Memory barriers for LIFO IWSQ on Power

Richard Bornat
Computer Laboratory, University of Cambridge, Cambridge, UK
School of Electrical Engineering and Computer Science, Queen Mary, University of London, UK
School of Engineering and Information Sciences, Middlesex University, London, UK
rb501@cam.ac.uk, richard@eecs.qmul.ac.uk, R.Bornat@mdx.ac.uk

March 20, 2012

Abstract

Barriers for LIFO IWSQ on Power, done rather properly. Includes first sight of $\mathbb{T}$ and $\mathbb{U}$ modalities for universal stable assertions and the $\mathbb{B}$ and $\mathbb{C}$ operators for and CASable values.

This is an attempt to do the barriers for LIFO IWSQ systematically – i.e. with formal treatment of propagation effects. The algorithm is in figure 1 with the authors' ordering annotations, which are inadequate: there's an additional need for an lwsync before line 3 of put() (which can be made conditional, though it doesn't save very much in this algorithm). Note that sz is local to the owner since the stealer never reads it: there needs to be a global auxiliary Sz which tracks it, to simplify the invariants, which is ugly, but at least we don't have to consider propagation issues with that variable. Note that Power doesn't have two-word indivisible reads and writes, let alone CASs, so Ac, which holds both a stack pointer t and a generation number g has to be a single 64-bit value; this sounds rather impractical and makes the proof an academic (in the sense of 'footling and pointless') exercise.

1 A program logic for Power

The PowerPC has two peculiarities: it reorganises execution of instructions, speculating where it can; and it delays writes to memory whilst maintaining a global coherence order of writes that all readers must observe (Sarkar et al., 2011). I began my investigation of the FIFO IWSQ algorithm by drawing dependency graphs of procedures to describe constraints on out-of-order execution, and then putting RGSep-style (Vafeiadis and Parkinson, 2007; Vafeiadis, 2007) assertions on their branches. This worked surprisingly well, using the insight that global invariants must be a constraint on every thread's view of shared data (there is no shared memory in models of Power), so must be maintained locally at least.

Dependency graphs allowed me to abduce five out of the seven barriers needed for FIFO IWSQ (Michael et al., 2009), one more than noted by the algorithm's authors, but two of those had a memory propagation effect (a side-effect of the local preservation of invariants), and there were two more barriers, not noticed by those authors (who should be forgiven because they weren't really thinking about Power's memory mechanisms), which were purely needed to deal with propagation effects. Mike Dodds helped me to see that the notion of several views of shared data was a potential clarification of the problem, and we collaborated on a semi-formal treatment of the FIFO IWSQ algorithm (Bornat and Dodds, 2012).

Now I realise that the semi-formal treatment, which reasons about inter-procedural effects in a handwavy fashion, can be further formalised. The first step is to describe a semantics. In (Sarkar et al., 2011) there

```
void put(Task tv) {
     Order write in 3 before write in 4
1: (t,g) := Ac;
2: if (t = sz) {expand(); goto 1;}
3: Ta[t] := tv;
4: Ac := (t+1,g+1);
}

TaskInfo take() {
1: (t,g) := Ac;
2: if (t=0) return EMPTY;
3: tv := Ta[t-1];
4: Ac := (t-1,g);
5: return tv;
}

TaskInfo steal() {
     Order read in 1 before read in 3
     Order read in 4 before CAS in 5
1: (t,g) := Ac;
2: if (t=0) return EMPTY;
3: a := Ta;
4: tv := a[t-1];
5: if !CAS(Ac,(t,g),(t-1,g)) goto 1;
6: return tv;
}

void expand() {
     Order writes in 2 before write in 3
     Order write in 3 before write in put:4
1: a := new Tasks[2*sz];
2: for i = 0:sz-1, a[i] := Ta[i];
3: Ta := a;
4: sz := 2*sz;
}
```

Figure 1: LIFO IWSQ (from (Michael et al., 2009) with shortened names and lower case for local variables)

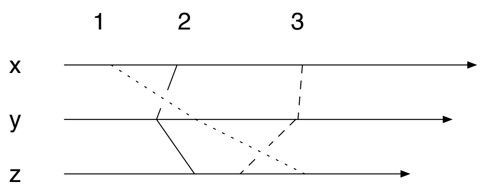


Figure 2: Three views of shared data

is no memory: reads are associated with the write which produced the value they read. There is a global *coherence order* of committed writes to each individual location, and threads must read in coherence order: not all from the same position, but two reads in a single thread can't read in the reverse of coherence order. It's possible to say what a *view* of shared data is: it's a set of positions on the coherence orders of the locations that a thread is interested in. In figure 2 there are three views of the shared variables x, y and z. View 1 (dotted) sees an earlier view of x than view 2 (solid line), but later views of y and z. View 3 (dashed) is later than view 2 at every variable. Assertions on the arcs of a dependency graph are about the executing thread's view of shared data, which may be changed by the thread's own assignment to a shared location – necessarily later than any write in the coherence order until it becomes committed – or by writes later than those in its view being propagated to it, moving its view of a location to a later view. An assertion about shared data made on an arc of a dependency graph is *stable* only if it is true in the thread's view and in every possible move to a later view it remains true.

As usual instantaneous assertions must be relaxed to take account of interference, which is now just a move to a later point in the coherence order, converting them into stable assertions, and as usual we describe interference by giving the context in which a single-word write – an action – by another thread can take place. But now there is an issue, because writes happen in stages rather than instantaneously and those stages happen at different times in different threads. We need, therefore, to describe the context in which a write is *received* in order to describe its effect. To do that we need universal stable assertions, of which invariants are just an example. Then just as in a conventional memory treatment we can assume the stability of invariants when calculating the effect of interference on a thread but must ensure that its actions preserve that stability, so we can use the constraints on other threads' actions, expressed as a shared-data context, and we must ensure that each issued action is received in a suitable context.

