $wf(s)$ to a region predicate, we capture what it means to know that a pool r is well-formed:

$$\mathit{wfpool}(r, \texttt{top}) \triangleq \mathit{region}(\{S \mid wf(S)\}, T_x, I_x(\texttt{top}), r)$$

Now, we consider T_x. We define it in figure 3, as a map from the atomic actions to their transitions. As in the weak transitions from section 3, each transition is given as a 'before and after' relationship between two schematics of state tuples. Now, however, we add additional constraints on the pre–and post-states.

Many of the constraints we add here concern insertion orderings. For example, $\mathit{allT}_{\geq}(a, b, T, \mathcal{T})$ means that a was inserted after b, and everything inserted in the range $[a, a \wedge_\mathcal{P} b)$ (where $a \wedge_\mathcal{P} b$ is the meet point, or greatest common physical descendent, between a and b) must be taken:

$$\mathit{allT}_{\geq}(a, b, T, \mathcal{P}, \mathcal{T}) \triangleq a >_\mathcal{T} b \land \forall c \cdot (a \geq_\mathcal{T} c >_\mathcal{T} a \wedge_\mathcal{P} b) \implies c \in T$$

The idea behind these constraints is that, if compression works correctly, then the methods can imagine that insertion and physical order are equal. Much of

$$\begin{aligned}
T_x(\textsc{Push}) &\triangleq \langle N, T, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle \\
&\rightsquigarrow \begin{pmatrix} \langle N \uplus \{n\}, T, n, s, \mathcal{P} \uplus \{(n, m)\}, \mathcal{T} \uplus \{(n, i)\}, E \uplus \{(n, e)\}, n\rangle \\ \wedge\, i \geq_{\mathcal{T}} m \geq_{\mathcal{T}} t \wedge \mathit{allT}_{\geq}(i, m, T, \mathcal{P}, \mathcal{T}) \wedge \mathit{allT}_{\geq}(m, t, T, \mathcal{P}, \mathcal{T}) \end{pmatrix} \\
T_x(\textsc{Pop}) &\triangleq \langle N \uplus \{n\}, T, t, s, \mathcal{P}, \mathcal{T}, E \uplus \{(n, -)\}, i\rangle \\
&\rightsquigarrow \langle N, T \uplus \{n\}, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle \\
T_x(\mathrm{FC}) &\triangleq \langle N, T, t, s, \mathcal{P} \uplus \{(a, b)\}, \mathcal{T}, E, i\rangle \wedge a >_{\mathcal{T}} c \wedge\ \mathit{allT}_{\geq}(b, c, T, \mathcal{P}, \mathcal{T}) \\
&\rightsquigarrow \langle N, T, t, s, \mathcal{P} \uplus \{(a, c)\}, \mathcal{T}, E, i\rangle \\
T_x(\mathrm{BC}) &\triangleq \langle N, T, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle \wedge\ \mathit{allT}_{\geq}(t, c, T, \mathcal{P}, \mathcal{T}) \\
&\rightsquigarrow \langle N, T, c, s, \mathcal{P}, \mathcal{T}, E, i\rangle \wedge\ \mathit{allT}_{\geq}(t, c, T, \mathcal{P}, \mathcal{T})
\end{aligned}$$

Fig. 3. The transition relation T_x, refining the informal transitions from section 3.

our proof thus involves constraining the relationship between the two orders. Finally, if `top` contains the address of our pool's top node, our I_x is[4]

$$I_x(\texttt{top})(\langle N, T, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle) \triangleq I_{nodes}(\mathcal{P}, N, T, E) * \texttt{top} \mapsto t \wedge t \in N \cup T$$

$$I_{nodes}(\mathcal{P}, N, T, E) \triangleq$$

$$\circledast\, a \in N \cup T \cdot \exists n \in N \cup T,\ b \in \mathit{Bool},\ e \in \mathit{Elem} \cdot$$

$$\mathit{node}(a, e, b, n) \wedge (a, n) \in \mathcal{P} \wedge b = \mathit{false} \iff (a \in N \wedge e = E(a))$$

$$\mathit{node}(x, e, s, n) \triangleq x.\texttt{element} \mapsto e * x.\texttt{taken} \mapsto s * x.\texttt{next} \mapsto n$$

Method proofs. To reduce our stability obligation from considering every possible interleaving of statements in every method to considering only the atomic actions in T_x, we must now show that each pool method only modifies the region through correct use of those atomic actions. We must also show that *wfpool* holds throughout every method, assuming the stability result we prove later.

To do this, we use Hoare-style reasoning: we produce a proof outline, or a sequence of judgements, over every command C_n in the method, $\{P_n\}\ C_n\ \{P_{n+1}\}$, where P_n is the pre-condition of C, and P_{n+1} its post-condition (and the pre-condition for C_{n+1}). We aim to show here that, given some sufficiently strong first and last P that are consistent with our intuition of the pre-conditions and post-conditions of each method, each C changing the shared state represents one of the atomic actions from T_x. This ensures that the proof can rest on the stability of the predicates used in the P conditions over T_x.

