A rational reconstruction of Harris's algorithm in RGSep (work in progress)

Richard Bornat
School of Engineering and Information Sciences, Middlesex University, UK

October 20, 2008

Abstract

Harris's algorithm so it doesn't look like magic.

1 Introduction (not yet ready for publication)

Refinement (**?**) is a process of taking an abstract but obviously adequate version of an algorithm and perturbing it, preserving its adequacy, towards a more implementable or more efficient version. The aim is to reduce abstraction, non-determinism and atomicity. 'Logical' refinement, in which the starting point includes a specification of the properties of a solution which is satisfied by the first and each of the refined versions, is frequently contrasted with 'ad-hoc' programming by invention. Refinement produces programs which are 'correct by construction'; invention produces programs which must be verified, often with great difficulty.

This claim doesn't quite ring true. For many programmers (including me) the abstract descriptions and specifications with which logical refinement starts are miracles of invention quite beyond our reach. We may not program 'logically' in that sense, but we are not irrational. It seems to me that our process is more dialectical than logical, embracing contradictions exposed by our early attempts and incorporating them creatively so as to advance towards a solution. In this our activity has echoes of the developments described by Lakatos (1976), though it would be too much to claim that we follow such an elegant path.

Dijkstra (1965) defined the first classic concurrency problem, that of two processes/threads communicating through a single shared 'buffer' variable, one writing, the other reading. The difficulty was and is that writing and reading aren't instantaneous operations, and if their executions overlap in time, the reader may extract a value from the buffer which is made up of parts of two or more successively-written values. The solution that Dijkstra put forward, synchronisation via semaphore instructions to ensure turn-taking, is nowadays challenged by so-called *non-blocking* algorithms in which the processes do not synchronise but somehow manage to avoid overlapping reads and writes.

In (Bornat and Amjad, 2008b) there is a development by logical refinement of a version of Simpson's four-slot non-blocking buffer algorithm (Simpson, 1990) by refinement, a non-blocking wait-free (**?**) asynchronous buffer which doesn't use any synchronisation mechanism – no semaphores, no locks, no CAS instructions or the like. In (Bornat and Amjad, 2008a) there is a verification, not by refinement, of Harris's three-slot buffer algorithm (Harris, 2004), which is also wait-free and asynchronous. So far as I am aware, Harris's algorithm

$$
\begin{array}{l}
\mathsf{local}\ b = \mathit{null}\ \mathsf{in} \\
\left(\begin{array}{l}
\ldots \\
\mathit{write}(w)\ \hat{=}\ \langle b := w \rangle
\end{array}
\;\middle\|\;
\begin{array}{l}
\ldots \\
\mathit{read}(\,)\ \hat{=}\ \mathsf{local}\ y\ \mathsf{in} \\
\qquad\qquad \langle y := b \rangle; \\
\qquad\qquad \mathsf{return}\ y \\
\qquad\quad\ \ \mathsf{ni}
\end{array}\right)
\end{array}
$$

Figure 1: A simple buffer, with angle-bracketed atomic buffer accesses

has had no other formal treatment. I cannot see how to develop that algorithm by refinement. I present here a rational dialectical development.

2 The atomic single-slot algorithm

Following (Bornat and Amjad, 2008b), I begin with the idealised buffer algorithm in figure 1: two processes in parallel, one with a write procedure, the other with a read, sharing a single buffer variable b. The angle-bracketed buffer accesses execute atomically – instantaneously in principle, but for instructions whose execution takes time, interleaving of execution will suffice. In this algorithm writer and reader act independently: the writer may overwrite a value which the reader hasn't read and/or the reader may read the same value more than once. This kind of buffer can be used to connect a simple data measuring device like a thermometer to a data collector: the collector wants to know what the current temperature is, and doesn't mind inspecting the same reading more than once; if the temperature is changing rapidly then the collector may miss some values. If we can find a straightforward way to avoid the necessity for atomicity then such a connection can be very useful indeed.

Again following (Bornat and Amjad, 2008b) my treatment uses RGSep (Vafeiadis, 2007; Vafeiadis and Parkinson, 2007). Assertions are a separated conjunction of an assertion about local state and a (boxed) assertion about shared memory. The mechanism depends on making assertions which are *stable* (invariant) under the actions of other threads: local assertions describe local memory and are inaccessible to those other threads; shared-memory assertions must in principle be stable no matter what the environment (those other threads) might do. We list the *actions* of each thread – the instantaneous changes it can make to shared memory by its atomic operations – in order to assess stability. As in the original rely-guarantee formalism of **?**, on which RGSep is partly based, there is a set R of actions which the environment may at any instant undertake, and we may rely on the fact that it will never do anything outside R, and a set G of actions which the current thread can undertake, and we guarantee that we will never do anything to the shared memory which isn't in G.

The RGSep mechanism relies on a special treatment of atomic and non-atomic commands. An atomic command may access and modify shared memory, and its instantaneous effect must be summarised in some subset of G: since it can't be interrupted or interfered with we can consider all of the memory, both local and shared, described in its pre-condition and affect any of the memory in its postcondition. For clarity in what follows I consider the instantaneous effect of a single atomic command without necessarily writing stable pre and post-conditions. But in a sequence of atomic commands this won't do: intermediate assertions must be stable under the effect of actions from the rely set R, because the environment can intervene between the separate commands. This is so-called *mid stability*, and it's one choice among several, which I've taken just for the sake of clarity. I have otherwise treated atomic commands relatively informally: the arguments below can be

formalised, but I'm concerned first with clarity of presentation.

Non-atomic commands for the most part don't affect shared memory, and we can use normal Hoare logic on the local parts of their pre- and post-conditions. When they do affect shared memory, as in the final version of the developed algorithm below, we have to use special rules to deal with them.

When the writer's atomic buffer access from figure 1 is augmented with an update to an auxiliary variable ws which records the sequence written, the command $\langle b := w; ws := ws.w \rangle$ has the effect

$$\begin{array}{l} \left\{ w = W \star \boxed{b = B \wedge ws = WS} \right\} \\ \left\langle \begin{array}{l} \{w = W \wedge b = B \wedge ws = WS\} \\ \quad b := w \\ \{w = W \wedge b = W \wedge ws = WS\} \\ ws := ws.w \\ \{w = W \wedge b = W \wedge ws = WS.w\} \end{array} \right\rangle \\ \left\{ w = W \star \boxed{b = W \wedge ws = WS.W} \right\} \end{array} \tag{1}$$

(note that in calculating the effect we first combine local and shared assertions and separate them afterwards) which produces the action

$$b = B \wedge ws = WS \rightsquigarrow b = B' \wedge ws = WS.B' \tag{2}$$

When the reader's buffer access is similarly augmented to become $\langle y := b; rs := rs.y \rangle$ the action is

$$b = B \wedge rs = RS \rightsquigarrow b = B \wedge rs = RS.B \tag{3}$$

An invariant chosen to show the relationship between ws and rs is then

$$\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b \tag{4}$$

– rs 'destuttered' (with repetitions removed) is a subsequence of ws, and the value in the buffer b is the last element of ws. This is invariant under (2) (adding the value in b to ws maintains $ws_\Omega = b$ and doesn't change $\lfloor rs \rfloor \preccurlyeq ws$), and it's invariant under (3) (adding the last element of ws to rs may cause stuttering, but otherwise maintains $\lfloor rs \rfloor \preccurlyeq ws$).

