Notes on Separation Logic for Shared-variable Concurrency

Peter O'Hearn

Queen Mary, University of London

January 3, 2002

1 Introduction

These notes record some ideas on adapting the storage separation logic of [9, 5, 7] to concurrency. We give obvious variants of the usual rules for conditional critical regions (CCRs) [2, 8], using $*$ to rule out interference through the heap. We could similarly use $*$ to adapt proof rules for monitors [3, 4, 1]. With the modified CCR rules we can at once handle many simple examples where a linked data structure rather than, say, an array is used to represent a resource. that itself holds pointer-free data.

Although the rules themselves are obvious, they have an interesting effect when the resource keeps track of pointers as part of its data, rather than just as part of its implementation. The $*$ connective allows us to partition memory in a dynamically reconfigurable way, extending the static partioning done by CCRs or monitors when there is no heap. This enables us to handle a number of subtler examples, where a pointer is transferred from one process to another, or between a process and a resource, and the ownership of the storage pointed to transfers with it. By "ownership" we mean the knowledge that the cell points to something, which in our formalism implies to the right to dereference or dispose without causing a race condition.

The examples below focus on this idea of ownership transfer. We begin with an idealized memory manager, which maintains a free list. When a program asks the manager for a new cell, an address is produced as expected. But more happens from a logical point of view: The knowledge that the cell points to something flows out of the manager and into the calling program. Further, when this knowledge transfers the manager is forced, by the proof rules, to forget: this is the sense in which the ownership transfers.

Our second example involves buffered communication, where one process sends a message to another, and where the message contains a pointer. This is typical of message passing in systems programs, and also of low-level implementations of higher-level message passing primitives. The essential safety property that must be guaranteed is that if the receiving process dereferences the pointer, then the sender must not; otherwise we will have a race condition. We account for this logically this by ensuring that the ownership of the passed pointer transfers with it.

Both examples would benefit from using variable-length segments [10]. The memory manager we treat only allows for fixed-size (binary) allocation and deallocation. It would be better, and probably not too much harder, to allow for variable-length allocation and deallocation. In the buffer example it would be more realistic to transfer a pointer to segment whose length was not fixed, to account for messages of different sizes.

Beyond what is presented here, there is an immediate need to have a proof of soundness of the rules; all of what follows is predicated on having one. Besides consistency, such a proof should show that verified programs don't have data races.

2 Proof Rules

To describe the proof rules we consider the basic case where there is a collection of resource declarations

$$\texttt{resource}\, r_1(\text{variable list}), ..., r_m(\text{variable list})$$

and a single concurrent composition

$$C_1 \parallel \cdots \parallel C_n.$$

We say that a variable *belongs to* resource r_i if it is in the associated variable list. As usual, we require that

- a variable belongs to at most one resource;
- if variable x belongs to resource r, it cannot appear in a parallel process except in a critical region for r; and
- if variable x is changed in one process, it cannot appear in another unless it belongs to a resource.

In the presence of pointers these syntactic restrictions are not enough to avoid interference, but with judicious placement of $*$ they are sufficient to enable simple proof rules. Although it is difficult for a compiler to rule out interference in pointer programs, our proof rules will ensure that interference is precluded in any program we verify. Syntactic restrictions control interference through variables, while $*$ controls interference through heap cells.

We assume that a formula RI_i, the resource invariant, has been chosen for each resource r_i. These formulae must satisfy

- all variables free in RI_i belong to resource r_i.

The rule for the region statement is

$$\frac{\{(P * RI_i) \wedge B\}\, C\, \{Q * RI_i\}}{\{P\}\, \texttt{region}\, r_i\, \texttt{when}\, B\, \texttt{do}\, C\, \texttt{endregion}\, \{Q\}} \quad \begin{array}{l}\text{No other process modifies} \\ \text{variables free in } P \text{ or } Q\end{array}$$

When inside a region statement the code gets to see the memory associated with the resource as well as that local to the process it is part of, while when outside a region statement it works without knowledge of the resource's state. Here, B is assumed to be heap independent.