It all sounds a bit difficult, and for certain there is no free lunch. Universal stability is a product of constraints on interference, expressed as global invariants and contexts on actions. We can see that cumulative barriers – sync and lwsync – cause a pause until other threads have seen at least the writes received by the barring thread – i.e. moved to a view which is not earlier than the barring thread at any point – so that a locally-stable assertion asserted before a cumulative barrier becomes a universal stable assertion after it. The way that this interacts with the dependency graph needs careful treatment: sync provides the universal stable assertion to reads and writes which follow it, but lwsync provides it only to writes (reads see the locally-stable version). Given a universally stable assertion as precondition for a write to shared data we can prove that the write will be received by any other thread in a view which satisfies that assertion.

The last claim requires some qualification: we can be sure that a write will be well-received provided that the universally stable assertion which is its precondition *persists* until the write is committed and propagated to the receiving thread. That means that the precondition must be invariant at least until the next cumulative barrier in the writing thread, because that is the latest point at which the write can be committed and propagated. As we said, no free lunch: stability and persistence are harsh requirements.

So there's a new modality (?) $\mathbb{U}(P)$ which asserts that every thread sees shared stable P (as usual with concurrent proofs, would see P if it could take a snapshot of the data available to it). Universal assertions are created from stable assertions by sync / lwsync, and can appear in pre and postconditions. They are mostly useful to provide pre-conditions for actions (i.e. global writes).

The next problem is the interaction between CAS and sync/lwsync. Concurrent algorithms use sync/lwsync to ensure that their writes have a universal effect before they proceed, and they use CAS to ensure that there haven't been any relevant universal effects before they make a change. CAS on Power has a peculiar property: it ensures that it reads from the last write in the coherence order and puts a new write as the next (and new last) element in the coherence order. So a location which is only accessed by CAS must have a coherence order in which every step is a CAS-step. A location which receives conventional as well as CAS writes must have a coherence order which consists of conventional writes separated by CAS-chains starting at those writes. sync/lwsync barriers can be used to force a thread's writes into the coherence order and if, as in IWSQ algorithms, other threads can only CAS, we can say quite a lot about the CAS-chains they are then forced to construct.

So there's a new operator $\mathbb{C}(location)$ which refers to the value at the end of the coherence order for $location$. CAS alters it, and universal stability allows us to say things about it. In particular it allows me to justify a condition lwsync barrier in put(), which I desperately wanted to do.

2 Invariants and actions

There's an auxiliary task sequence Ts indexed $0..G-1$. I can't see any way to say which bits of it are in the array more than to say that the tasks a thread can see don't include those not yet added to Ts. I don't want to get into propagation effects with sz, so I made it a local variable, but there's an auxiliary Sz which tracks it so the main invariant (1) can describe the (apparently unused) part of the array that doesn't contain visible queue elements. (2) says that Ac_1, the index value, is never negative and never above Sz; (3) that Ac_2, the anti-ABA task-write count, only increases; (4) says that Sz increases.

$$\mathbb{U}\left(\exists T, G\left(Ac = (T,G) \wedge \circledast_{i=0}^{T-1}\left(\exists j(0 \leq j < G \wedge Ta + i \mapsto Ts_j)\right) \star \circledast_{k=T}^{Sz-1}\left(Ta + k \mapsto \_\right)\right)\right) \tag{1}$$

$$\mathbb{U}\left(\forall T\left(Ac = (T,\_) \Rightarrow 0 \leq T < Sz\right)\right) \tag{2}$$

$$\mathbb{U}\left(\forall G\left(Ac = (\_,G) \Rightarrow \Box\forall G'\left(Ac = (\_,G') \Rightarrow G \leq G'\right)\right)\right) \tag{3}$$

$$\mathbb{U}\left(\forall S\left(Sz = S \Rightarrow \Box\forall S'\left(Sz = S' \Rightarrow S \leq S'\right)\right)\right) \tag{4}$$

(5) is local to the owner.

$$sz = Sz \tag{5}$$

Any thread can reduce Ac_1, but thieves do it with a CAS in (6) and the owner does it more slowly in (7). The owner's action can increase T when a take() overlaps with two or more steal()s, and then we must be sure that thieves could access the exposed elements ever since they were last visibly part of the queue.

$$T > 0 : \quad \mathbb{C}(Ac) = (T,G) \rightsquigarrow \mathbb{C}(Ac) = (T-1,G) \tag{6}$$

$$\left(\begin{array}{l} 0 \leq k < T \wedge \\ \quad \exists Vs\left(\circledast_{i=0}^{T-1}\left(\mathbb{U}\left(\begin{array}{c} Ta + i \mapsto Vs_i \ \mathsf{S} \\ Ac = (T-1,G) \end{array}\right)\right)\right)\end{array}\right) : \quad Ac = (T-k,G) \rightsquigarrow Ac = (T-1,G) \tag{7}$$

To write into the array the owner must know that every thief agrees that the destination element is not part of the queue. To alter the index it must know that thieves have received the new queue value (and, if it increases by more than one, that any queue values it skips over have remained unchanged since they were last part of the queue):

$$sz = S \wedge \mathbb{C}(Ac)_1 \leq i < S : \quad Ta + i \mapsto V \rightsquigarrow Ta + i \mapsto V' \tag{8}$$

$$\left(\begin{array}{l} 0 \leq k \leq T \wedge \mathbb{C}(Ac)_1 \leq T \wedge \\ \quad \mathbb{U}\left(Ta + T \mapsto V\right) \star \\ \qquad \exists Vs\left(\circledast_{i=0}^{T-1}\left(\mathbb{U}\left(\begin{array}{c} Ta + i \mapsto Vs_i \ \mathsf{S} \\ Ac = (T,G) \end{array}\right)\right)\right)\end{array}\right) : \quad \begin{array}{l} Ac = (T-k,G) \wedge \\ \quad Ts = TS \end{array} \rightsquigarrow \begin{array}{l} Ac = (T+1,G+1) \wedge \\ \quad Ts = TS.V \end{array} \tag{9}$$

```
TaskInfo steal() {
1:  <(t,g) := Ac>;
2a:  if  t=0 then
2b:      return EMPTY;
3:  <a := Ta>;
4a: ae := a+t-1;
4b:      <tv := [ae]>;
5a: <b := CAS(Ac,(t,g),(t-1,g)>);
5b:  if  ¬b then
5c:      goto 1;
6:  return tv;
}
```

