Modular Verification of a Non-Blocking Stack

Matthew Parkinson
Computer Laboratory,
University of Cambridge
Cambridge CB3 0FD, UK
matthew.parkinson@cl.cam.ac.uk

Richard Bornat
School of Computer Science
Middlesex University
London NW4 4BT, UK
R.Bornat@mdx.ac.uk

Peter O'Hearn
Department of Computer Science
Queen Mary, University of London
London E1 4NS, UK
ohearn@dcs.qmul.ac.uk

Abstract

This paper contributes to the development of techniques for the modular proof of programs that include concurrent algorithms. We present a proof of a non-blocking concurrent algorithm, which provides a shared stack. The inter-thread interference, which is essential to the algorithm, is confined in the proof and the specification to the modular operations, which perform push and pop on the stack. This is achieved by the mechanisms of separation logic. The effect is that inter-thread interference does not pollute specification or verification of clients of the stack.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—correctness proofs, formal methods validation; F.3.1 [*Logics and meanings of programs*]: Specifying, Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Separation Logic, Concurrency, Non-blocking

1. Introduction

Concurrent separation logic [11, 4] is a program resource logic based on the notion that separate parts of a program that depend on separated resources can be dealt with independently. Dijkstra's advice [5] on the design of concurrent programs was to limit interference to (rare) moments of synchronisation. So far, proofs in concurrent separation logic have followed this advice, considering the resources of separate threads independently, and at synchronisation points temporarily adding the resources owned by a semaphore or some other unit of mutual exclusion. This has led to pleasingly modular proofs of some well-known problems: for example, parallel merge sort, pointer-transferring buffers, and the classic readers-and-writers [11, 1, 2].

But synchronisation using units of mutual exclusion is not the only way of controlling interaction between threads, and in multiprocessor concurrent programming it is widely criticised for efficiency reasons because threads may often have to wait to gain exclusive access to the unit. In so-called 'non-blocking' concurrency, threads attempt to make concurrent changes to shared data structures, looping and trying again if a particular attempt fails to achieve the desired result. It is beyond the scope of this paper to judge if and when such mechanisms are to be preferred to mutual exclusion techniques.

Previously, the most effective formal treatment of non-blocking concurrency has been as 'interference' in the rely-guarantee approach [9, 12]. Each thread must specify properties which it guarantees to preserve in interaction with the shared data, and other properties which it relies on other threads to preserve. The guarantee must be preserved by each action of the thread, and if necessary it is possible to appeal to the rely conditions in showing that the guarantee holds. 'Interference' between threads thus floods the proof of every client: every single instruction in every single thread must ensure the guarantees of its thread, because every specification must include those guarantees.

We show in the proof below that in separation logic interference may be contained even in non-blocking concurrency. We equip a shared data structure with an invariant, and we provide operations on the data structure as procedures. We allow non-blocking interaction by allowing concurrent executions of the procedures in concurrent threads. The procedures have conventional pre- and post-conditions which do not involve the invariant. Within the bodies of the procedures the invariant is exposed, and the proof engages with it. Even though those proofs may be horrible, their horrors are confined, and we can consider threads as independent when outside those procedures.

Non-blocking concurrent algorithms rely for their correctness on certain hardware properties. They all rely on the atomic nature of single-word access to the store: that is, that reads and writes to any particular single-word location are effectively serialised and cannot overlap or be reordered. As in our example, many also rely on a `CAS` (compare and swap) instruction that can *atomically* read a location, compare its value with some previously-determined value and, if equal, overwrite it with a new value. An atomic access to the data structure (a single read or write of a single-word value, or a single `CAS`) is treated as a unit of mutual exclusion: we temporarily add the resources described by the invariant, prove the effect of the atomic operation, re-establish the invariant, and once again separate ourselves from it.

2. Concurrent Separation Logic

We simplify the Brookes/O'Hearn presentation [11, 4] by using a single invariant rather than a set indexed by resource names. We refer readers to those descriptions, and give here only the differences of our usage. We make use of procedure-call specifications $\{Q\}\, f(\overline{x})\, \{R\}$ and an invariant for the data structure shared between threads. Then

$$\Gamma; I \vdash \{Q\}\ C\ \{R\}$$

POPL'07 January 17–19, 2007, Nice, France.

means that with procedure specifications Γ, and a shared-data invariant satisfying I, the command C satisfies the specification $\{Q\}\, C\, \{R\}$.