The rule for $\|$ is

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \quad \cdots \quad \{P_n\}\, C_1\, \{Q_n\}}{\begin{array}{l}\{RI_1 * \cdots * RI_m * P_1 * \cdots * P_n\} \\ C_1 \parallel \cdots \parallel C_n \\ \{RI_1 * \cdots * RI_m * Q_1 * \cdots * Q_n\}\end{array}}$$

The side conditions on this rule are:

- no variable free in P_i or Q_i is changed in C_j when $j \neq i$.

Code that appears within a component of a parallel composition, but outside the body of a region statement, will be referred to as user process code. Code within a region statement allows a resource to be "accessed from" a user process.

Finally, for presentational convenience, in the following we use definitions of the form

$$\mathtt{name}(\boldsymbol{x}) \;\stackrel{\Delta}{=}\; \mathtt{region}\, r \;\mathtt{when}\; B \;\mathtt{do}\; C \;\mathtt{endregion}$$

to encapsulate operations on a resource. We emphasize that in this we are not introducing a procedure mechanism, but will merely use $\mathtt{name}(\boldsymbol{x})$ as an abbreviation.

3 A Memory Manager

As our first example we consider an idealized memory manager (mm) that doles out memory in chunks of size two. The manager maintains a free list, which is a singly-linked list of binary cons cells. The free list is pointed to by f, which is part of a resource declaration:

$$\mathtt{resource}\; mm(f).$$

The resource invariant is just that f points to a singly-linked list without any dangling pointers in the link fields:

$$RI: \quad \mathit{list}\ f.$$

The *list* predicate is the least satisfying the following recursive specification.

$$\mathit{list}\, x \;\stackrel{\Delta}{\Longleftrightarrow}\; (x = \mathtt{nil} \wedge \mathtt{emp}) \vee (\exists y.\, x \mapsto -, y \,*\, \mathit{list}\, y)$$

When a user program asks for a new cell, the mm gives it a pointer to the first element of the free list, if the list is nonempty. In case the list is empty the mm calls $\mathtt{cons}$ to get an extra element.

$$
\begin{array}{l}
\texttt{alloc}(x) \;\stackrel{\Delta}{=}\; \texttt{region}\; mm\; \texttt{when}\,\texttt{true}\,\texttt{do} \\
\qquad\qquad\qquad\qquad \texttt{if}\; f = \texttt{nil}\;\texttt{then}\; x := \texttt{cons}(-,-) \\
\qquad\qquad\qquad\qquad \texttt{else}\; x := f;\; f := [x+1]; \\
\texttt{endregion}
\end{array}
$$

The interaction between `alloc` and `cons` is a microscopic idealization of the treatment of `malloc` in Section 8.7 of [6]. There, `malloc` manages a free list but, occasionally, it calls a system routine `sbrk` to request additional memory to manage. Besides fixed versus variable sized allocation, the significant difference is that we will assume that `cons` always succeeds, while `sbrk` might fail (return an error code) if there is no extra memory to be given to `malloc`. This is a simplifying assumption on our part; it would be straightforward, though more tedious, to modify our account to allow for the possibility of failure when calling `cons`.

Here is a proof outline for the body of the region statement in `alloc`.

$$
\begin{array}{l}
\{\texttt{emp} * \mathit{list}\, f\} \\
\{\mathit{list}\, f\} \\
\;\texttt{if}\, f = \texttt{nil}\;\texttt{then} \\
\quad \{\mathit{list}\, f \;\wedge\; f = \texttt{nil}\} \\
\quad \{f = \texttt{nil} \;\wedge\; \texttt{emp}\} \\
\quad x := \texttt{cons}(-,-) \\
\quad \{(x \mapsto -,-) * (f = \texttt{nil} \;\wedge\; \texttt{emp})\} \\
\quad \{(x \mapsto -,-) * \mathit{list}\, f\} \\
\;\texttt{else} \\
\quad \{\mathit{list}\, f \;\wedge\; f \neq \texttt{nil}\} \\
\quad \{\exists y.\, f \mapsto -, y \;*\; \mathit{list}\, y\} \\
\quad x := f;\; f := [x+1]; \\
\quad \{(x \mapsto -,-) * \mathit{list}\, f\} \\
\{(x \mapsto -,-) * \mathit{list}\, f\}
\end{array}
$$