Because the actions which manipulate shared memory are all atomic it's clear that we have what Simpson calls 'coherence' (the reader always gets a complete value from the buffer, never a value which is a composite of parts of more than one written value). It's clear also from (3) and (4) that we have what Simpson calls 'freshness': the value added to rs is always the last element written up to that instant.

3 An attempted refinement to two slots

In (Bornat and Amjad, 2008b) one possible refinement of figure 1 is figure 2. Buffer variable b is refined by the two-slot buffer array $c[2]$ and the variable l; the C notation $!l$ is used, for conciseness, rather than $1 - l$, to invert the value in the single-bit variable l so as to alternate writing between the two slots of c. The writer acts so that the latest value written is always in $c[l]$ and the reader reads from $c[l]$. With augmentation the writer's action is

$$l = L \wedge ws = WS \star c[!L] = B \rightsquigarrow l = !L \wedge ws = WS.B' \star c[!L] = B' \tag{5}$$

$$
\begin{array}{l}
\mathsf{local}\ c[2] = (null, null), l = 0\ \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \langle l := !l;\, c[l] := w\rangle & \quad \mathsf{local}\ y\ \mathsf{in} \\
 & \quad\quad \langle y := c[l]\rangle; \\
 & \quad\quad \mathsf{return}\ y \\
 & \quad \mathsf{ni}
\end{array}\right)
\end{array}
$$

Figure 2: Data refinement to two slots

$$
\begin{array}{l}
\mathsf{local}\ c[2] = (null, null), l = 0\ \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \mathsf{local}\ wt\ \mathsf{in} & \quad \mathsf{local}\ y, rt\ \mathsf{in} \\
\quad\quad \langle\!\langle wt := !l\rangle\!\rangle; & \quad\quad \langle\!\langle rt := l\rangle\!\rangle; \langle y := c[rt]\rangle; \\
\quad\quad \langle c[wt] := w\rangle; & \quad\quad \mathsf{return}\ y \\
\quad\quad \langle\!\langle l := wt\rangle\!\rangle & \quad \mathsf{ni} \\
\quad \mathsf{ni} &
\end{array}\right)
\end{array}
$$

Figure 3: Two-slot algorithm after attempted atomicity refinement

and the reader's is

$$l = L \wedge rs = RS \star c[L] = B \leadsto l = L \wedge rs = RS.B \star c[L] = B \tag{6}$$

each of which preserves the refined invariant

$$\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l] \tag{7}$$

The algorithm of figure 2 still depends on atomicity, so it might seem that little has been gained in moving from one buffer slot to two. The desired gain is that the two processes can operate in different slots. If initially $l = L$ then the writer changes $c[!L]$ and the reader reads from $c[L]$.

The next step is to attempt an atomicity refinement, to give figure 3. The writer changes l atomically after it has written to $c[!l]$, in order not to disturb the invariant (7); the reader first takes a snapshot of the value of l and then uses it to access the buffer. Accesses to the shared single-bit index variable l are performed in double-angle-bracketed instructions which are 'naturally atomic' because each makes a single read or write access to l which we can expect would automatically be serialised by a computer store.[1] That means that we don't need any special hardware to make $\langle\!\langle wt := !l\rangle\!\rangle$ and $\langle\!\langle l := wt\rangle\!\rangle$ interleave with $\langle\!\langle rt := l\rangle\!\rangle$. For the time being we still have to rely on atomicity to interleave the presumably much larger accesses to the slots of c: as shown in figure 4, both processes can attempt to access the same slot at the same time.

Such 'collisions' are a problem, but they aren't the worst of this algorithm's difficulties. The major problem is that the invariant (7) isn't preserved. The reader can read out-of-sequence as illustrated in figure 5, seeing W' and then W, even though the writer writes W and then W'. So in figure 3 we no longer have $\lfloor rs \rfloor \preccurlyeq ws$.

[1] This isn't an unusual requirement: all non-blocking algorithms, so far as I am aware, depend upon a similar property, often for 'word-sized' values of 16, 32 or 64 bits. I have adhered in this discussion to Simpson's ascetic restriction only to depend on single-bit atomicity.

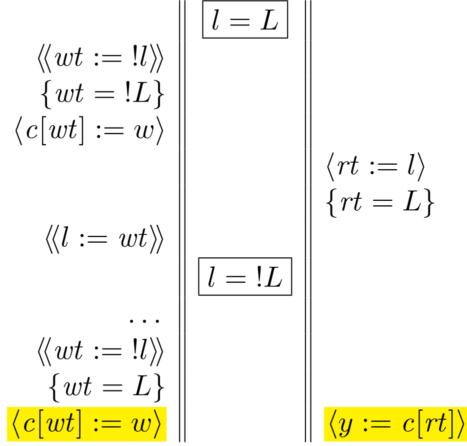


Figure 4: Collisions in refined two-slot algorithm

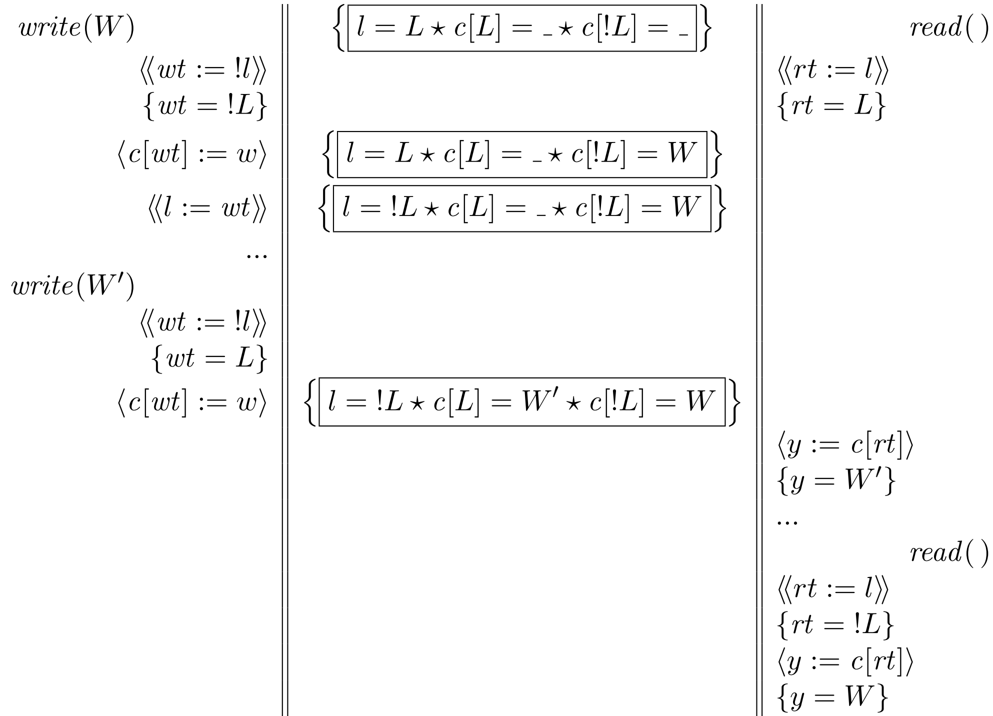


Figure 5: Reader sees values out of sequence

4 Identifying the problem

It's worth analysing the problem that has been exposed in figure 5. To do so we need to look precisely at the actions of either side. The augmented writer has the effect