The shared data structure can only be validly accessed within atomic operations, and to do so brings the invariant temporarily into play.

$$\frac{\Gamma; \mathbf{emp} \vdash \{Q \star I\}\ C\ \{R \star I\}}{\Gamma; I \vdash \{Q\}\ \texttt{atomic\{}C\texttt{\}}\ \{R\}} \quad (1)$$

C must be executed in mutual exclusion with other 'atomic' commands. In principle this might be achieved by the mechanisms of transactional memory [8, 6], but in this paper we rely on the serialisation properties of the hardware: C must invoke at most one single-word read or write in the shared data structure governed by I, or be a `CAS` instruction. Semantically we rely on Brookes' proof of soundness, which enables us to treat the shared data structure as a single 'resource' and use a version of separation logic's CCR rule.[1]

Parallel composition requires that we satisfy pre-conditions separately, and guarantees that the post-conditions are separate on termination. The data-structure invariant is shared between the two parallel compositions, just like named resource invariants in concurrent separation logic.

$$\frac{\Gamma; I \vdash \{Q1\}\ C1\ \{R1\} \quad \Gamma; I \vdash \{Q2\}\ C2\ \{R2\}}{\Gamma; I \vdash \{Q1 \star Q2\}\ C1 \| C2\ \{R1 \star R2\}} \quad (2)$$

The bodies of module procedures can make use of the invariant; the rest of the program cannot. We state the module rule for a single procedure definition; extension to several procedures is obvious. Note that the invariant is not incorporated into the pre- and post-conditions of C_f: atomic instructions inside C_f can make use of the invariant, but the procedure body is not itself treated as an atomic instruction.[2]

$$\frac{\begin{array}{c}\Gamma; I \vdash \{Q_f\}\ C_f\ \{R_f\} \\ \Gamma, \{Q_f\} f(\overline{x_f}) \{R_f\}; \mathbf{emp} \vdash \{Q\}\ C\ \{R\}\end{array}}{\Gamma; \mathbf{emp} \vdash \{Q \star I\}\ \texttt{module}\ f(\overline{x_f}) = C_f\ \text{in}\ C\ \{R \star I\}} \quad (3)$$

In order to construct our invariant we have had to make some use of permissions [1]. $E \mapsto F$ can be read as a total permission for the heap cell at location E – i.e. permission to read, write and dispose. It can be split into a collection of read-only permissions, which can then be given to separate threads, enabling read-only sharing. In this paper we require only that the invariant shares some permissions with each thread, so that we can split a total permission into two read permissions, one for the invariant and one for the thread:

$$E \mapsto F \iff E \overset{r}{\mapsto} F \star E \overset{r}{\mapsto} F \quad (4)$$

To read a heap cell we only need a read permission:

$$\{E = N \wedge N \overset{r}{\mapsto} M\}\, x := [E]\, \{N \overset{r}{\mapsto} M \wedge x = M\}$$

[1] It follows that `atomic{atomic{...}}` would fail to terminate.

[2] In this rule we have used **emp** as the invariant formula outside the proof of C_f. It would be possible to take a more general approach: the module rule can be derived from a version of the make-named-resource rule

$$\frac{\Gamma; I1 \star I2 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I1 \vdash \{Q \star I2\}\ C\ \{R \star I2\}}$$

and the resource-weakening rule

$$\frac{\Gamma; I1 \vdash \{Q\}\ C\ \{R\}}{\Gamma; I1 \star I2 \vdash \{Q\}\ C\ \{R\}}$$

Note that these two rules can be combined to give the standard separation-logic frame rule.

```
pop() {
  local t,n;
  while(true) {
    t = TOP;
    if (t == nil) break;
    H[tid] = t;
    if (t != TOP)
      continue;
    n = t->tl;
    if (CAS(&TOP,t,n))
       break;
  }
  H[tid] = nil;
  return t
}
```

```
push (b) {
  local t,n;
  for(n=0; n<=THREADS; n++)
    if (H[n] == b)
      return false;
  while(true) {
    t = TOP;
    b->tl = t;
    if (CAS(&TOP, t, b))
      break;
  }
  return true
}
```

Figure 1. Source for non-blocking stack. The hazard pointer code is highlighted. For compactness of the presentation we use a `for` loop in `push` to perform the `scan` operation from Michael's algorithm.

If a read permission is in the invariant, any thread can read a shared cell in an atomic operation, but only the thread that has the matching permission can write it (also in an atomic operation).

3. Specification of the algorithm

Michael's algorithm, shown in figure 1, implements a shared stack, with operations `push` and `pop`. In [10], Michael proves the correctness of the stack algorithm using hazard pointers.