Figure 3: LIFO steal() with single-memory-access instructions

To switch arrays it must be the case that the new array is seen with the same queue values as the old.

$$Ac = (T, \_) \wedge \mathbb{U}\left(\circledast_{i=0}^{T-1}\left(\exists V\left(A+i \mapsto V \star A'+i \mapsto V\right)\right)\right): \qquad Ta = A \rightsquigarrow Ta = A' \tag{10}$$

Privately to the owner, to alter sz it must be the case that allocation of the new array elements has been seen by thieves (properly this isn't an action, but I put it here to jog my memory).

$$\mathbb{U}\left(\circledast_{j=S}^{2\times S-1}\left(Ta+j \mapsto \_\right)\right): \qquad sz = Sz = S \rightsquigarrow sz = Sz = 2\times S \tag{11}$$

3 Coherence orders

The array pointer Ta and the elements of the array it points to are all written only by the owner. It is clear that each of these locations has a linear coherence order consisting of the owner's writes in order, and therefore unproblematic to claim that lwsync in the owner converts an assertion about the value of any of these locations in the owner's view into a universally stable assertion.

The anchor Ac is written by the owner and CASed by thieves who reduce its first projection (T in (T, G)) by one. Since there is a single linear coherence order for Ac, we can be sure that it is a sequence of segments each consisting of a mini-sequence of the owner's writes followed by a chain of T-descending CAS writes from the last of that mini-sequence. It is then unproblematic to claim that lwsync in the owner inserts any uncommitted writes at the end of the coherence order and thieves must CAS, if they can, from the last of those writes. If there are no uncommitted writes then the coherence order is unchanged, but the same property applies: it ends with a (possibly empty) CAS chain starting from the last of the owner's writes.

4 LIFO steal()

Figure 3 shows steal() and figure 4 its implied dependencies. From experience I guess that we shall need

- an r/r barrier between lines 1 and 3, so that we read from a stack which might contain an item addressed by t;
- an r/w barrier between lines 4b and 5a, so that we read [ae] in the interval between line 1 and line 5a, before the check that Ac=(t,g).

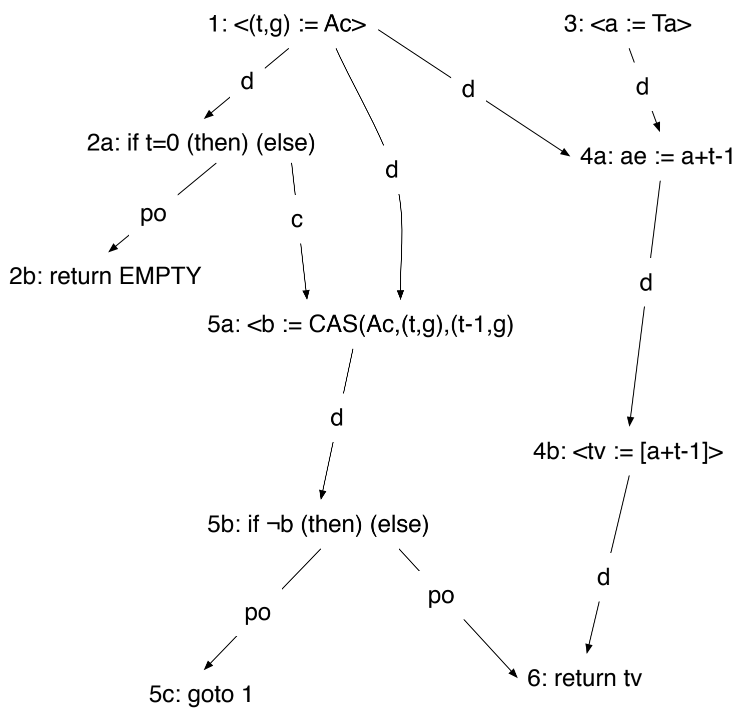


Figure 4: Dependencies in LIFO steal()

```
TaskInfo  steal ()  {
1:   <(t ,g)  :=  Ac>;
2a:  if   t=0 then
2b:        return  EMPTY;
2c:  --  line  1  /  any rw barrier  ( line  2a+isync) --
3:   <a := Ta>;
4a:  ae := a+t-1;
4b:        <tv  :=  [ae]>;
4c:  --  line  4b /  any rw barrier  ( ctrl +isync)  --
5a:  <b := CAS(Ac,(t,g),(t -1,g)>);
5b:  if  ¬b then
5c:        goto 1;
6:   return  tv ;
}
```

Figure 5: LIFO steal() with barriers

$\boxed{\exists T, G, J \left(Ac = (T, G) \wedge (T > 0 \Rightarrow Ta + T - 1 \mapsto Ts_J) \right)}$

`<(t,g) := Ac>`

$\exists T, G, J$

$t = T \wedge g = G \wedge \boxed{Ac = (T, G) \wedge (T > 0 \Rightarrow Ta + T - 1 \mapsto Ts_J)}$

$\Rightarrow$ (stability)

$t = T \wedge g = G \wedge \boxed{\exists V \left(T > 0 \Rightarrow (Ta + T - 1 \mapsto V \wedge (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J))) \right)}$

`if t=0`

(then)
emp

`return EMPTY`

(else)
$t = T \wedge g = G \wedge \boxed{\exists V \left(Ta + T - 1 \mapsto V \wedge (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J)) \right)}$

`isync`

`<a := Ta>`

$\exists A$

$t = T \wedge g = G \wedge a = A \wedge \boxed{\exists V \begin{pmatrix} Ta = A \wedge A + T - 1 \mapsto V \wedge \\ (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J)) \end{pmatrix}}$

$\Rightarrow$ (stability)

$t = T \wedge g = G \wedge a = A \wedge \boxed{\exists V \left(A + T - 1 \mapsto V \wedge (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J)) \right)}$

`ae:=a+t-1`

$t = T \wedge g = G \wedge ae = A + T - 1 \wedge \boxed{\exists V \left(A + T - 1 \wedge (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J)) \mapsto V \right)}$