$$\begin{array}{l}
\left\{ w = W \star \boxed{l = L \wedge ws = WS \star c[!L] = B} \right\} \\
\quad \langle\!\langle wt := !l \rangle\!\rangle \\
\left\{ w = W \wedge wt = !L \star \boxed{l = L \wedge ws = WS \star c[!L] = B} \right\} \\
\quad \langle c[wt] := w \rangle \\
\left\{ w = W \wedge wt = !L \star \boxed{l = L \wedge ws = WS \star c[!L] = W} \right\} \\
\quad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle \\
\left\{ w = W \wedge wt = !L \star \boxed{l = !L \wedge ws = WS.W \star c[!L] = W} \right\}
\end{array} \tag{8}$$

(all of the shared assertions are stable under the reader's actions, because the reader never alters l or ws or c or the slots of c). The first command doesn't affect shared memory, so has the identity action; the other two generate the actions

$$l = L \star c[!L] = B \rightsquigarrow l = L \star c[!L] = B' \tag{9}$$

$$l = L \wedge ws = WS \star c[!L] = B \rightsquigarrow l = !L \wedge ws = WS.B \star c[!L] = B \tag{10}$$

Action (9) has no effect on the invariant (7); action (10), by altering l and modifying ws accordingly, preserves the invariant. That half of the atomicity refinement has had the desired effect.

The modified reader has the effect

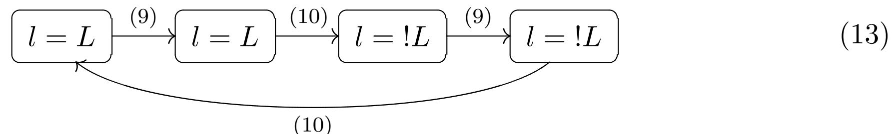
$$\begin{array}{l}
\left\{ \boxed{l = L \wedge rs = RS \star c[L] = B} \right\} \\
\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\left\{ rt = L \star \boxed{l = L \wedge rs = RS \star c[L] = B} \right\} \;\Rightarrow\; \text{(stability)} \\
\left\{ rt = L \star \boxed{\exists L', B' (l = L' \wedge rs = RS \star c[L] = B')} \right\} \\
\quad \langle y := c[rt]; rs := rs.y \rangle \\
\left\{ \exists B' \left(rt = L \wedge y = B' \star \boxed{\exists L' (l = L' \wedge rs = RS.B' \star c[L] = B')} \right) \right\}
\end{array} \tag{11}$$

After $\langle\!\langle rt := l \rangle\!\rangle$ we have, instantaneously $rt = L = l \star c[L] = B$, but $l = L$ isn't stable under (10), and $c[L] = B$ isn't stable under (9), so to deal with the subsequent atomic buffer access we have to weaken that assertion, breaking the connection between rt and l. That makes it clear that the buffer access doesn't necessarily read $c[l]$, and the horrible mixup of figure 5 is now possible. After the last instruction we have instantaneously $y = c[rt] = rs_\Omega$, and that's not stable under (9) either, but because I'm working with mid stability I can show the instantaneous effect. That instruction does generate a coherent action

$$rs = RS \star c[L] = B \rightsquigarrow rs = RS.B \star c[L] = B \tag{12}$$

but it's clear that this need not preserve the invariant (7) because we don't necessarily have $l = L$ and therefore we can't claim that $c[L]$ is in ws, the cause of the problem in figure 5.

But this is still only a symptom of the problem: the source lies deeper. In the stability-by-weakening step in (11) we lose the link between l and rt because the writer may carry out action (10). If we look carefully at the possible actions of the writer after $\langle\!\langle rt := l \rangle\!\rangle$ we can see that they form a little state machine:



(13)

$$
\begin{array}{l}
\mathsf{local}\ c[2] = (null, null), l = 0, ok\ \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \mathsf{local}\ wt\ \mathsf{in} & \quad \mathsf{local}\ y, rt\ \mathsf{in} \\
\qquad \langle\!\langle wt := !l \rangle\!\rangle; & \qquad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\qquad c[wt] := w; & \qquad \langle\!\langle rt := l \rangle\!\rangle; \\
\qquad \langle\!\langle l := wt \rangle\!\rangle; & \qquad y := c[rt]; \\
\qquad \langle\!\langle ok := \mathrm{false} \rangle\!\rangle & \qquad \langle\!\langle rt := ok \rangle\!\rangle; \\
\quad \mathsf{ni} & \qquad \mathsf{return}(rt, y) \\
 & \quad \mathsf{ni}
\end{array}\right)
\end{array}
$$

Figure 6: A tentative two-slot algorithm with non-atomic buffer accesses

The leftmost action (9), when $l = L = rt$, affects only $c[!L]$ and thus not $c[rt]$. The next action (10), when $l = L = rt$, breaks the link between l and rt but it also updates ws to preserve the invariant. Action (9), when $l = !L = !rt$, is the nasty one: it affects $c[L]$ which is $c[rt]$, putting there a value which is not yet in ws and allowing the reader to read out of sequence.

State machines like (13) are an informal tool of description. They are easily formalised by adding an auxiliary variable or two to the mix. I stick with the informal because my aim is to clarify, not to demonstrate my formal skills.

5 Fixing the problem

In conventional refinement terms we've hit a brick wall: our attempt to produce an algorithm that exhibits coherence and freshness and communicates according to invariant (7) has failed. Thinking dialectically, on the other hand, we may have an opportunity: the thesis that this algorithm solves our problem in the way we anticipated is contradicted by the antithesis of figure 5, but an analysis of the way the writer interferes with the reader suggests a surprising synthesis. If we allow the possibility that incoherent transfers *may* occur, we can perhaps detect when they *do* occur, or if that's impossible, when they *might* occur.

Suppose there's a shared variable ok which the reader sets true when it starts, just before $\langle\!\langle rt := l \rangle\!\rangle$. Then if that variable were to be set false by the writer during action (9), when $l = !L = !rt$, we would detect the writer's dangerous action. But the writer can't do that: it can't see rt so it can't tell when action (9) is dangerous and when it's not. It could, though, mark action (10), the one which must precede (9): if that action doesn't happen then all is well; if it does happen then things *might* get nasty.

In fact that same communication does much more. If the reader were to find that it had finished $\langle y := c[rt] \rangle$ and ok was still true, then the writer couldn't have altered $c[rt]$, because it couldn't have passed the second action in the state machine of (13) and therefore couldn't have reached the third, dangerous, action. That means that there was no collision, and the buffer access didn't need to be atomic. That argument leads to the modified two-slot algorithm of figure 6, in which the reader returns a pair: if the first element of the pair is true then the second element is a valid reading.