We can see this as a simple implementation of a storage allocator for fixed-size heap records: `pop` is a kind of `malloc` or `cons`, `push` a kind of `free`, the stack a kind of freelist. We can give separation-logic safety specifications for his operations, allowing for the possibility that `pop` will not work on an empty stack and, because of the vagaries of his mechanism, that `push` might fail:

Method	Pre-condition	Post-condition
`pop()`	**emp**	$(ret \mapsto \_) \vee (ret = \text{nil} \wedge \mathbf{emp})$
`push`(x)	$x \mapsto \_$	$(ret \wedge \mathbf{emp}) \vee (\neg ret \wedge x \mapsto \_)$

Note that this specification need say nothing about the way that `push` and `pop` are implemented. A client is completely insulated from those details.

3.1 A client

Because the interface presented by the stack is somewhat inconvenient, and to demonstrate that the client is insulated from the interference used in the stack, we show how to build a simple memory manager.

We require an `alloc` mechanism, which works even if the stack is empty. We presume that the records on the heap are single-cell records, which originated from the system memory allocator `new`. `Alloc` tries `pop`, and if that fails, uses the (perhaps much slower) `new`.

$\{\mathbf{emp}\}$
```
alloc() {
  local y;
  y=pop();
  if (y==nil) y=new();
  return y;
}
```
$\{ret \mapsto \_\}$

We present a brief sketch of `alloc`'s verification:

$\{\mathbf{emp}\}$

```
  y=pop();
{(y = nil ∧ emp) ∨ y ↦ _}
  if (y==nil) {
    {emp}
      y=new();
    {y ↦ _}
  }
{y ↦ _}
```

Verification is trivial depending only on the specification of `pop` and the axiom $\{\mathbf{emp}\}$ `y=new();` $\{y \mapsto \_\}$. The point is that we can make these proof steps without any reference to the complex interference that we will need to account for when proving the module procedures, illustrating the sense in which interference is contained.

We require a `free` which does not bother us if `push`ing fails. This is not hard: we can keep a local list `fl` of the elements `free`d which have not yet been `push`ed, and each time we call `free`, try to `push` the lot. Specification requires a list predicate for our single-cell records.

$$list(x) \stackrel{\text{def}}{=} (x = \text{nil} \wedge \mathbf{emp}) \vee (\exists n \cdot (x \mapsto n \star list(n))) \quad (5)$$

```
{x ↦ _}
free(x) {
  local y;
  y = fl;
  [x] = y;
  fl = pushall(x);
}
{emp}
```

```
{list(l)}
pushall(l) {
  local n;
  if(l==nil) return l;
  n = [l];
  n = pushall(n);
  if(push(l)) return n;
  else { [l] = n; return l; }
}
{list(ret)}
```

`Free` uses the local freelist `fl`, in a straightforward use of separation-logic's hypothetical frame rule: the body has precondition $x \mapsto \_ \star list(\texttt{fl})$ and postcondition $\mathbf{emp} \star list(\texttt{fl})$. Verification of `pushall` is straightforward, depending only on the specification of `push`.

An essential point is that verification of `alloc`, `free` and `pushall` does not have to consider the non-blocking stack beyond the specifications of `push` and `pop`. The proof is completely isolated from the complexity of the interference that goes on inside those operations.

3.2 Without hazard pointers

Michael assumes a fixed population of threads. Each thread has its own local variables, named in lower case `t`, `n`, `b`, etc. They share the stack through a global variable named `TOP`, which can be read atomically (e.g. in `t=TOP`) and is written in atomic CAS instructions otherwise involving local variables (e.g. in `CAS(&TOP,t,b)`).

Michael describes first an algorithm in which the highlighted code of figure 1 is elided. In this algorithm, `push(b)` works reliably and need not return a success/fail result (i.e. it has postcondition $\mathbf{emp}$): if it finds a stack whose head is pointed to by `t`, it atomically replaces it with one whose head is pointed to by `b`. `Pop` is flawed, however: the fact that the head of the stack is pointed to by `t` does *not* imply that there is a second element pointed to by a value `n` which was previously found in the cell pointed to by `t`. By removing the top of the stack, modifying what comes below and then replacing the original top element, another thread can destroy the `TOP`/`n` association in the temporal interval between `n=t->tl` and `if CAS(&TOP,t,n)`: see figure 2. Michael refers to this as the ABA problem: a stack may be in state A (with `t` on top), then change to state B (some other stack), then revert to state A (`t` on top again) yet we know nothing else about this second A-state other than the fact that the variable `TOP` stores the same value as before.