`<tv := [ae]>`

$\exists V$

$t = T \wedge g = G \wedge tv = V \wedge \boxed{A + T - 1 \mapsto V \wedge (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J))}$

$\Rightarrow$ (stability)

$t = T \wedge g = G \wedge tv = V \wedge \boxed{A + T - 1 \mapsto \_ \wedge (\mathbb{C}(Ac) = (T, G) \Rightarrow \boxslash(V = Ts_J))}$

`ctrl+isync`

`<b := CAS(Ac,(t,g),(t-1,g)>)`

$(b \Rightarrow tv = Ts_J) \wedge \boxed{A + T - 1 \mapsto \_}$

`if !b`

(then)
$\boxed{A + T - 1 \mapsto \_}$

`goto 1`

(else)
$tv = Ts_J$

`return tv`

Figure 6: Formal proof of LIFO steal()

```
void put(Task tv) {
1:  <(t,g) := Ac>;
2a:  if  t=sz {
2b:      expand();
2c:      goto 1
     };
3a: <ta := Ta>;
3b: <[ta+t] := tv>;
4:  <Ac := (t+1,g+1)>;
5:  return
}
```

Figure 7: LIFO put() with single-memory-access instructions

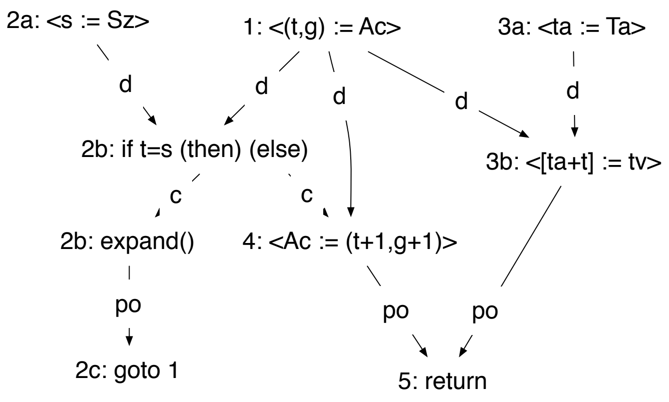


Figure 8: TIoSE dependencies in LIFO put()

We don't seem to need anything else. Barriered code in figure 5, ribbon proof in figure 6. I don't like the game I've played with $\exists$ and $\boxed{\sim}$[1], but the implication from $\mathbb{C}(Ac)$ is valid: it holds instantaneously after <(t,g) := Ac> and then it's stable because the owner only writes at a index higher than $\mathbb{C}(Ac)_1$ in (8) and then is committed to (9) as its next action affecting Ac; so subsequent interference can only be via (6). But it's clunky: I hope to discover an invariant which makes it obvious.

5 LIFO put()

Figure 7 shows put() with single-memory-access instructions and figure 8 the implied dependencies. Before the write on line 3b we need to know that

$$\mathbb{C}(Ac)_1 \leq t \tag{12}$$

– i.e. we are not writing where any thief will productively think there is a queue element. That needs at least an lwsync somewhere before line 3b to provide universality. We obviously need another lwsync between lines 3b and 4 (message passing idiom: make the actions (8) and (9) happen in order). The first lwsync is only necessary when there has been a take() since the previous put(). To make this work logically, I've made put() and take() use a local variable ws (for 'writesafe') and have as pre and postcondition

$$ws \Rightarrow \boxed{\mathbb{C}(Ac)_1 \leq Ac_1} \tag{13}$$

[1] 'Always since start': has held since the procedure began executing. Ok, ok.

```
void put(Task tv) {
1:  <(t,g) := Ac>;
2a: if  t=sz {
2b:      expand();
2c:      goto 1
     };
3a: <ta := Ta>;
3b: if  ¬ws then
3c:       -- rw/w barrier  (lwsync) --;
3d: <[ta+t]  := tv>;
4a: -- rw/w barrier  (lwsync) --
4b: ws := true;
4c: <Ac := (t+1,g+1)>;
5:  return
}
```

Figure 9: LIFO put() with barriers

(i.e. an intra-procedural invariant). Code in figure 9, proof in figure 10. Note that lwsync, together with the fact that the only interference is (6) from thieves, turns stable $Ac_1 \leq T$ into $\mathbb{U}(\mathbb{C}(Ac)_1 \leq Ac_1 \leq T)$. Useful, that: I shall use it in expand().