To make use of this synthesis we have to modify the invariant. The sequence of values rs returned from calls of *read* is now a sequence of pairs. If $\widetilde{rs}$ is rs restricted to those pairs whose first component is true and then with the first component discarded, we might hope

that

$$\lfloor \widetilde{rs} \rfloor \preccurlyeq ws \wedge ws_\Omega = c[l] \tag{14}$$

To show that the writer preserves this invariant is straightforward once we introduce rules to deal with non-atomic commands that access shared memory. The assignment $c[wt] := w$ writes to the heap because it is shorthand for $[c + wt] := w$, so we must use the non-atomic heap write rule

$$\frac{P \star kE \mapsto X \text{ stable under } R \quad P \star kE \mapsto X \rightsquigarrow P \star kE \mapsto X' \subseteq G}{\left\{E = kE \wedge F = kF \star \boxed{P \star kE \mapsto _}\right\}\ [E] := F\ \left\{E = kE \wedge F = kF \star \boxed{P \star kE \mapsto kF}\right\}} \text{ non-atomic heap write} \tag{15}$$

In considering the effect of $[E] := F$ the formulae E and F should be stable under the rely R, and they can in principle be made up of a mixture of shared and local variables. For simplicity in defining the rule I've supposed that E and F mention only local variables, which makes them automatically stable. Then E has to point to a heap cell; shared P has to guarantee that the value of that cell isn't changed by the environment; and the action of changing the value of the cell – which might take place in several steps – has to be declared as part of the writer's guarantee.

In our example $[c + wt] := w$ the variable c isn't local but it's effectively constant, so we're safe there:

$$\begin{array}{l}
\left\{w = W \star \boxed{l = L \wedge ws = WS \star c[!L] = B}\right\} \\
\quad \langle\!\langle wt := !l \rangle\!\rangle \\
\left\{w = W \wedge wt = !L \star \boxed{l = L \wedge ws = WS \star c[!L] = B}\right\} \\
\qquad — \left\{\begin{array}{l} c[wt] = B \text{ stable under } R \\ c[wt] = B \rightsquigarrow c[wt] = B' \subseteq G \end{array}\right\} \\
\quad c[wt] := w \\
\left\{w = W \wedge wt = !L \star \boxed{l = L \wedge ws = WS \star c[!L] = W}\right\} \\
\quad \langle\!\langle l := wt;\ ws := ws.w \rangle\!\rangle \\
\left\{w = W \wedge wt = !L \star \boxed{l = !L \wedge ws = WS.W \star c[!L] = W}\right\} \\
\quad \langle\!\langle ok := \text{false} \rangle\!\rangle \\
\left\{w = W \wedge wt = !L \star \boxed{\neg ok \wedge l = !L \wedge ws = WS.W \star c[!L] = W}\right\}
\end{array} \tag{16}$$

This generates actions (9) and (10) as before, plus

$$ok \rightsquigarrow \neg ok \tag{17}$$

(the final instruction may generate either the identity action, if $\neg ok$ already, or this action). Clearly these three actions maintain the invariant (14): extending ws doesn't invalidate the subsequence relation, and ws is updated at the same instant as l is changed to preserve $ws_\Omega = c[l]$. The final value of ok is instantaneous, and we shall see that the reader may change it, but I can use it because I'm dealing in mid stability and I can overall give instantaneous pre and post conditions.

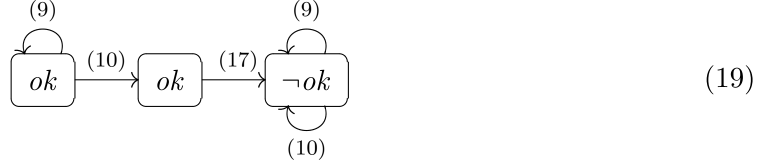
To deal with the reader we have to use a conditional heap read rule:

$$\frac{P \star kE \mapsto _,\ P \star (C \Rightarrow kE \mapsto kF),\ \neg C \text{ stable under } R}{\left(\begin{array}{l} \left\{y = _ \wedge E = kE \star \boxed{P \star kE \mapsto kF}\right\} \\ \quad y := [E] \\ \left\{\exists kF', kF'' \left(y = kF' \wedge E = kE \star \boxed{P \star kE \mapsto kF'' \wedge C \Rightarrow kF = kF'}\right)\right\} \end{array}\right)} \text{ non-atomic heap read} \tag{18}$$

$E = kE$ has to be stable (local will do for simplicity); it has to address a heap cell with contents kF initially; that heap cell has to exist throughout the read; there has to be some

C which guarantees that that cell contains kF; if C becomes false then it stays false. In the worst case the value kF' which is read into y is neither the original nor the final value in the heap cell, but if C still holds at the end of the read we can guarantee that y has the value kF. In our case C is just the formula ok.

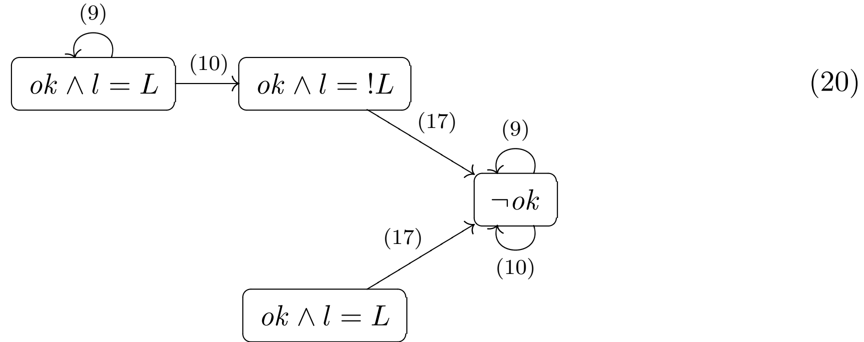
The reader's first instruction is $\langle\!\langle ok := \mathrm{true} \rangle\!\rangle$. The writer's actions after ok has been set can be described by a simple finite state machine:



(19)

Because of the way that stores work, non-atomic reads and writes are in practice implemented by a sequence of atomic reads and writes, so action (9) may need to happen several times to produce a complete buffer write. Once $\neg ok$ is set by (17), it stays set until the reader intervenes. In reasoning about the reader we don't care what happens as a result of the writer's actions after it has set $\neg ok$.

When I first drew the machine of (19) I labelled the first two boxes $ok \wedge l = L$, where L is the instantaneous value of l at $\langle\!\langle ok := \mathrm{true} \rangle\!\rangle$, and the third $ok \wedge l = !L$, thinking that it would be reached by the writer's action (10). But that was a mistake, as I found when I tried a verification. The finite state machine, when we take into account the value of l, has to be



(20)

– the writer can start in any of the states labelled $ok \wedge l = L$, and make any of the transitions shown. Before the reader executes its second instruction $\langle\!\langle rt := l \rangle\!\rangle$ the writer could have made any number of transitions. If $ok \wedge l = L$ still holds, we will have $rt = L$, and the reader will try to read the cell which was $c[l]$ when it started; if $ok \wedge l = !L$ it will try to read the cell which is $c[l]$ after action (10); either way, provided the read is completed before $\neg ok$ is set, it will get a value which is a valid extension of $\widetilde{rs}$.