We reason about the method entirely using region predicates until we reach an atomic action. At this point we open the region, using I_x to translate our

[4] $\circledast\, a \in A \cdot P(a)$ denotes a separate heap for each a such that $P(a)$ holds in that heap, ie $P(a_1) * P(a_2) * \ldots * P(a_n)$

predicates to a concrete heap onto which we apply our atomic action. After this, we close the region: we must show that the region predicate after close can be mapped by I_x into our modified concrete heap, and that the change from original to modified region predicates is consistent with an action in T_x. This reduces the burden of stability proof from checking each predicate against each method step to that of checking against each action in T_x.

Now, we show that, for every atomic action a and predicate p, p is stable with regards to a. To demonstrate, suppose we want to prove wf_{os} stable. First, only $\mathcal{P}$ and $\mathcal{T}$ are mentioned in wf_{os}; this means that only atomic actions affecting those can invalidate the predicate: Push and FC. For Push, we add $\{n, m\}$ to $\mathcal{P}$ and $\{n, i\}$ to $\mathcal{T}$. From the transition, we know that $i \geq_{\mathcal{T}} m \geq_{\mathcal{T}} t$, and as wf_{os} originally held then we can deduce that $n \geq_{\mathcal{T}} m$, $n \geq_{\mathcal{P}} m$, and the transitive orderings added to both sides must preserve the subset relation. For FC, we note that we substitute some $\{a, c\}$ for $\{a, b\}$ in $\mathcal{P}$, but $b >_{\mathcal{T}} c$, thus the resulting physical order is still a subset of insertion order.

5 Evaluation of the Proof Method

In this section we evaluate whether *iCAP* was a suitable tool for verifying the SP pool. Largely, we feel our use of *iCAP* has been a success. It provides a well-defined structure for organising proofs, reducing mental burden and improving confidence. This has helped us debug our proofs and discover previously unknown properties of the pool. For example, stability of a predicate $S_t = x$ for some x was required by an early insertion proof, and falsified when attempting to prove it under BC; we thus discovered the race between backwards compression and insertion. However, we note room for potential improvement on our method:

Leaky abstraction. The usual stack abstraction is a sequence, with popping and pushing occurring on one end. Our abstract view of the pool is the 8-tuple, which provides only a thin layer of abstraction, and leaks pool implementation details, such as: taken-ness, partial physical ordering, and sentinels. Abstract predicates help by hiding these under a concept of accumulating facts about the pool. However, being able to relate the concrete details of the pool to a more abstract notion of a stack in the logic, perhaps through some form of abstraction refinement, may assist with answering questions about functional correctness.

Unused complexity. The *iCAP* logic targets re-entrant, higher-order concurrent algorithms. Therefore, *iCAP* is large, with many concepts not used in this work. This is not a major problem, as the complexity rarely affects our particular proof, but our case may be better served with a smaller logic. For example, a smaller logic may be easier to automate—the lack of automation in *iCAP* makes our proof difficult to check. Frameworks such as *Views* [2] or *IRIS* [5] may make defining a small, focused logic on top of an existing, sound foundation easier.

Abstract state should be separable. If our abstraction was a sequence, we could use the machinery of separation logic to pull out parts of the sequence and work on them while holding the rest of the sequence invariant. Working with list segments and related shapes is well-understood in separation logic.

However, because our abstract state is effectively a collection of assorted set-theory facts about the concrete state, none of the tools of separation logic are available in *iCAP* until we open the region. This is a missed opportunity, because our transition relations must account for every part of our tuple, even if only one item is changed; this can be seen in the FC transition. If we could separate the parts of state we know will not change, and work only with the changing elements, this would eliminate much redundancy in our proof.

Again, abstract predicates partially help by masking parts of tuples for which we hold no knowledge. However, eventually, our proof must deal with the fact that $\mathit{pooled}(r, a)$ expands towards the set $\{\langle N, T, t, s, \mathcal{P}, \mathcal{T}, E, i\rangle \mid a \in N \cup T\}$.

We could abandon abstraction and work with the separable concrete state, but our tree-shaped state is difficult to work with using standard separation logic machinery. Work on logics for tree structures, such as that by Calcagno *et al.* [1], may fit better because the pool tree would be a first-class structure in the logic. This would give us logical tools to separate and re-combine pool fragments.

6 Conclusion and Future Work

Our aim was to prove the stability of various well-formedness properties on the SP pool; we now have a first draft of that proof. The ideal next step is to finalise and release it. One could then adapt the proof to a more suitable concurrent logic, to address the impedance mismatch described in section 5.

To prove the SP pool correct, we would look to a formal correctness condition, such as linearisability. Initial attempts using the Stack Theorem of Dodds et al. [3] show promise. These attempts depend on our *wf* invariant, but do not interact with *iCAP*. The Stack Theorem is based on the demonstration that linearisability-violating chains of events across the pool's lifetime cannot happen; iCAP is more suited to per-method and per-action specifications. A technique for order-based linearisability in the logic would complement the existing work.

References

1. Calcagno, C., Dinsdale-Young, T., Gardner, P.: Adjunct elimination in context logic for trees. Information and Computation 208(5) (2010)
2. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: Compositional reasoning for concurrent programs. POPL (2013)
3. Dodds, M., Haas, A., Kirsch, C.M.: A scalable, correct time-stamped stack. POPL (2015)
4. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. TOPLAS 12(3), 463–492 (Jul 1990)
5. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. pp. 637–650. POPL (2015)
6. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. PODC '10 (2010)
7. Svendsen, K., Birkedal, L.: Impredicative concurrent abstract predicates. In: Shao, Z. (ed.) Programming Languages and Systems (2014)