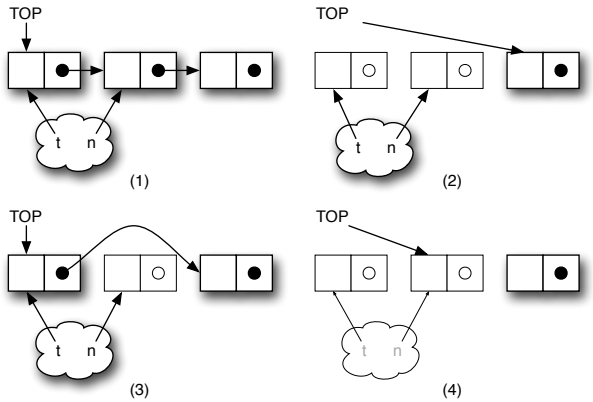

Figure 2. ABA problem for stack algorithm without hazard pointers. A thread begins to `pop` an element from the stack. It reads `t` and `n`, (1), but is then preempted. Another thread `pop`s two elements from the stack, (2), and `push`es the first element back onto the stack, (3). The original thread is rescheduled and attempts its CAS, which succeeds and destroys the data-integrity of the stack, (4).

3.3 Stack using hazard pointers

To fix the ABA problem, Michael adds a global array `H` of 'hazard pointers'. He assumes a fixed collection of numbered threads: thread `i` alone writes to element `H[i]` of the array, but all threads can read the elements written by others. In figure 1 we assume that each thread stores its number in a local variable `tid`. The global array is, in fact, a heap record pointed to by `H`, and for readability we write `H[E]` instead of `[H+E]`.

Before `push`ing a cell, a thread scans `H` to see if the cell is recorded as a hazard pointer by some other thread: if it is, the `push` is delayed (`return false`); if not, then the `push` proceeds, eventually returning `true`. Before `pop`ping a cell, a thread puts its address into its own element of `H`; then (and this is crucial!) checks that the cell is still on top of the stack before going on to read its tail and execute a CAS, clearing the hazard pointer before it returns.

The effect is remarkable: the algorithm works, and it can be implemented on any machine that provides CAS. It is a challenge to produce a formal proof in a program logic; in separation logic we can exploit the essential modularity of the algorithm, confining use of the invariant to the bodies of `push` and `pop`.

4. The proof

The effectiveness of the hazard-pointer array in Michael's algorithm derives from the following temporal property:

> Between execution of `H[i]=t` and assignment of another value to `H[i]`, the cell pointed to by `t` will not be removed from the stack and subsequently re-inserted. (†)

It is a subtle property: removal alone is possible; insertion alone is possible; insertion followed by removal is possible; only removal followed by insertion is prohibited. It is this property which dictates the test `t==TOP` in `pop`: if we know that at some instant the cell is on top of the stack then, because `H[i]` $=$ `t`, we know that it will not be re-inserted until we change `H[i]`; it follows that if the `CAS` finds it on top of the stack, then it has not been reinserted and the ABA problem is averted. A `push`ing thread can ignore hazard settings that are made *after* it has `pop`ped a cell b from the stack, so it can

```
1  pop() {
2    local t,n;
3    while(true) {
4      atomic{ t = TOP; }
5      if (t == nil) break;
6      atomic{ H[tid] = t; H'[tid] = Req; }
7      atomic{ if (TOP != t) continue;
8          else H'[tid] = Tail(t->tl);
9      }
10     atomic{ n = t->tl;
11       if (H'[tid]!=Tail(n)) { H'[tid]=Left; }
12     }
13     atomic{ if (CAS(&TOP,t,n)) break; }
14   }
15   atomic{ H[tid] = nil; H'[tid] = Unset;}
16   return t;
17 }
```

```
1  push (b) {
2    local t,n;
3    for(n=0; n≤THREADS; n++) {
4      atomic{
5        if (H[n] == b) { G[tid] = Unset; return false;}
6        G[tid] = NotHaz(b,{0,...,n});
7      }
8    }
9    while(true) {
10     atomic{ t = TOP; }
11     b->tl = t;
12     atomic{ if (CAS(&TOP, t, b)) { G[tid] = Unset; break;} }
13   }
14   return true;
15 }
```

Figure 3. Algorithm with explicit atomicity assumptions and additional (highlighted) code to manipulate auxiliary state, which represent the informal temporal property

certainly ignore any assignments that occur after it has begun to scan the array `H`.

Clearly, the algorithm is over-cautious in the sense that a `pop`per uses the atomic test `t==TOP` as a surrogate for the more general property '`t` is somewhere in the stack', and a `push`er takes note of hazard-pointer settings that occur after it `pop`ped a cell but before it began to `push` – but these are merely efficiency considerations.