That informal argument is just what is captured by a more laboured semi-formal (formal

apart from the treatment of stability) verification:

$$\left\{\boxed{l = L \wedge rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \star c[L] = B \star c[!L] = _}\right\}$$
$$\langle\!\langle ok := \text{true} \rangle\!\rangle$$
$$\left\{\boxed{ok \wedge l = L \wedge rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \star c[L] = B \star c[!L] = _}\right\} \Rightarrow \text{(stability)}$$
$$\left\{\exists C, C' \left(\boxed{\begin{array}{l} rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow (B = C \wedge l = !L \Rightarrow \lfloor \widetilde{rs}.B.C' \rfloor \preccurlyeq ws) \star \\ \qquad c[L] = C \star c[!L] = C' \end{array}}\right)\right\}$$
$$\langle\!\langle rt := l \rangle\!\rangle$$
$$\left\{\exists C, C', L' \left(\begin{array}{l} rt = L' \star \\ \boxed{\begin{array}{l} l = L' \wedge rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow (B = C \wedge l = !L \Rightarrow \lfloor \widetilde{rs}.B.C' \rfloor \preccurlyeq ws) \star \\ \qquad c[L] = C \star c[!L] = C' \end{array}} \end{array}\right)\right\} \Rightarrow \text{(stability)}$$
$$\left\{\exists C, C', L' \left(rt = L' \star \boxed{\begin{array}{l} rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow (B = C \wedge L' = !L \Rightarrow \lfloor \widetilde{rs}.B.C' \rfloor \preccurlyeq ws) \star \\ \qquad c[L] = C \star c[!L] = C' \end{array}}\right)\right\}$$

(21)

To begin we have, instantaneously, some values in l, rs and the slots of c, and we know from the invariant (14) that $c[L]$ is a valid extension to $\widetilde{rs}$. The instantaneous postcondition of $\langle\!\langle ok := \text{true} \rangle\!\rangle$ isn't stable under the writer's action, but we can see from (20) that so long as ok stays true, $c[L]$ won't be overwritten and if l changes value then $c[!L]$ will have become a valid extension to $\widetilde{rs}$. Next $\langle\!\langle rt := l \rangle\!\rangle$ reads the value of l, which of course need not be the value we started with, and stability weakening loses the connection between rt and l. All of this generates only a single non-identity action:

$$\neg ok \rightsquigarrow ok \tag{22}$$

To deal with the rest of the reader it's simplest to use the specification-disjunction rule of Floyd-Hoare logic and deal separately with the two cases $L' = L$ and $L' = !L$. First with $L' = L$:

$$\left\{\exists C \left(rt = L \star \boxed{rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge ok \Rightarrow B = C \star c[L] = C \star c[!L] = _}\right)\right\}$$
$$\text{— } \{c[L] = _, ok \Rightarrow c[L] = C, \neg ok \text{ all stable under } R\}$$
$$y := c[rt]$$
$$\left\{\exists C, D, D' \left(\begin{array}{l} rt = L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow B = C = D = D' \star c[L] = D' \star c[!L] = _ \end{array}} \end{array}\right)\right\} \Rightarrow$$
$$\left\{\exists D \left(rt = L \wedge y = D \star \boxed{\begin{array}{l} rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow B = D \star c \mapsto _, _ \end{array}}\right)\right\} \tag{23}$$
$$\langle\!\langle rt := ok; rs := rs.(rt, y) \rangle\!\rangle$$
$$\left\{\exists D \left(rt = ok \wedge y = D \star \boxed{\begin{array}{l} rs = RS.(ok, D) \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow B = D \star c \mapsto _, _ \end{array}}\right)\right\}$$

The side-condition $c[L] = _$ stable under R holds because the writer never deallocates a heap cell; $\neg ok$ is stable under R because writer action (17) sets $\neg ok$ and no other action

touches ok; $ok \Rightarrow c[L] = C$ is stable under R because when ok the writer's only action which writes to a slot of c is (9) with $l = L$, which affects $c[!L]$. Weakening throws away lots of confusion. Finally we extend rs with a pair which is either (true, B), where B was the value in $c[l]$ when we started, or (false, D), where D may not even be a coherent value or an extension to $\widetilde{rs}$. We've achieved coherence and freshness, provided that the first of the pair is true. This actually generates two actions

$$rs = RS \wedge \lfloor \widetilde{rs}.B \rfloor \preccurlyeq ws \rightsquigarrow rs = RS.(\text{true}, B) \tag{24}$$

$$rs = RS \rightsquigarrow rs = RS.(\text{false}, B) \tag{25}$$

each of which clearly preserves the invariant In the other case when $L' = !L$:

$$\begin{array}{l}
\left\{ \exists C' \left(rt = !L \star \boxed{rs = RS \wedge ok \Rightarrow \lfloor \widetilde{rs}.B.C' \rfloor \preccurlyeq ws \star c[L] = _ \star c[!L] = C'} \right) \right\} \\
\quad \text{— } \{ c[!L] = _,\ ok \Rightarrow c[!L] = C',\ \neg ok \text{ all stable under } R \} \\
\quad y := c[rt] \\
\left\{ \exists C', D, D' \left(\begin{array}{l} rt = !L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge ok \Rightarrow (C' = D = D' \wedge \lfloor \widetilde{rs}.B.C' \rfloor \preccurlyeq ws) \star \\ \quad c[L] = _ \star c[!L] = D' \end{array}} \end{array} \right) \right\} \Rightarrow \\
\left\{ \exists D \left(\begin{array}{l} rt = !L \wedge y = D \star \\ \boxed{rs = RS \wedge ok \Rightarrow \lfloor \widetilde{rs}.D \rfloor \preccurlyeq ws \star c \mapsto _, _} \end{array} \right) \right\} \\
\quad \langle\!\langle rt := ok; rs := rs.(rt, y) \rangle\!\rangle \\
\left\{ \exists D \left(\begin{array}{l} rt = L \wedge y = D \star \\ \boxed{rs = RS.(ok, D) \wedge ok \Rightarrow \lfloor \widetilde{rs}.D \rfloor \preccurlyeq ws \star c \mapsto _, _} \end{array} \right) \right\}
\end{array} \tag{26}$$

This generates exactly the same actions as (23). We have freshness – note that $\lfloor \widetilde{rs}.B.C' \rfloor \preccurlyeq ws$ guarantees that if we don't get the value B that was in $c[l]$ when we started, we will get C' which is later in the sequence and therefore even fresher – we have coherence, we have the invariant (14).

5.1 Is the problem really fixed?

I was confronted with a problem – sometimes the buffer might communicate values out of sequence – and a minor annoyance – despite using two buffer slots I couldn't use non-atomic reads and writes. Both problems were resolved by changing the specification of the reader so that it signals the possibility of incoherence and/or out-of-sequence values. Technically I've barred an exception to produce a synthesis of the original attempt and the contradictory problems. The algorithm I produced is wait-free: *read* and *write* operations are guaranteed to terminate in finite time no matter what the interference.