Since $\mathit{list}\, f$ is the resource invariant, a consequence of the proof rule for `region` statements is

$$\{\texttt{emp}\}\texttt{alloc}(x)\{x \mapsto -,-\}.$$

The proof of the body of `region` statement in `alloc` is on its own unremarkable. What is more interesting is the way that the property that the cell x points to something becomes detached from the resource invariant. This is what we mean by ownership transfer: the code for `alloc` transfers not only the value of x from the resource to a user program, but also the property that it points to something. And because of the use of $*$ to separate the storage used by parallel processes and resources, this knowledge is unique to the user process; that is the sense in which it is "owned".

Since we have used a critical region to protect the free list from corruption, it should be possible to have parallel processes that ask the mm to give them memory. The simplest example of this is just two processes, each of which uses a copy of the `alloc` code.

$$
\begin{array}{l}
\{\mathit{list}\, f\} \\
\{\mathit{list}\, f * \mathtt{emp} * \mathtt{emp}\} \\
\quad \mathtt{alloc}(x_1) \parallel \mathtt{alloc}(x_2) \\
\{\mathit{list}\, f * (x_1 \mapsto -,-) * (x_2 \mapsto -,-)\}
\end{array}
$$

This proof follows at once from the the property of $\mathtt{alloc}(x)$ established above, together with the proof rule for parallel composition.

We now do a similar treatment of deallocation. When a user program gives a cell back to the memory manager it is put on the front of the free list; there is no need for interaction with a system routine here.

$$
\begin{array}{l}
\mathtt{free}(y) \;\stackrel{\Delta}{=}\; \mathtt{region}\ \mathit{mm}\ \mathtt{when}\,\mathtt{true}\,\mathtt{do} \\
\qquad\qquad\qquad [y+1] := f; \\
\qquad\qquad\qquad f := y; \\
\qquad\qquad \mathtt{endregion}
\end{array}
$$

We can easily establish

$$
\begin{array}{l}
\{(y \mapsto -,-) * \mathit{list}\, f\} \\
[y+1] := f; \\
f := y; \\
\{\mathit{list}\, f\} \\
\{\mathtt{emp} * \mathit{list}\, f\}
\end{array}
$$

and the rule for region statements then gives us

$$\{y \mapsto -,-\}\mathtt{free}(y)\{\mathtt{emp}\}.$$

Using this and the rule for parallel composition we obtain

$$
\begin{array}{l}
\{\mathit{list}\, f * (y_1 \mapsto -,-) * (y_2 \mapsto -,-)\} \\
\quad \mathtt{free}(y_1) \parallel \mathtt{free}(y_2) \\
\{\mathit{list}\, f * \mathtt{emp} * \mathtt{emp}\} \\
\{\mathit{list}\, f\}
\end{array}
$$

We should comment on the form of the specifications of $\mathtt{alloc}(x)$ and $\mathtt{free}(y)$. These specifications do not mention the free list at all, even though their code uses it. What is happening is that effective resource separation allows us to present a localized view, where the state of a resource is hidden from user programs (when outside region statements). This localizing effect was already present in [2]; we have extended the treatment there to include the heap.

We finish this part with an example that is excluded by our proof rules. Consider a parallel composition of size one, which tries to corrupt the memory manager.

$$
\begin{array}{l}
\mathtt{alloc}(z); \\
\mathtt{free}(z); \\
[z+1] := z;
\end{array}
$$

This code makes use of knowledge of the way `free` works; after z is put back on the front of the free list, the program sneakily creates a cycle. It could equally set $[z+1]$ to some arbitrary value, say 42.

