AMP Exercises
Ch 3

Max Weiss

24 February 2020

Before working the exercises, let me summarize my understanding of the idea of a correctness condition for parallel execution, and then give definitions of the various correctness conditions in the text. Any program includes (if implicitly) a specification which distinguishes executions as correct or incorrect. However, that specification is in general defined only for executions which are sequential, in the sense that their method calls are totally ordered. It does not draw that distinction for the broader class of parallel executions. So there arises a question how systematically to generalize specifications to the parallel case; and this is where the idea of a "correctness condition" comes from.

How should such a generalization be constructed? The problem with questions of correctness of parallel execution is that they are essentially ambiguous. Sequential executions are systems of totally ordered method calls, whereas the calls for parallel executions are only partially ordered. A partial ordering can be extended to a total one in many ways; and in this context, some but not all extensions will be sequentially correct. Indeed, *a priori* it may not be obvious how to understand the idea that, when programs are parallelized, the ordering of instructions along a single thread ought fulfil the correctness conditions of the sequential execution.

Because a parallel execution H can be interpreted sequentially in many ways, it is natural to define a function which maps H to the set $\sigma(H)$ of all sequential interpretations of H which are in some sense "plausible". We might then apply the sequential program specification to the parallel execution H by requiring, for example, that at least one element of $\sigma(H)$ is sequentially correct.

In this way, correctness conditions can be naturally formulated by mapping parallel histories to sets of "plausible" sequential interpretations. Correctness conditions may then differ according to what they take to be plausible. (I suppose that they might also deploy the notion $\sigma(H)$ differently, for example by requiring sequential correctness of all or "most" of its elements, rather than just one.) Indeed, the three conditions stated in Chapter 3 of AMP follow this model.

Totality. It is plausible that an interpretation of a (parallel) history should at least account for everything that happens. This leads to one simple ingredient in various natural concepts of sequentialization:

> A sequentialization of H is *total* for H if it contains every event in H.

Atomicity. Atomicity is the idea that method calls should "appear to happen one at a time", rather than intermingling their effects. More precisely,

> A sequentialization S of events of H is *atomic* if it is a concatenation of invocation-response pairs u, v such that either v is the next matching response for u in H, or u has no next matching response in H and v is a newly introduced response to match u.

Atomicity (perhaps conjoined with totality) is a very weak condition, since it allows those calls to be arbitrarily reordered, provided only that executions and responses are not interleaved.

Quiescent consistency. In a parallel history, a single object x may undergo overlapping method calls, i.e., their invocations and responses may be interleaved to extend a "busy period" of x beyond the duration of a single call. By contrast, a point p in history is "quiet" or *quiescent* for x if no unmatched call of a method of x precedes it. Events u, v with $u < v$ can then be said to be *quiescently separated* if there is at least one point p such that $u \leq p < v$.

We might then require a sequential intepretation S to *respect quiescent separation*: in other words, if events u, v for x are quiescently separated in H, then u precedes v in S.

Finally we reach the first useful correctness condition in the AMP book:

> A total, atomic sequentialization of H is *quiescently consistent* if it respects quiescent separation.

Notice that in particularly "busy" histories, there may be little or no quiescence. In that case, quiescent consistency will become very weak, tolerating arbitrary reordering of matched invocation-response pairs.

Sequential consistency. The problem of evaluating parallel execution is precisely that it is not clear how events on different threads should be ordered. But what about for events which occur on the same thread? Quiescent consistency makes no mention of the thread of an event. As long as the process on a single thread A unfolds alongside some other busy threads, quiescent consistency will tolerate arbitrary atomic reorderings along A. It is not clear that this makes sense: perhaps all ambiguity of ordering ought to derive essentially from the parallelism, so that the sequential correctness condition applies directly to the ordering of events along any single thread.

Let's say that a sequentialization S of H is *threadwise faithful* if for any events u, v on the same thread, if $u < v$ in H then u precedes v in S. This leads to the second correctness condition.

> a total, atomic sequentialization S of H is *sequentially consistent* if it is threadwise faithful.

Sequential consistency certainly redresses the mentioned weakness of quiescent consistency. But as AMP authors note, it has its own problem: if they deposit their rent check on Monday, and you try, on another thread, to withdraw their rent the following Friday, sequential consistency would allow the bank to reorder those events.

Linearizability. Linearizability combines the ideas of quiescent and sequential consistency. Roughly speaking, the idea is to accept any sequentialization which treats each method call as happening instantaneously sometime between its invocation and its response. The one subtlety here seems to be (I think in contrast with the previous conditions) that a linearization may simply ignore pending method calls. Let's say that S *weakly total* if S contains every matched pair of invocations and responses from H. And, let's say that a sequentialization S of H *respects method call order* if whenever method call m_0 returns in H before method call m_1 is invoked, then the same is true in S.

> A weakly total, atomic sequentialization S of H is a *linearization* if it is threadwise faithful and it respects method call order.

21. A correctness property P is *compositional* provided that a history H satisfies P whenever every history $H|x$ of an object x of H satisfies P.

I'll now argue that quiescent consistency is compositional.

Suppose that $H|x$ is QC for all objects x of H. Then each $H|x$ has a corresponding sequence S_x which satisfies (A) and (B). Let S be the concatenation of all sequences S_x. Then S is a sequence of just method invocations and responses in H. It's clear that S satisfies (A). Since any invocation in $H|x$ can be matched only by any invocation in $H|x$, any state which is quiescent in H must be quiescent in $H|x$. So any quiescently separated u, v for x in H with $u < v$ must also be quiescently separated in $H|x$. So, u must occur later than v in S_x, and must therefore also occur later than v in S. Therefore (B) holds as well.