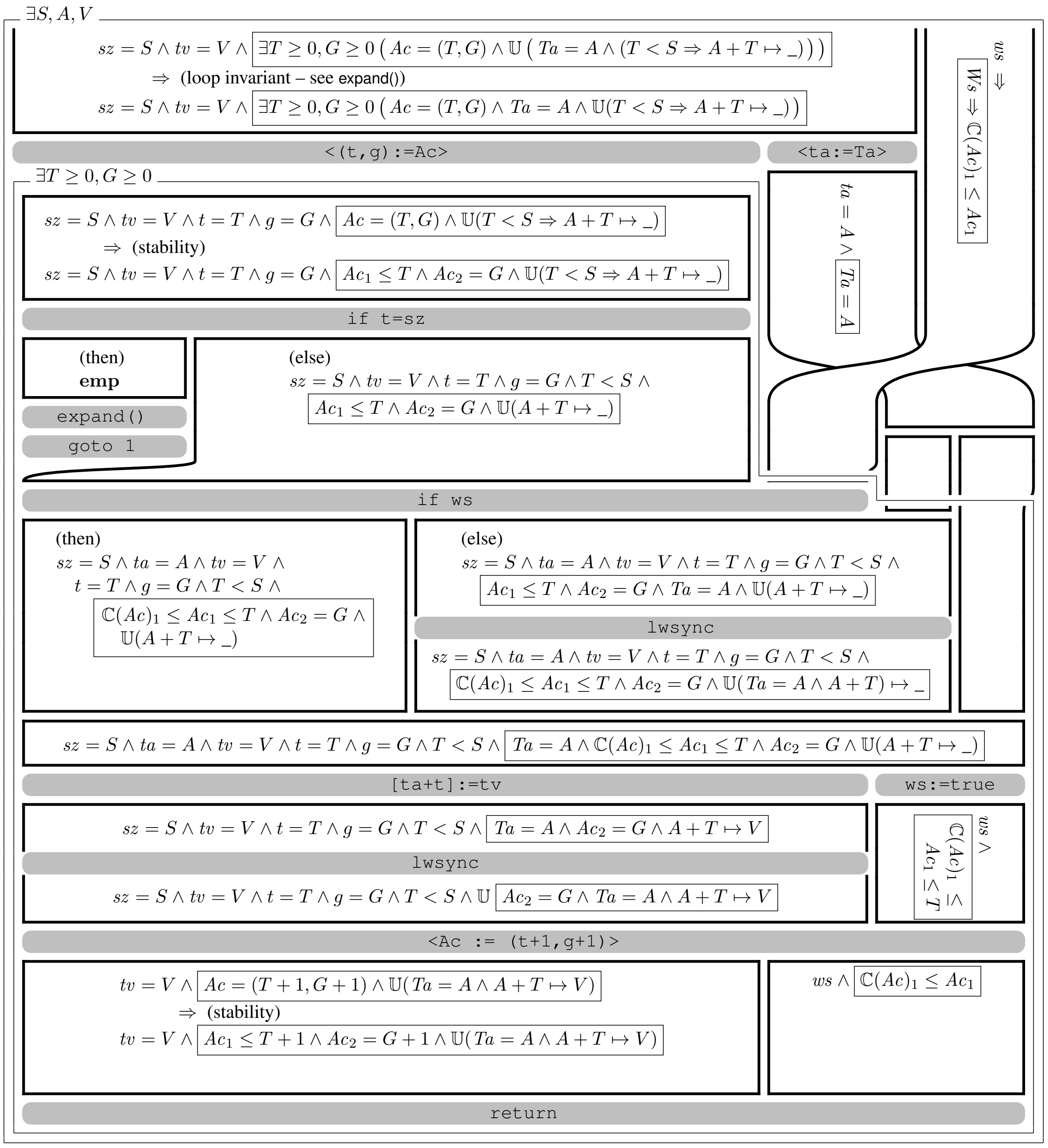
6 LIFO expand()

Code in figure 11, in which the for loop and the copying assignment have been unpicked to a low-level single-memory-access instruction program and the necessary barrier is included – an lwsync before line 3, so that all the assignments of line 1b and lines 2f-2h have been universalised before the new array is publicised. I've also assigned ws := true after the barrier. Proof in figure 12: I haven't been too fussy about following the dependency structure of lines 1a to 2j, because it all executes without interference and under TIoSE (The Illusion of Sequential Execution). so that it arrives at the lwsync with a stable local view of the new array. Note the precondition $\exists T(Ac_1 \leq T)$, which is stable because interference is only from (6). Note that the assignment to Ta performs the action of moving the new array from local to global space (a standard RGSep trick), and causes an obvious space leak.[2] Note how the assignment to ws avoids an lwsync in the enclosing put().

7 LIFO take()

Michael et al. say no barriers in take, which seems fair. Single-memory-access code in figure 13, which also includes an assignment ws := false after Ac := (t-1,g), which may falsify (13). Ribbon proof in figure 14. There's a strangeness in the proof: tv:=[ae] reads from the task array, but would generate an exception if $t = 0$ – i.e. it's an instruction which can have an effect, so you might think it should respect the control dependency from the 'else' side of t!=0. But when $t \neq 0$ the read doesn't have an effect, so the control dependency doesn't matter. I've chosen to show this by making the postcondition of tv:=[ae] conditional, to show that in TIoSE it can safely be speculated.

[2] The space leak is not very costly: the new array takes one more cell than the leaked space taken up by all its predecessors.



$\exists S, A, V$

$sz = S \wedge tv = V \wedge \boxed{\exists T \geq 0, G \geq 0 \left(Ac = (T,G) \wedge \mathbb{U}\left(Ta = A \wedge (T < S \Rightarrow A + T \mapsto \_) \right)\right)}$

$\Rightarrow$ (loop invariant – see expand())

$sz = S \wedge tv = V \wedge \boxed{\exists T \geq 0, G \geq 0 \left(Ac = (T,G) \wedge Ta = A \wedge \mathbb{U}(T < S \Rightarrow A + T \mapsto \_)\right)}$

$ws \Rightarrow \boxed{Ws \Rightarrow \mathbb{C}(Ac)_1 \leq Ac_1}$

```
<(t,g):=Ac>
```

```
<ta:=Ta>
```

$ta = A \wedge \boxed{Ta = A}$

$\exists T \geq 0, G \geq 0$

$sz = S \wedge tv = V \wedge t = T \wedge g = G \wedge \boxed{Ac = (T,G) \wedge \mathbb{U}(T < S \Rightarrow A + T \mapsto \_)}$

$\Rightarrow$ (stability)

$sz = S \wedge tv = V \wedge t = T \wedge g = G \wedge \boxed{Ac_1 \leq T \wedge Ac_2 = G \wedge \mathbb{U}(T < S \Rightarrow A + T \mapsto \_)}$

```
if t=sz
```

(then)

emp

```
expand()
goto 1
```

(else)

$sz = S \wedge tv = V \wedge t = T \wedge g = G \wedge T < S \wedge$

$\boxed{Ac_1 \leq T \wedge Ac_2 = G \wedge \mathbb{U}(A + T \mapsto \_)}$

```
if ws
```

(then)

$sz = S \wedge ta = A \wedge tv = V \wedge$

$t = T \wedge g = G \wedge T < S \wedge$

$\boxed{\mathbb{C}(Ac)_1 \leq Ac_1 \leq T \wedge Ac_2 = G \wedge \mathbb{U}(A + T \mapsto \_)}$

(else)

$sz = S \wedge ta = A \wedge tv = V \wedge t = T \wedge g = G \wedge T < S \wedge$

$\boxed{Ac_1 \leq T \wedge Ac_2 = G \wedge Ta = A \wedge \mathbb{U}(A + T \mapsto \_)}$

```
lwsync
```

$sz = S \wedge ta = A \wedge tv = V \wedge t = T \wedge g = G \wedge T < S \wedge$