We also need to know a property of the stack: `push`ing and `pop`ping other cells does not affect the tail-value of cells already in the stack.

> Once a cell is inserted into the stack, its `tail` value will not be altered until it is removed. (‡)

In order to be able to express these properties in our invariant we consider a hazard pointer to be in one of four states:

(*Unset*) `H[tid]` $=$ nil;

(*Req*) `H[tid]` $= b$ but b has not been observed in the stack;

(*Tail*(k)) while `H[tid]` $= b$, b has been observed in the stack with tail k;

(*Left*) while `H[tid]` $= b$, b has been observed in the stack, but is now known not to be in the stack.

If a hazard pointer is in *Tail*(k) or *Left* state, it ought not to be `push`ed: we call such an entry a *confirmed hazard*. *Unset* and *Req* states merely help deductions. We record these states in an auxiliary array `H'`, and we use additional code in `pop`, shown in figure 3, to manipulate this array.

When we initially set a hazard pointer (line 6) its status is *Req*, because we have not observed the cell in the stack while it was hazard-pointered. If the atomic test on line 7 fails, we know that `t` $=$ `TOP`, and hence the value was in the stack while `H[tid]` $=$ `t`, so we record the tail-value of the cell (instantaneously, it is an auxiliary assignment) in `H'`.

When we perform `n = t->tl` on line 10, we may find that `t->tl` has changed: that is only possible if `t` has left the stack (second temporal property (‡)), and then (because of the first temporal property (†)) it will not come back till we alter `H[tid]`, so the `CAS` will be certain to fail: we record that fact by altering `H'`. On line 15 we nullify our hazard pointer entry.

In order to become a confirmed hazard (state *Tail*$(\_)$ or *Left*) a cell must be observed in the stack. A cell that is being `push`ed is not in the stack:[3] therefore, during the `push` operation, no thread's hazard pointer can change from *Req* to *Tail*$(\_)$. So a thread that is `pop`ping can assign that cell's address to its `H` entry and it will not matter to a `push`ing thread that has gone past that entry in its hazard search: it cannot possibly become *Tail*$(\_)$ unless there is a successful `CAS` in `push`, and it cannot become *Left* unless it first becomes *Tail*$(\_)$.

For the `push`er we need yet another auxiliary array `G` to capture that reasoning, with values *Unset*, if we are not currently trying to push a value, and *NotHaz*(b, S), if we are trying to `push` b, and b is not a confirmed hazard for the set of threads S. In figure 3, we add the necessary code to `push`.

4.1 The Invariant

Next we relate the informal temporal restrictions and the auxiliary state added in figure 3 in an invariant. We use T for the set of thread identifiers $\{0, \ldots, \texttt{THREADS}\}$. We use h and h' as a mathematical representation of the contents of the arrays `H` and `H'`.

The hazard-pointer array h and hazard-status array h' restrict each node y of the stack with tail z as follows

$$\begin{array}{l}\mathrm{RNode}(y, z, h, h') \stackrel{\text{def}}{=} y \mapsto z \\ \quad \wedge\, \forall i \in T \cdot (h(i) = y) \;\Rightarrow\; (h'(i) = \mathit{Tail}(z) \vee h'(i) = \mathit{Req})\end{array}$$

If y is hazard for thread i, $h(i) = y$, and y is in the stack then its status is either *Tail*(z) or *Req*: it cannot be *Left* or *Unset*. We can then apply this restriction to every element of the stack, represented as a list starting from y, as:

$$\begin{array}{l}\mathrm{RList}(y, h, h') \stackrel{\text{def}}{=} \\ (y = \mathrm{nil} \wedge \mathbf{emp}) \vee \exists z.\, \big(\mathrm{RNode}(y, z, h, h') \star \mathrm{RList}(z, h, h')\big)\end{array}$$

Next we consider how `G` restricts the hazard pointer and status arrays. We want to record, for each thread `i`, two facts about the cell it might be `push`ing: (1) it is not a confirmed hazard for any of the threads whose hazard-pointers have already been checked in

[3] This is a consequence of separation logic's separation principle: if all threads start with separated resources and the shared data structure has a separated invariant, and if all threads communicate only with atomic commands, then separation will be preserved.

the for-loop; (2) it is not in the stack, and no other thread is `push`ing the same cell.

Here we begin to play with permissions: crucially, we do not give `G`'s total permission to the invariant, only a read permission. This allows the invariant to depend upon the value of `G`, but does not give permission to any thread to modify `G` in an atomic operation. We give the other read permission for `G[i]` to thread `i`: that thread can then, in an atomic operation, modify that entry of `G`. Hence thread `i` can depend on the value of `G[i]` even outside an atomic block.