However, it is impossible to verify this code in our formalism with any precondition other than *false*. The reason is that the proof rule for region statements would require us to reason about the body of `free` as follows

$$
\begin{array}{l}
\{\mathit{list}\, f * z \mapsto -,-\} \\
[z+1] := f; \\
f := z; \\
\{\mathit{list}\, f * A * (z+1 \mapsto -)\}
\end{array}
$$

for some A, because if the statement $[z+1] := z$ is to verified, we must have $z{+}1 \mapsto -$ in its precondition (unless that precondition is *false*). But after $[z{+}1] := f$; $f := z$ the formula $\{\mathit{list}\, f) * (z+1 \mapsto -)$ cannot hold, since $f = z$.

The same principle applies with less malicious, but still erroneous, programs. A program might, through using a disposed pointer or simply through an address arithmetic error, corrupt the free list. Such an error would in principle show up in a failed verification attempt, since such corruption is ruled out by our formalism.

4 Buffered Transfer

In the *mm* example memory could transfer between a user process and a resource. Memory can also transfer from one user process to another as a result of targeted communications, such as when a message is sent from one process to another where the message contents contains a pointer.

Even when the programmer does not desire to send a pointer as the contents of a message, the underlying implementation of message passing often works by maintaining a queue buffer of pending messages, where each element in the buffer contains a memory address which identifies a data area. The contents of a message is to be found in the addressed data area; the queue does not itself contain the message contents. When the receiving process picks up the message and uses the embedded address to accesses the data, other processes cannot at the same time access the data area for the message, or we will have a race condition.

In this section we consider a pared-down version of this buffered ownership transfer, using a simple one-place buffer.

We assume a resource declaration, preceded by an an initialization of one of the resource variables.

$$
\begin{array}{l}
\mathit{full} := false; \\
\texttt{resource}\ \mathit{buf}(c, \mathit{full})
\end{array}
$$

We have standard code for putting a value into the buffer and for reading it out.

$$
\begin{array}{lll}
\texttt{put}(x) & \triangleq & \texttt{region}\ \mathit{buf}\ \texttt{when}\ \neg\mathit{full}\ \texttt{do} \\
& & \qquad c := x;\ \mathit{full} := \texttt{true} \\
& & \texttt{endregion}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{get}(y) & \triangleq & \texttt{region}\ \mathit{buf}\ \texttt{when}\ \mathit{full}\ \texttt{do} \\
& & \qquad y := c;\ \mathit{full} := \texttt{false} \\
& & \texttt{endregion};
\end{array}
$$

An extreme example of the necessity to account for ownership transfer is when the receiving process disposes the memory associated with an address it receives.

$$
\begin{array}{lcl}
x := \texttt{cons}(-); & \quad\|\quad & \texttt{get}(y); \\
\texttt{put}(x); & & \texttt{dispose}(y);
\end{array}
$$

This code creates a new pointer in one process, places the pointer in the buffer, and then the second process reads the pointer out and disposes it. To reason about the `dispose` operation in the second process, we must ensure that $y \mapsto -$ holds after $\texttt{get}(y)$ has finished. Notice that, since we have assumed that *full* has been initialized to *false* above, the code for $\texttt{get}(y)$ cannot proceed until $\texttt{put}(x)$ has been done.

To prove this program we use the following resource invariant:

$$
RI: \quad (\mathit{full} \wedge c \mapsto -) \vee (\neg\mathit{full} \wedge \texttt{emp}).
$$

The presence of `emp` in the right disjunct corresponds to the idea that when the buffer is empty it does not own the storage which might or might not be pointed to by c in the global state.