$\boxed{\mathbb{C}(Ac)_1 \leq Ac_1 \leq T \wedge Ac_2 = G \wedge \mathbb{U}(Ta = A \wedge A + T) \mapsto \_}$

$sz = S \wedge ta = A \wedge tv = V \wedge t = T \wedge g = G \wedge T < S \wedge \boxed{Ta = A \wedge \mathbb{C}(Ac)_1 \leq Ac_1 \leq T \wedge Ac_2 = G \wedge \mathbb{U}(A + T \mapsto \_)}$

```
[ta+t]:=tv
```

```
ws:=true
```

$sz = S \wedge tv = V \wedge t = T \wedge g = G \wedge T < S \wedge \boxed{Ta = A \wedge Ac_2 = G \wedge A + T \mapsto V}$

```
lwsync
```

$sz = S \wedge tv = V \wedge t = T \wedge g = G \wedge T < S \wedge \mathbb{U}\boxed{Ac_2 = G \wedge Ta = A \wedge A + T \mapsto V}$

$ws \wedge \boxed{\mathbb{C}(Ac)_1 \leq Ac_1 \leq T}$

```
<Ac := (t+1,g+1)>
```

$tv = V \wedge \boxed{Ac = (T+1, G+1) \wedge \mathbb{U}(Ta = A \wedge A + T \mapsto V)}$

$\Rightarrow$ (stability)

$tv = V \wedge \boxed{Ac_1 \leq T + 1 \wedge Ac_2 = G + 1 \wedge \mathbb{U}(Ta = A \wedge A + T \mapsto V)}$

$ws \wedge \boxed{\mathbb{C}(Ac)_1 \leq Ac_1}$

```
return
```

Figure 10: Formal proof of LIFO put()

```
void expand() {
1:   a := new Tasks[2∗sz];
2a: i := 0;
2b: <ta := Ta>;
2c: if  i=sz then           // for i : 0 .. sz-1 do
2d:       goto 3a;
2e: ae := ta+i;
2f: tv := [ae];             // tv := Ta[i]
2g: ae := a+i;
2h: [ae] := tv;             // a[i] := tv
2i: i := i+1;
2j: goto 2c;                // od
3a: -- rw/w barrier (lwsync) --
3b: ws := true;
3c: <Ta := a>;
4: sz := 2∗sz;
}
```

Figure 11: LIFO expand() with single-access memory instructions and barrier

8 Persistence

The assignment $[ta + t] := tv$ in put() is immediately followed by a barrier. Its precondition is therefore persistent until the write is committed.

To make the assignment <Ac:=(t+1,g+1) at the end of put() with $t = T$, we need three conditions to persist.

- $\mathbb{C}(Ac)_1 \leq T$: persists until the assignment is committed.
- $\mathbb{U}(Ta + T \mapsto V)$: persists until possibly overwritten by [ta+t]:=tv in the next put(). If the assignment remains uncommitted at that point then either (a) there has been no other write to Ac in the owner and $[Ta + T]$ will not be overwritten or (b) there has been an intervening write to Ac by an intervening take() and the potential overwrite will be preceded by an lwsync.
- fixed values for $[Ta + i]_{i=0}^{T-1}$ since the last time $Ac_1 = T$: same argument as above – if they were fixed before the assignment (not yet shown in my proof, but watch this space) then they are fixed afterwards.

To make <Ta:=a> at the end of expand() we need fixed universal values of $[Ta + i]_{i=0}^{Ac_1-1}$. We begin with fixed universal values for $[Ta + i]_{i=0}^{Sz-1}$ and then overwrite only $[Ta + T]$ in a context where $\mathbb{U}(Ac_1 \leq T)$. We have persistence.

To make <Ac:=(t-1,g) with $t = T \wedge Ta = A$ at the end of take() we need fixed values for $[A + i]_{i=0}^{T-1}$ until the next barrier. Because of the assignment $ws := \mathrm{false}$ there will be a barrier either after the initialisation in expand(), which does not touch the array pointed to by A, or before the assignment to $[A + \_]$ in put(). We have persistence.

I'd love to make treatment of persistence more formal. Intraprocedural invariants might do it.

9 Coda

Wow! Didn't think I could do all that.