But all this is missing a rather large elephant in the room. A sequence of values all decorated with false is no use at all to the process which is trying to read from the buffer, and yet that is what might be generated if the writer works so fast that it trips up each read operation by setting $\neg ok$ before the reader has finished. And in any case, what could the process which invokes the reader do with a pair (false, B) except to ignore it and try again? Any

```
local c[2] = (null, null), d, l = 0, ok, ws = singleton(null), rs = empty in
( ...                               ||  ...
  write(w) =                        ||  read( ) =
    local wt in                     ||    local y, rt in
      ⟪wt := !l⟫;                   ||      ⟪ok := true⟫;
      c[wt] := w;                   ||      ⟪rt := l⟫;
      ⟪l := wt; ws := ws.w⟫;        ||      y := c[rt];
      ⟨d := w⟩;                     ||      ⟪rt := ok⟫;
      ⟪ok := false⟫                 ||      if ¬rt then ⟨y := d⟩ else skip fi;
    ni                              ||      ⟪rs := rs.y⟫;
                                    ||      return y
                                    ||    ni                              )
```

Figure 7: A two-slot algorithm with an atomically-accessed side channel

programmer would surely rather write a reader which keeps trying until it gets an answer:

```
read( ) =
  local y, rt in
    do
      ⟪ok := true⟫; ⟪rt := l⟫; y := c[rt]; ⟪rt := ok⟫
    while ¬rt;                                              (27)
    ⟪rs := rs.y⟫;
    return y
  ni
```

– it doesn't necessarily terminate, so it too won't necessarily ever deliver a result, so it has no operational advantage over figure 6, but it does feel more honest.

The problem isn't fixed – or rather, it isn't fixed very well. The synthesis has produced an algorithm and a specification which are only weakly useful. Time to think again.

6 Another attempt to fix the problem

In fixing the problem with a single-bit signalling variable ok I wasn't ambitious enough. The writer can do more than simply signal that nasty things might happen: it can first transmit the value it has just written in a side buffer and then signal; the signal would mean that the buffer can't be relied upon but the side-buffer can. Wary of collisions, I at first use atomic instructions for the side transmission, as in figure 7. (You might wonder why a one-slot side buffer will work if we need two slots for the main buffer: we shall see, eventually, that there is an answer to that question; for the moment I press on.)

It does also seem to work. If the reader executes $\langle y := d\rangle$ then it picks up a value that cannot be earlier than the $c[L]$ which it was guaranteed at the instant it executed $\langle\!\langle ok := \text{true}\rangle\!\rangle$. That's because it doesn't read from d unless $\neg ok$.

But there would be some difficulty proving that, because the writer repeatedly writes to d. In fact the reader is only interested in the first value that it writes, just before setting $\neg ok$. For once my instincts as a prover and as a programmer are in complete alignment: to save itself work the writer ought to write to d only if ok, just as the reader only reads from d if $\neg ok$. And then, surely, those accesses can be *non*-atomic, because they surely don't overlap. The result is figure 8, in which I have restrained my programming instincts and

local $c[2] = (null, null), d, l = 0, ok,$ $ws = singleton(null), rs = empty$ in

$$
\left(\begin{array}{l||l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \textsf{local}\ wt\ \textsf{in} & \quad \textsf{local}\ y, rt\ \textsf{in} \\
\quad\quad \langle\!\langle wt := !l \rangle\!\rangle; & \quad\quad \langle\!\langle ok := \mathrm{true} \rangle\!\rangle; \\
\quad\quad c[wt] := w; & \quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\quad\quad \langle\!\langle l := wt; ws := ws.w \rangle\!\rangle; & \quad\quad y := c[rt]; \\
\quad\quad \langle\!\langle wt := ok \rangle\!\rangle; & \quad\quad \langle\!\langle rt := ok \rangle\!\rangle; \\
\quad\quad \textsf{if}\ wt\ \textsf{then}\ d := w; \langle\!\langle ok := \mathrm{false} \rangle\!\rangle\ \textsf{else skip fi} & \quad\quad \textsf{if}\ \neg rt\ \textsf{then}\ y := d\ \textsf{else skip fi}; \\
\quad \textsf{ni} & \quad\quad \langle\!\langle rs := rs.y \rangle\!\rangle; \\
 & \quad\quad \textsf{return}\ y \\
 & \quad \textsf{ni}
\end{array}\right)
$$

Figure 8: Almost Harris's three-slot algorithm

not saved the writer work by setting $\neg ok$ after the reader has finished: that introduces a complication which we will confront later.

In programming terms the writer now writes d only when the reader seems to be asking for it – i.e. when ok. Verification of the writer is straightforward, and the first part is identical

to what has gone before:

$$
\begin{array}{l}
\left\{ w = W \star \boxed{l = L \wedge ws = WS \star c[!L] = B} \right\} \\
\quad \langle\langle wt := !l \rangle\rangle \\
\left\{ w = W \wedge wt = !L \star \boxed{l = L \wedge ws = WS \star c[!L] = B} \right\} \\
\qquad — \left\{ \begin{array}{l} c[wt] = B \text{ stable under } R \\ c[wt] = B \rightsquigarrow c[wt] = B' \subseteq G \end{array} \right\} \\
\quad c[wt] := w \\
\left\{ w = W \wedge wt = !L \star \boxed{l = L \wedge ws = WS \star c[!L] = W} \right\} \\
\quad \langle\langle l := wt; ws := ws.w \rangle\rangle \\
\left\{ w = W \wedge wt = !L \star \boxed{l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\quad \langle\langle wt := ok \rangle\rangle \\
\left\{ \begin{array}{l} w = W \wedge wt = OK \star \\ \quad \boxed{ok = OK \wedge l = !L \wedge ws = WS.W \star c[!L] = W} \end{array} \right\} \Rightarrow \text{(stability)} \\
\left\{ w = W \wedge wt = OK \star \boxed{OK \Rightarrow ok \wedge l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\quad \textsf{if } wt \textsf{ then} \\
\qquad \left\{ w = W \wedge wt \star \boxed{ok \wedge l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\qquad\quad — \left\{ \begin{array}{l} d = D \text{ stable under } R \\ ok \wedge d = D \rightsquigarrow ok \wedge d = D' \subseteq G \end{array} \right\} \\
\qquad d := w; \\
\qquad \left\{ w = W \wedge wt \star \boxed{d = W \wedge ok \wedge l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\qquad \langle\langle ok := \text{false} \rangle\rangle \\
\qquad \left\{ w = W \wedge wt \star \boxed{d = W \wedge \neg ok \wedge l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\quad \textsf{else} \\
\qquad \left\{ w = W \wedge \neg wt \star \boxed{l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\qquad \textsf{skip} \\
\qquad \left\{ w = W \wedge \neg wt \star \boxed{l = !L \wedge ws = WS.W \star c[!L] = W} \right\} \\
\quad \textsf{fi} \\
\left\{ w = W \wedge wt \star \boxed{l = !L \wedge ws = WS.W \star c[!L] = W} \right\}
\end{array} \tag{28}
$$