When pushing the cell pointed to by `b`, we should have preferred in the invariant to describe invariant properties of the stack and then to say also $\wedge\ \neg(b \mapsto \_ \star \mathrm{true})$ – the cell does not appear anywhere in the stack. But this is not enough to guarantee that the same cell cannot somehow be pushed by another thread (perhaps permission could be passed behind the scenes). We need to know that any thread that has got part-way through the `push`ing process actually owns the cell it is pushing. To do this we have used two more kinds of permission:

$$E \mapsto F \iff E \overset{w}{\longmapsto} F \star E \overset{e}{\mapsto} \_ \tag{6}$$

A w permission allows writing (and reading, but we shall not need to appeal to that); an e permission allows nothing, but ensures that the cell pointed to exists in the heap. We cannot have two existence permissions to the same cell:[4]

$$E \overset{e}{\mapsto} \_ \star E' \overset{e}{\mapsto} \_ \Rightarrow E \neq E' \tag{7}$$

Thread `i`, pushing the cell pointed to by `b`, gives $b \overset{e}{\mapsto} \_$ to the invariant and keeps $b \overset{w}{\longmapsto}$ for itself: it has lost total permission and so no rogue thread can begin to push a cell and then pass the cell to another thread that successfully `push`es it.

$$\mathrm{HCons}(h,h') \overset{\text{def}}{=} \circledast_{i\in T} \cdot \begin{pmatrix} (\mathtt{G}[i] \overset{r}{\longmapsto} \mathit{Unset}) \vee \\ \exists b, S \cdot \begin{pmatrix} (\mathtt{G}[i] \overset{r}{\longmapsto} \mathit{NotHaz}(b,S) \star b \overset{e}{\mapsto} \_\_) \\ \wedge\ \forall j \in S \cdot (h(j) = b \Rightarrow h'(j) = \mathit{Req}) \end{pmatrix} \end{pmatrix}$$

This has a particularly useful property:

$$\forall b,i,h,t \cdot\ \mathrm{HCons}(h,h') \star b \mapsto x \Rightarrow \mathrm{HCons}(h[i \mapsto b], h'[i \mapsto \mathit{Tail}(x)]) \star b \mapsto x \tag{8}$$

We combine the $\mathrm{RList}(\mathtt{TOP},h,h')$ and $\mathrm{HCons}(h,h')$ predicates into the algorithm's invariant. In addition, we also give a concrete representation to the auxiliary state h and h'. Note that in each case we give only a read permission to the invariant: it may depend upon the whole array but only thread `j` can alter `H[j]` or `H'[j]`.

$$\mathrm{Inv} \overset{\text{def}}{=} \exists h,h' \cdot \begin{pmatrix} (\circledast_{j\in T} \cdot \mathtt{H}[j] \overset{r}{\longmapsto} h(j) \star \mathtt{H'}[j] \overset{r}{\longmapsto} h'(j)) \star \\ \mathrm{HCons}(h,h') \star \mathrm{RList}(\mathtt{TOP},h,h') \star \\ \forall i \in T \cdot h(i) = \mathrm{nil} \Leftrightarrow h'(i) = \mathit{Unset} \end{pmatrix}$$

4.2 The actual proof

Finally we present some details of the proof. In particular, we show how our informal reasoning earlier can be given formally, using the invariant and auxiliary state. In what follows we use the shorthand: $E \doteq E'$ to mean $E = E' \wedge \mathbf{emp}$. Additionally, for compactness, we use Hoare's rule for jumps (`break`, `continue` and `return`), for example in a loop with invariant I and exit condition X, `break` has pre-condition X and post-condition false and `continue` has pre-condition I and post-condition false. The outline of the proof is given in figure 4. We discuss the four marked atomic commands in the rest of this sections.

In **(a)**, updating the hazards status requires

$$h(\mathtt{tid}) = \mathtt{t} \wedge \mathrm{HCons}(h,h') \star \mathtt{t} \mapsto n \Rightarrow \mathrm{HCons}(h, h'[\mathtt{tid} \mapsto \mathit{Tail}(n)]) \star \mathtt{t} \mapsto n$$

This holds because of (8).