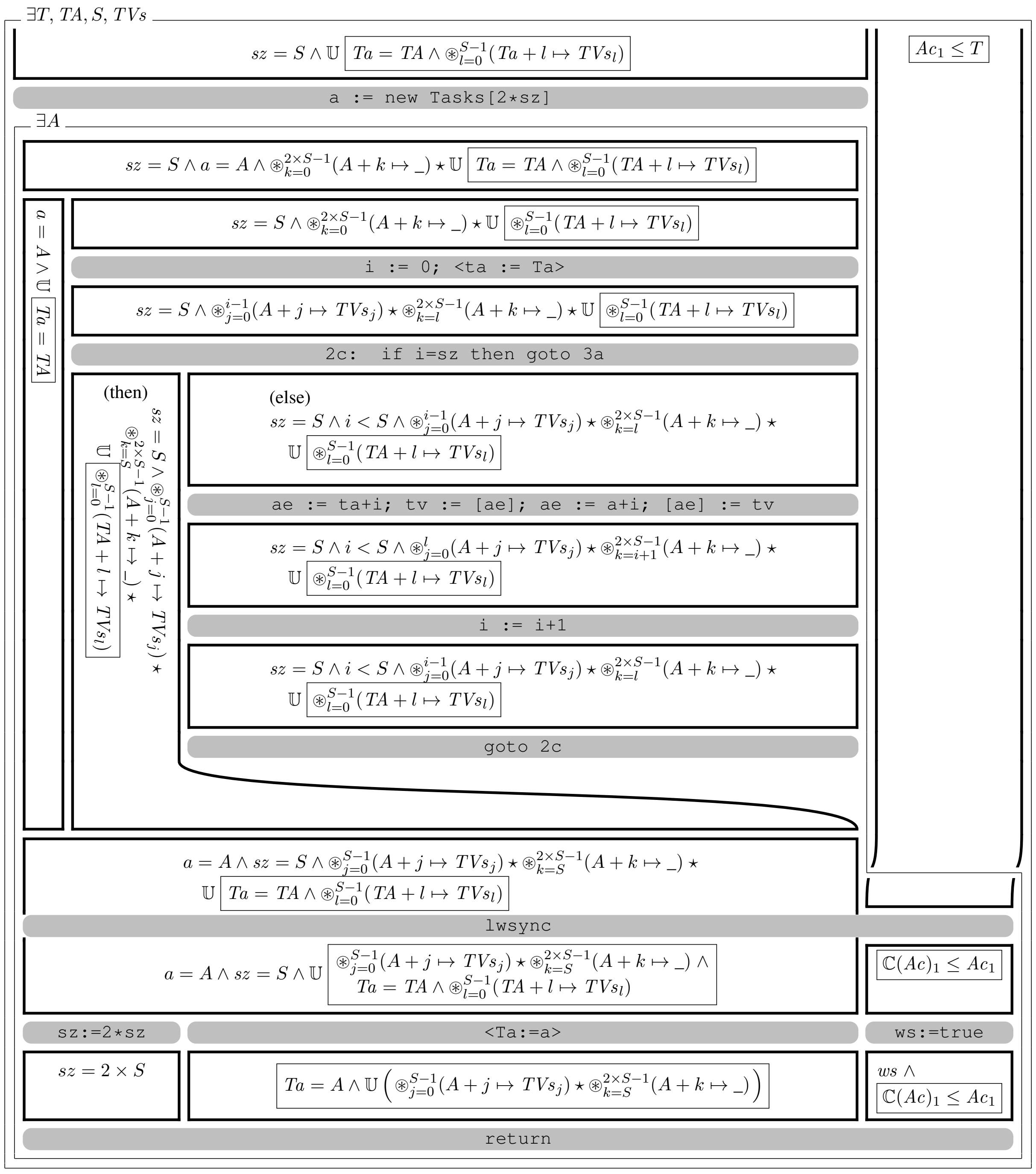
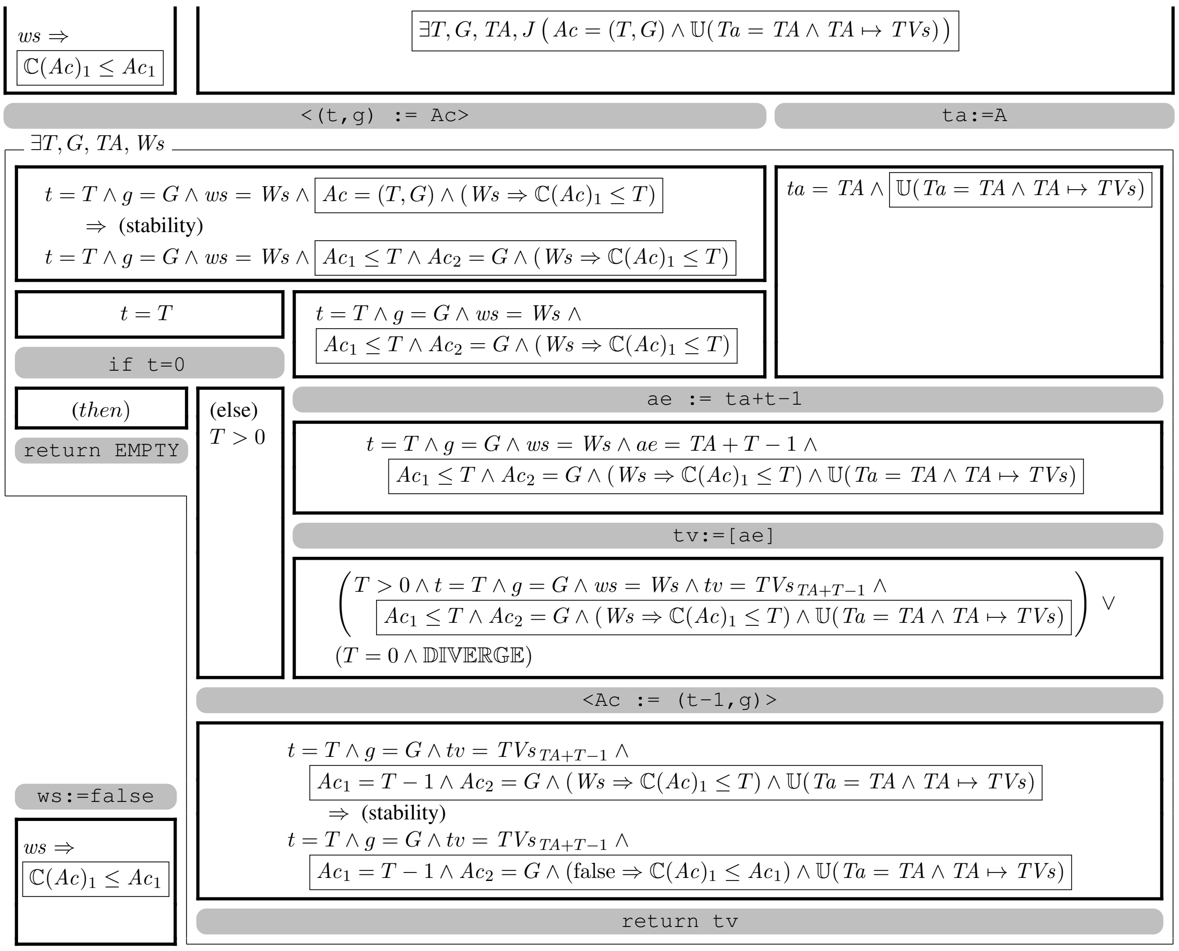