The actions of the writer are now

$$l = L \star c[!L] = B \rightsquigarrow l = L \star c[!L] = B' \tag{29}$$

$$l = L \wedge ws = WS \star c[!L] = B \rightsquigarrow l = !L \wedge ws = WS.B \star c[!L] = B \tag{30}$$

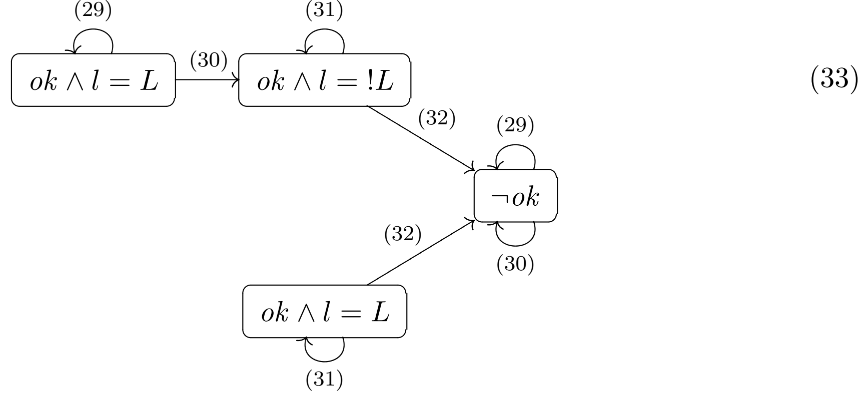
$$ok \wedge d = D \rightsquigarrow ok \wedge d = D' \tag{31}$$

$$ok \wedge l = L \star c[l] = d = B \rightsquigarrow \neg ok \wedge l = L \star c[l] = d = B \tag{32}$$

The first two of these we have seen before: they are (9) and (10). The third is new, recording the fact that the writer can write to d. The last is an expanded version of (17), noting that when the writer sets $\neg ok$ the value in d is the latest value written to the two-slot buffer and thus, by the invariant (7), the last element of ws.

From the point of view of the reader following $\langle\langle ok := \text{true} \rangle\rangle$ the writer's finite-state machine

is now



(33)

– the only way to set $\neg ok$ is via action (32), and after it has set $\neg ok$ the writer doesn't execute action (31) and so doesn't touch d.

Verification of the reader reveals that after $\langle\langle ok := \text{true}\rangle\rangle$ $c[L]$ is a valid extension to rs so long as ok holds, and when it stops holding, d is a valid extension instead.

$$
\begin{array}{l}
\left\{\boxed{l = L \wedge rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \star c[L] = B \star c[!L] = _}\right\} \\
\quad \langle\langle ok := \text{true}\rangle\rangle \\
\left\{\boxed{ok \wedge l = L \wedge rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \star c[L] = B \star c[!L] = _}\right\} \Rightarrow \text{(stability)} \\
\left\{ \exists C, C' \left(\boxed{\begin{array}{l} rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow (B = C \wedge l = !L \Rightarrow \lfloor rs.B.C' \rfloor \preccurlyeq ws) \star \\ \qquad c[L] = C \star c[!L] = C' \end{array}} \right) \right\} \\
\quad \langle\langle rt := l \rangle\rangle \\
\left\{ \exists C, C', L' \left(\begin{array}{l} rt = L' \star \\ \boxed{\begin{array}{l} l = L' \wedge rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow (B = C \wedge l = !L \Rightarrow \lfloor rs.B.C' \rfloor \preccurlyeq ws) \star \\ \qquad c[L] = C \star c[!L] = C' \end{array}} \end{array} \right) \right\} \Rightarrow \text{(stability)} \\
\left\{ \exists C, C', L' \left(rt = L' \star \boxed{\begin{array}{l} rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow (B = C \wedge L' = !L \Rightarrow \lfloor rs.B.C' \rfloor \preccurlyeq ws) \star \\ \qquad c[L] = C \star c[!L] = C' \end{array}} \right) \right\}
\end{array}
\tag{34}
$$

As before I analyse the non-atomic read in two cases, one where $L = L'$ and one where

$!L = L'$. First $L = L'$:

$$\left\{\exists C \left(rt = L \star \boxed{\begin{array}{l} rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ok \Rightarrow B = C \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c[L] = C \star c[!L] = _ \end{array}} \right)\right\}$$

$$— \ \{c[L] = _, ok \Rightarrow c[L] = C, \neg ok \text{ all stable under } R\}$$

$$y := c[rt]$$

$$\left\{\exists C, D, D' \left(\begin{array}{l} rt = L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge \\ \quad ok \Rightarrow B = C = D = D' \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c[L] = D' \star c[!L] = _ \end{array}} \end{array} \right)\right\} \Rightarrow \tag{35}$$

$$\left\{\exists D \left(\begin{array}{l} rt = L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge ok \Rightarrow \lfloor rs.D \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c \mapsto _, _ \end{array}} \end{array} \right)\right\}$$

Second $!L = L'$:

$$\left\{\exists C' \left(rt = !L \star \boxed{\begin{array}{l} rs = RS \wedge ok \Rightarrow \lfloor rs.B.C' \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c[L] = _ \star c[!L] = C' \end{array}} \right)\right\}$$

$$— \ \{c[!L] = _, ok \Rightarrow c[!L] = C', \neg ok \text{ all stable under } R\}$$

$$y := c[rt]$$

$$\left\{\exists C', D, D' \left(\begin{array}{l} rt = !L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge ok \Rightarrow (C' = D = D' \wedge \lfloor rs.B.C' \rfloor \preccurlyeq ws) \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c[L] = _ \star c[!L] = D' \end{array}} \end{array} \right)\right\} \Rightarrow \tag{36}$$

$$\left\{\exists D \left(\begin{array}{l} rt = !L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge ok \Rightarrow \lfloor rs.D \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c \mapsto _, _ \end{array}} \end{array} \right)\right\}$$

These two come together for the final part, reading from d if necessary:

$$\left\{\exists D\left(\begin{array}{l} rt = !L \wedge y = D \star \\ \boxed{\begin{array}{l} rs = RS \wedge ok \Rightarrow \lfloor rs.D \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c \mapsto _,_ \end{array}} \end{array}\right)\right\}$$

$\langle\!\langle rt := ok \rangle\!\rangle$

$$\left\{\exists D\left(\begin{array}{l} rt = OK \wedge y = D \star \\ \boxed{\begin{array}{l} ok = OK \wedge rs = RS \wedge ok \Rightarrow \lfloor rs.D \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c \mapsto _,_ \end{array}} \end{array}\right)\right\} \Rightarrow \text{(stability)}$$

$$\left\{\exists D\left(\begin{array}{l} rt = OK \wedge y = D \star \\ \boxed{\begin{array}{l} \neg OK \Rightarrow \neg ok \wedge rs = RS \wedge ok \Rightarrow \lfloor rs.D \rfloor \preccurlyeq ws \wedge \\ \quad \neg ok \Rightarrow \lfloor rs.B.d \rfloor \preccurlyeq ws \star c \mapsto _,_ \end{array}} \end{array}\right)\right\}$$

if $\neg rt$ then

$$\left\{\exists D, E\left(y = D \star \boxed{\neg ok \wedge rs = RS \wedge d = E \wedge \lfloor rs.d \rfloor \preccurlyeq ws \star c \mapsto _,_}\right)\right\} \tag{37}$$