In **(b)** we case split on whether `t` is still in the stack. If `t` is in the stack, we can use the following to prove the condition in the `if` will never hold.

$$h(\mathtt{tid}) = \mathtt{t} \wedge h'(\mathtt{tid}) = \mathit{Tail}(a) \wedge \mathrm{RList}(y,h,h') \wedge (\mathtt{t} \mapsto x \star \mathrm{true}) \Rightarrow x = a$$

If `t` is not in the stack, then we do not have permission to read the location, so we must use the empty read rule:

$$\{\mathbf{emp}\}\mathtt{x=[E];}\{\mathbf{emp}\} \tag{9}$$

This allows us to perform a racy read on a location, but we know nothing about `x`'s value afterwards. This is really capturing the algorithm's optimism as we do the read, but will fail later when we get to the CAS. However, we know it is valid to set the status to *Left* from the following property:

$$h(\mathtt{tid}) = \mathtt{t} \wedge \mathrm{RList}(\mathtt{TOP},h,h') \wedge \neg(\mathtt{t} \mapsto \_ \star \mathrm{true}) \Rightarrow \mathrm{RList}(\mathtt{TOP}, h, h'[\mathtt{tid} \mapsto \mathit{Left}])$$

This holds by induction on the definition of $\mathrm{RList}(x,h,h')$, and because $- \wedge \neg(\mathtt{t} \mapsto \_ \star \mathrm{true})$ distributes over $\star$.

In **(c)**, we case-split on the hazards status. If the status is *Left* we know the CAS will fail, because of the following implications:

$$\mathtt{H}[\mathtt{tid}] \overset{r}{\longmapsto} \mathtt{t} \star \mathtt{H'}[\mathtt{tid}] \overset{r}{\longmapsto} \mathit{Left} \star \mathrm{Inv} \Rightarrow \mathtt{TOP} \neq \mathtt{t}$$

If the status is $\mathit{Tail}(\mathtt{n})$ and the CAS fails, then the proof holds trivially. If the CAS succeeds, then the proof follows from the following implication:

$$h(\mathtt{tid}) = \mathtt{t} \wedge h'(\mathtt{tid}) = \mathit{Tail}(\mathtt{n}) \wedge \mathrm{RList}(\mathtt{t},h,h') \wedge \mathtt{t} \neq \mathrm{nil} \Rightarrow \mathtt{t} \mapsto \mathtt{n} \star \mathrm{RList}(\mathtt{n}, h[\mathtt{tid} \mapsto (\mathtt{t}, \mathit{Tail}(\mathtt{n}))])$$

In **(d)**, if the CAS succeeds we require

$$\mathrm{HCons}(h,h') \star \mathtt{G}[tid] \overset{r}{\longmapsto} \mathit{NotHaz}(b,T) \Rightarrow \forall i \in T \cdot h(i) = b \Rightarrow h'(i) = \mathit{Req}$$

to know the value being pushed it not a confirmed hazard, and

$$b \mapsto t \star \mathrm{RList}(t,h,h') \star (\forall i \in T \cdot h(i) = b \Rightarrow h'(i) = \mathit{Req}) \Rightarrow \mathrm{RList}(b,h,h')$$

to know that it is valid to add it to the stack.

5. Conclusions and Future work

We do not claim to have made the first proof of Michael's algorithm; nor do we claim that our proof is simple; nor do we claim to have proved everything that is important, such as absence of livelock.

We have formally contained the inter-thread interference within the operations on the shared stack, as Michael clearly intended it to be contained. Modularity is essential for scalability of proof, and interference containment is an important kind of modularity. We are not aware of a proof of a non-blocking algorithm in another logic which provides such modularity. In all other proofs we have seen, the possibility of interference floods into specifications and proofs that do not have anything to do with it.

[4] Permissions models require that there is no zero permission and that total permission is a maximum. We have given only the axioms that are required in our proof.