22. Let (r, s) be a memory object encompassing two register components r, s. And suppose that (r, s) is quiescently consistent. Must each of r, s be quiescently consistent?

Let's suppose that each of the three memory objects has methods to read and to write. The read method of the global object calls the read methods of each inner object successively, checks that the result is the same, and returns it if so. The write method of the global object successively calls, with the same argument, the write methods of each inner object.

In this case, some method calls of the global event may overlap even if no calls of the inner event overlap. Then, there may be quiescently consistent orderings of the global history which satisfy the specification, even though there are no correct QC orderings for some inner object.

For example,

1. invoke (r,s).write(1)
2. invoke (r,s).read()
3. return (r,s).write(1)
4. invoke r.read()
5. return null from r.read()
6. invoke s.read()
7. return null from s.read()
8. invoke (r,s).write(2)
9. return null from (r,s).read()
10. return (r,s).write(2)
11. invoke (r,s).read()
12. return 1 from (r,s).read()

23. Show that quiescent and sequential consistency are incomparable.

Here is a sample history to show quiescent $\not\Rightarrow$ sequential:

1. A: invoke x.write(1)
2. B: invoke x.read()
3. A: return x.write(1)
4. A: invoke x.write(2)
5. B: return null from x.read()
6. A: return x.write(2)
7. A: invoke x.read()
8. B: return 1 from x.read()

Here is a sample history to show sequential $\not\Leftarrow$ quiescent:

1. A: invoke x.write(1)

2. A: return x.write(1)
3. B: invoke x.write(2)
4. B: return x.write(2)
5. A: invoke x.read()
6. A: return 1 from x.read()

24a. The history in figure 3.13 is QC, SC, and linearizable. Since the first three calls happen on different threads, sequential consistency tolerates any ordering of them. Also, there is an interval in which they are all pending. So, no quiescent period separates them, and QC tolerates any ordering of them as well. Likewise, linearization points may be chosen from that interval to justify any linearizably tolerated of those three. The only thing that must happen (and indeed it must, according to all three definitions) is that the last call has to come after the first three. So all three correctness conditions are witnessed by the following sequentialization:

1. B: invoke and return from r.write(1)
2. A: invoke and return 1 from r.read()
3. C: invoke and return r.write(2)
4. B: invoke and return 2 from r.read()

24b. For pretty much the same reason as above, the history in figure 3.14 satisfies all three correctness conditions because they are witnessed by the following sequentialization:

1. C: invoke and return from r.write(2)
2. B: invoke and return from r.write(1)
3. A: invoke and return 1 from r.read()
4. B: invoke and return 1 from r.read()

25. Given how I've explicated linearizability in the summary above, sequential consistency is equivalent to the result of dropping the method call order constraint from the linearizability condition. But, I find the text to be pretty unclear about details and so I'm not sure if this is what they intend.

26. Suppose that H is linearizable, and let x be an object of H. Since H is linearizable, it has a total atomic sequentialization S which is threadwise faithful and respects method call order. Let S_x be the result of dropping from S all calls of methods of objects other than x. It is clear that S_x is total, atomic, and threadwise faithful and that it respects method call order with respect to

$H|x$ (all these notions are defined with purely universal quantifiers so they are preserved under substructures).

27. Suppose that first, thread A initializes the queue q, and then successively enqueues the integers 1, 2, and 3. After this, threads A and B dequeue on q simultaneously, so that gets the return value 1. Finally, A dequeues again, receiving the return value 3. Even a weakly total sequentialization of this history cannot be correct.

28. The reader method could issue a call to divide by zero. The reason is that the Java memory model is not sequentially consistent. Within the write method, a compiler optimization might reorder the assignments to x and v. Then, the read method may divide by zero if it is invoked after the assignment to v but before the assignment to x.

29. Amp says that a method is wait-free if it "guarantees that every call finishes its execution in a finite number of steps." What I think this means is that the method guarantees that its execution will not take an infinite number of steps.

Now consider property P of x: for every infinite history H of x, every thread that takes an infinite number of steps in H completes an infinite number of method calls.

I'll argue that Px is equivalent to wait-freedom of x.

In one direction, suppose that thread A takes an infinite number of steps in H, but that A completes only a finite number of method calls. Since the steps taken in A are partitioned into the method call executions, some method call execution must contain an infinite number of steps. In that case, the object cannot be wait-free. So wait-freedom does imply P.

Conversely, suppose that the execution of some method call on thread A takes an infinite number of steps. In that history, no further method calls can be completed, so the number of completed calls is finite.

30. An method is lock-free if it guarantees that infinitely often at least one of its calls finishes in a finite number of steps.

Is the property P equivalent to lock-freedom? No, because lock-freedom does not imply P. Lock-freedom allows that thread A executes the method infinitely often, while some execution of the same method on thread B cycles infinitely. In this case, the underlying object does not satisfy P.

Conversely, of course wait-freedom implies lock-freedom, so (by the previous exercise) P imples lock-freedom too.

31. Suppose that in every history, the ith time a thread calls method m, the call returns after $2i$ steps. Is this method wait-free, bounded wait-free, or neither? The method is wait-free, because after infinitely many steps, it will have been completed infinitely many times. But it is not bounded wait-free, because for any k, some execution (and every subsequent one) will take more than k steps.