— $\{\neg ok \wedge d = E \text{ stable under } R\}$

$y := d$

$$\left\{\exists E\left(y = E \star \boxed{\neg ok \wedge rs = RS \wedge d = E \wedge \lfloor rs.d \rfloor \preccurlyeq ws \star c \mapsto _,_}\right)\right\}$$

else

$$\left\{\exists D\left(y = D \star \boxed{\lfloor rs.D \rfloor \preccurlyeq ws \star c \mapsto _,_}\right)\right\}$$

skip $\left\{\exists D\left(y = D \star \boxed{rs = RS \wedge \lfloor rs.D \rfloor \preccurlyeq ws \star c \mapsto _,_}\right)\right\}$

fi

$$\left\{\exists F\left(y = F \star \boxed{rs = RS \wedge \lfloor rs.F \rfloor \preccurlyeq ws \star c \mapsto _,_}\right)\right\}$$

$\langle\!\langle rs := rs.y \rangle\!\rangle$;

$$\left\{\exists F\left(y = F \star \boxed{rs = RS.F \star c \mapsto _,_}\right)\right\}$$

The buffer read $y := d$ makes use of the simple non-atomic variable read rule

$$\frac{P \star y = Y \text{ stable under } R}{\left\{x = _ \star \boxed{P \star y = Y}\right\}\ x := y\ \left\{x = Y \star \boxed{P \star y = Y}\right\}} \text{ non-atomic variable read} \tag{38}$$

The reader's actions are

$$\neg ok \rightsquigarrow ok \tag{39}$$

$$rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \rightsquigarrow rs = RS.B \tag{40}$$

The first of these is (22) and the second is very similar to (24). The verification shows that the second is indeed an action of the reader, that after $y := d$ we do indeed have $\lfloor rs.y \rfloor \preccurlyeq ws$.

6.1 One last thing

In this development I was led to guard the assignment $d := w$ in the writer because I was looking for a way of making accesses to d non-atomic. Writing when ok, reading when $\neg ok$ does the job because the writer and reader don't set those values arbitrarily: the reader sets ok and the writer sets $\neg ok$.

But I would have done the same thing even without that consideration: the writer doesn't always have to write to d; it should only do so if the reader is currently active, i.e. if it has

$$
\begin{array}{l}
\mathsf{local}\ c[2] = (null, null), d, l = 0, ok, ws = singleton(null), rs = empty\ \mathsf{in} \\
\left(
\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \mathsf{local}\ wt\ \mathsf{in} & \quad \mathsf{local}\ y, rt\ \mathsf{in} \\
\quad\quad \langle\langle wt := !l \rangle\rangle; & \quad\quad \langle\langle ok := \mathrm{true} \rangle\rangle; \\
\quad\quad c[wt] := w; & \quad\quad \langle\langle rt := l \rangle\rangle; \\
\quad\quad \langle\langle l := wt; ws := ws.w \rangle\rangle; & \quad\quad y := c[rt]; \\
\quad\quad \langle\langle wt := ok \rangle\rangle; & \quad\quad \langle\langle rt := ok \rangle\rangle; \\
\quad\quad \mathsf{if}\ wt\ \mathsf{then}\ d := w; \langle\langle ok := \mathrm{false} \rangle\rangle\ \mathsf{else\ skip\ fi} & \quad\quad \mathsf{if}\ \neg rt\ \mathsf{then}\ y := d\ \mathsf{else} \langle\langle ok := \mathrm{false} \rangle\rangle\ \mathsf{fi}; \\
\quad \mathsf{ni} & \quad\quad \langle\langle rs := rs.y \rangle\rangle; \\
 & \quad\quad \mathsf{return}\ y \\
 & \quad \mathsf{ni}
\end{array}
\right)
\end{array}
$$

Figure 9: A version of Harris's three-slot algorithm

set ok. For the same reason, I want to make the reader set $\neg ok$ if it manages to successfully read from the two-slot buffer as in figure 9. To a programmer, that's obviously the right thing to do. But it breaks my proof, because it's no longer true that the writer writes to d only when ok: now the reader can set $\neg ok$ after the writer has decided to write and before it has finished. It would seem that we have to modify the writer's action (31) simply to read $d = D \rightsquigarrow ok \wedge d = D'$, which says that the writer can write to d at any time.

But surely, from the point of view of a passive reader at any point after $\langle\langle ok := \mathrm{true} \rangle\rangle$ the writer still behaves like the finite-state machine (33). Once the writer has set $\neg ok$ it won't write to d because to start writing to d it needs ok and while the reader is passive that won't happen.

To show the same thing formally needs, unfortunately, an auxiliary variable. Suppose the reader sets a variable ra (for reader active) when it sets ok and sets $\neg ra$ when it sets $\neg ok$. Then the writer writes d when $ok \vee \neg ra$. The rest is tedious detail, which is omitted.

Figure 9 is not exactly Harris's original. His algorithm writes d, if necessary, and sets $\neg ok$ at the *beginning* of the write procedure rather than at the end, which has the same effect since from the reader's point of view the writer endlessly cycles through its actions. It's unlikely that either version has an advantage: at least I can't see one.

7 Conclusion

Harris's algorithm is simple and surprising and for most people difficult at first to understand. A formal proof doesn't seem to illuminate much: indeed, by wrapping all the intricacies in a couple of implications and a lot of RGSep tedium it only seems to add to the difficulty. By showing a rational development I hope to have explained it to some more.

I would have developed it by conventional refinement techniques, but I couldn't see how. By making a dialectical development, by producing and fixing bugs, I hope to have corrected the misconception that that kind of programming is necessarily an irrational business.

References

Richard Bornat and Hasan Amjad. Inter-process Buffers in Separation Logic with Rely-Guarantee. Submitted to FAC, 2008a. URL `www.cs.mdx.ac.uk/staffpages/r_bornat/papers/rgsl1facs.pdf`. 1

Richard Bornat and Hasan Amjad. Simpson's algorithm by atomicity refinement. 2008b. URL www.cs.mdx.ac.uk/staffpages/r_bornat/papers/4slotrefinement.pdf. 1, 2, 3

P. Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002. 19

Edsger W. Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123, Technical University, Eindhoven, 1965. Reprinted in Genuys (1968) and Brinch Hansen (2002). 1

F. Genuys, editor. *Programming Languages*. Academic Press, 1968. 19

Tim Harris. A non-blocking three-slot buffer. Private communication, 2004. 1

Imre Lakatos. *Proofs and Refutations: the logic of mathematical discovery*. Cambridge University Press, 1976. 1

H.R. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137(1):17–30, January 1990. 1

Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007. 2

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. 2