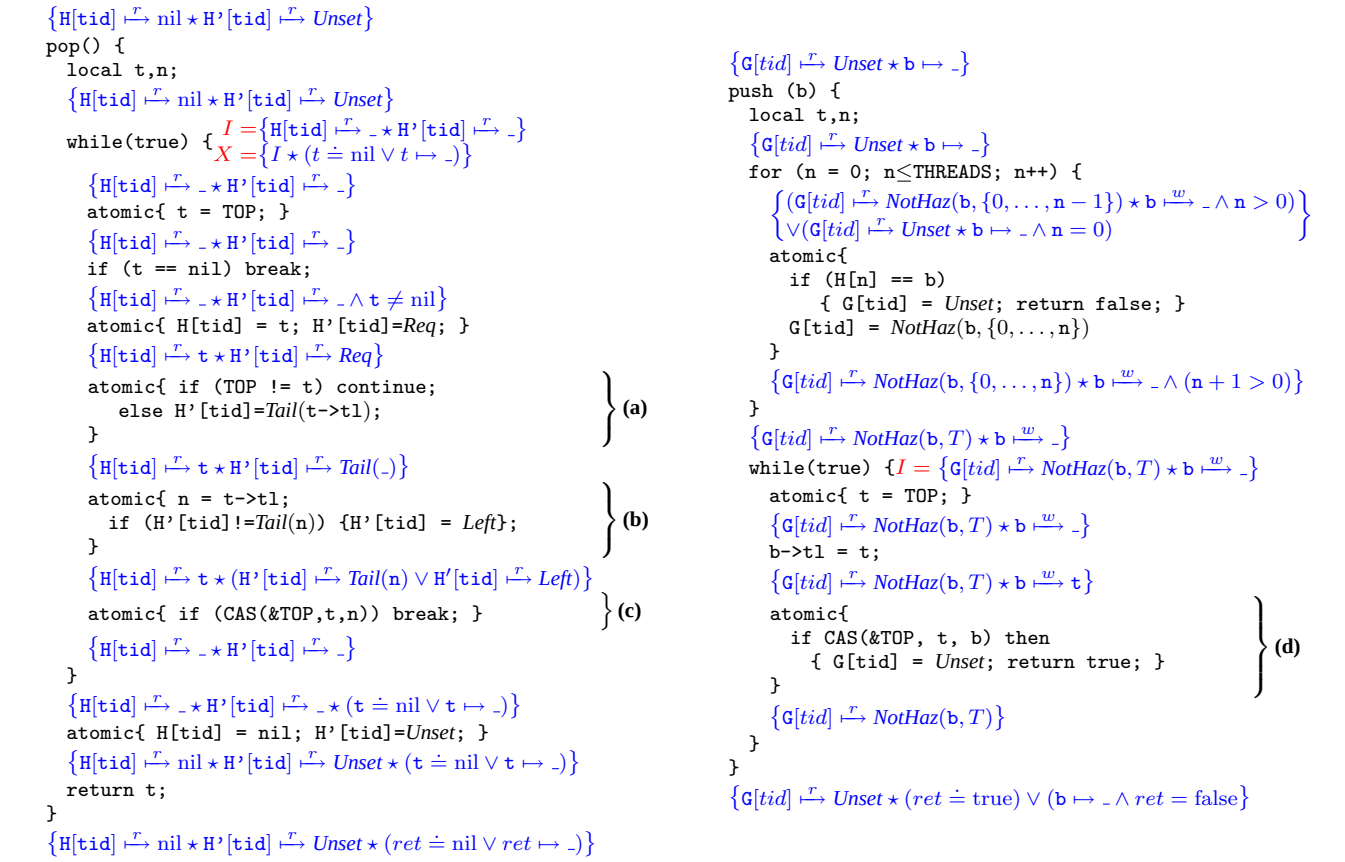

Figure 4. Outline proof

The stack operations can be used by only considering the pre- and post-conditions. This is similar to linearisability [7], which allows concurrent linearisable operations to be considered like atomic actions. We do not require an operation to be linearisable to reason purely by its pre- and post-condition.

In this paper, we have not attempted to show more than the safety of certain operations, that they do not leak memory and that they preserve the invariant of the shared stack. We are investigating adding liveness rules to separation logic to capture properties such as obstruction/lock/wait-freedom.

Finally, we intend to explore the combination of mechanisms from rely/guarantee and temporal logic with those of separation logic. The aim is a logic which exploits separation logic's modularity but provides a simpler treatment of interference than is possible in that logic alone; hopefully, for example, that would reduce the use of auxiliary state in proofs.

Acknowledgments We should like to thank the East London Massive, Bart Jacobs, Rok Strnisa and Viktor Vafeiadis for feedback on this work. This work was supported by the EPSRC (Bornat, O'Hearn and Parkinson), the Royal Academy of Engineering (Parkinson) and Intel Research Cambridge.

References

[1] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, pages 259–270, 2005.

[2] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Proceedings of MFPS XXI*. Elsevier ENTCS, May 2005.

[3] P. Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.

[4] S. Brookes. A semantics for concurrent separation logic. Invited paper, in *Proceedings of CONCUR*, 2004.

[5] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968. Reprinted in [3].

[6] T. Harris, S. Marlow, S. Peyton-Jones, and M. P. Herlihy. Composable memory transactions. In *Proceedings of PPOPP*, 2005.

[7] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[8] M. P. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[9] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[10] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[11] P. W. O'Hearn. Resources, concurrency and local reasoning. To appear in *Theoretical Computer Science*; preliminary version in CONCUR'04.

[12] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of PPoPP*, pages 129–136, 2006.