$\exists T, TA, S, TVs$
$sz = S \wedge \mathbb{U}\,\boxed{Ta = TA \wedge \circledast_{l=0}^{S-1}(Ta + l \mapsto TVs_l)}$
$\boxed{Ac_1 \leq T}$
a := new Tasks[2*sz]
$\exists A$
$sz = S \wedge a = A \wedge \circledast_{k=0}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{Ta = TA \wedge \circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
$a = A \wedge \mathbb{U}\,\boxed{Ta = TA}$
$sz = S \wedge \circledast_{k=0}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{\circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
i := 0; <ta := Ta>
$sz = S \wedge \circledast_{j=0}^{i-1}(A + j \mapsto TVs_j) \star \circledast_{k=l}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{\circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
2c: if i=sz then goto 3a
(then)
$sz = S \wedge \circledast_{j=0}^{S-1}(A + j \mapsto TVs_j) \star \circledast_{k=S}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{\circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
(else)
$sz = S \wedge i < S \wedge \circledast_{j=0}^{i-1}(A + j \mapsto TVs_j) \star \circledast_{k=l}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{\circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
ae := ta+i; tv := [ae]; ae := a+i; [ae] := tv
$sz = S \wedge i < S \wedge \circledast_{j=0}^{l}(A + j \mapsto TVs_j) \star \circledast_{k=i+1}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{\circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
i := i+1
$sz = S \wedge i < S \wedge \circledast_{j=0}^{i-1}(A + j \mapsto TVs_j) \star \circledast_{k=l}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{\circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
goto 2c
$a = A \wedge sz = S \wedge \circledast_{j=0}^{S-1}(A + j \mapsto TVs_j) \star \circledast_{k=S}^{2\times S-1}(A + k \mapsto \_) \star \mathbb{U}\,\boxed{Ta = TA \wedge \circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
lwsync
$a = A \wedge sz = S \wedge \mathbb{U}\,\boxed{\circledast_{j=0}^{S-1}(A + j \mapsto TVs_j) \star \circledast_{k=S}^{2\times S-1}(A + k \mapsto \_) \wedge Ta = TA \wedge \circledast_{l=0}^{S-1}(TA + l \mapsto TVs_l)}$
$\boxed{\mathbb{C}(Ac)_1 \leq Ac_1}$
sz:=2*sz
<Ta:=a>
ws:=true
$sz = 2 \times S$
$\boxed{Ta = A \wedge \mathbb{U}\left(\circledast_{j=0}^{S-1}(A + j \mapsto TVs_j) \star \circledast_{k=S}^{2\times S-1}(A + k \mapsto \_)\right)}$
$ws \wedge \boxed{\mathbb{C}(Ac)_1 \leq Ac_1}$
return

Figure 12: Formal proof of LIFO expand()

```
TaskInfo take() {
1:  <(t,g) := Ac>;
2a: if  t=0 then
2b:     return EMPTY;
3a: <ta := Ta>;
3b: ae := ta+t-1;
3c: tv := [ae];
4a: <Ac := (t-1,g)>;
4b: ws := false;
5:  return tv;
}
```

Figure 13: LIFO take() with single-memory-access instructions – no barriers needed



$ws \Rightarrow \boxed{\mathbb{C}(Ac)_1 \le Ac_1}$
$\boxed{\exists T, G, TA, J\,\big(Ac = (T,G) \wedge \mathbb{U}(Ta = TA \wedge TA \mapsto TVs)\big)}$
<(t,g) := Ac>
ta:=A
$\exists T, G, TA, Ws$
$t = T \wedge g = G \wedge ws = Ws \wedge \boxed{Ac = (T,G) \wedge (Ws \Rightarrow \mathbb{C}(Ac)_1 \le T)}$
$\Rightarrow$ (stability)
$t = T \wedge g = G \wedge ws = Ws \wedge \boxed{Ac_1 \le T \wedge Ac_2 = G \wedge (Ws \Rightarrow \mathbb{C}(Ac)_1 \le T)}$
$ta = TA \wedge \boxed{\mathbb{U}(Ta = TA \wedge TA \mapsto TVs)}$
$t = T$
if t=0
$t = T \wedge g = G \wedge ws = Ws \wedge$ $\boxed{Ac_1 \le T \wedge Ac_2 = G \wedge (Ws \Rightarrow \mathbb{C}(Ac)_1 \le T)}$
$(then)$
return EMPTY
(else) $T > 0$
ae := ta+t-1
$t = T \wedge g = G \wedge ws = Ws \wedge ae = TA + T - 1 \wedge$ $\boxed{Ac_1 \le T \wedge Ac_2 = G \wedge (Ws \Rightarrow \mathbb{C}(Ac)_1 \le T) \wedge \mathbb{U}(Ta = TA \wedge TA \mapsto TVs)}$
tv:=[ae]
$\left(\begin{array}{l} T > 0 \wedge t = T \wedge g = G \wedge ws = Ws \wedge tv = TVs_{TA+T-1} \wedge \\ \boxed{Ac_1 \le T \wedge Ac_2 = G \wedge (Ws \Rightarrow \mathbb{C}(Ac)_1 \le T) \wedge \mathbb{U}(Ta = TA \wedge TA \mapsto TVs)} \end{array}\right) \vee$
$(T = 0 \wedge \mathbb{DIVERGE})$
<Ac := (t-1,g)>
$t = T \wedge g = G \wedge tv = TVs_{TA+T-1} \wedge$ $\boxed{Ac_1 = T - 1 \wedge Ac_2 = G \wedge (Ws \Rightarrow \mathbb{C}(Ac)_1 \le T) \wedge \mathbb{U}(Ta = TA \wedge TA \mapsto TVs)}$
$\Rightarrow$ (stability)
$t = T \wedge g = G \wedge tv = TVs_{TA+T-1} \wedge$ $\boxed{Ac_1 = T - 1 \wedge Ac_2 = G \wedge (\text{false} \Rightarrow \mathbb{C}(Ac)_1 \le Ac_1) \wedge \mathbb{U}(Ta = TA \wedge TA \mapsto TVs)}$
ws:=false
$ws \Rightarrow \boxed{\mathbb{C}(Ac)_1 \le Ac_1}$
return tv

Figure 14: Formal proof of LIFO take()

References

Richard Bornat and Mike Dodds. Abducing memory barriers. submitted to PODC 2012, 2012. 1

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, 2009. 1, 2

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 175–186, 2011. 1

Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007. 1

Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, volume 4037 of *LNCS*, pages 256–271, August 2007. 1