Here is a proof for the body of the region statement in $\texttt{get}(x)$.

$$
\begin{array}{l}
\{(RI * x \mapsto -) \wedge \neg\mathit{full}\} \\
\{(\neg\mathit{full} \wedge \texttt{emp}) * x \mapsto -\} \\
\{x \mapsto -\} \\
\ \ c := x;\ \mathit{full} := \texttt{true} \\
\{\mathit{full} \wedge c \mapsto -\} \\
\{RI\} \\
\{RI * \texttt{emp}\}
\end{array}
$$

The rule for region statements then gives us

$$
\{x \mapsto -\}\texttt{put}(x)\{\texttt{emp}\}.
$$

A crucial point in the proof of the body is the implication

$$
\mathit{full} \wedge c \mapsto - \ \Rightarrow\ RI,
$$

which is applied in the second-last step. This step reflects the idea that the knowledge "x points to something" flows out of the user program and into the resource. On exit from the critical region x does indeed point to something in the global state, but this information cannot be recorded in the postcondition of `put`. The reason is that we used $c \mapsto -$ to re-establish the resource invariant; having $x \mapsto -$ as the postcondition would be tantamount to asserting $(x \mapsto -) * (c \mapsto -)$ at the end of the body of the `region` statement, and this assertion is necessarily false when c and x are equal, as they are at that point.

The fact that `emp` is the postcondition of $\mathtt{put}(x)$ is what indicates that the first process has given up ownership of x. If we were to add code on the end of the process that dereferenced x then we could not use this postcondition to the `region` statement in conjunction with it.

The flipside of the first process giving up ownership is the second's assumption of it:

$$
\begin{array}{l}
\{(RI * \mathtt{emp}) \wedge \mathit{full}\} \\
\{\mathit{full} \wedge c \mapsto -\} \\
\quad y := c;\ \mathit{full} := \mathtt{false} \\
\{y \mapsto - \wedge \neg \mathit{full}\} \\
\{(\neg \mathit{full} \wedge \mathtt{emp}) * y \mapsto -\} \\
\{RI * y \mapsto -\},
\end{array}
$$

which gives us

$$\{\mathtt{emp}\}\mathtt{get}(y)\{y \mapsto -\}.$$

We can then prove the parallel processes as follows.

$$
\begin{array}{lcl}
 & \{RI\} & \\
 & \{RI * \mathtt{emp} * \mathtt{emp}\} & \\
\{\mathtt{emp}\} & & \{\mathtt{emp}\} \\
x := \mathtt{cons}(-); & \| & \mathtt{get}(y); \\
\{x \mapsto -\} & & \{y \mapsto -\} \\
\mathtt{put}(x); & & \mathtt{dispose}(y) \\
\{\mathtt{emp}\} & & \{\mathtt{emp}\} \\
 & \{RI * \mathtt{emp} * \mathtt{emp}\} & \\
 & \{RI\} &
\end{array}
$$

This annotated program pictures the local verifications for each of the two parallel processes, enclosed by the precondition and postcondition mandated by the rule for parallel composition, enclosed by precondition strengthening and postcondition weakening.

5 Combining the Buffer and the Memory Manager

We now show how to put the treatment of the buffer together with the homegrown memory manager *mm*, using `alloc` and `free` in place of `cons` and `dispose`. The point is to show different resources interacting in a modular way.

Let RI_{mm} denote the resource invariant *list f* for the memory manager. We do not have to change any of the code or verifications done with *mm*.

Neither do we have to change any of the code for the buffer. However, we change the resource invariant, replacing – by –,– to reflect the idea that we will transfer binary rather than unary cells.

$$RI_{buf}: \quad (\mathit{full} \wedge c \mapsto -,-) \vee (\neg \mathit{full} \wedge \texttt{emp}).$$

With the new resource invariant all of our work with `get` and `put` goes through with –,– in place of –, giving us

$$\{x \mapsto -,-\}\texttt{put}(x)\{\texttt{emp}\}$$

$$\{\texttt{emp}\}\texttt{get}(y)\{y \mapsto -,-\}.$$

It is irritating to have to formulate a new resource invariant here. If we had used address arithmetic, and accounted for transfer of variable-length segments, then we wouldn't have had to.

The general point, however, is that the `region` rule gives us properties of `free` and `alloc` that amount to "small axioms" [7]. This makes it easy to replace uses of `cons` and `dispose` with `free` and `alloc`, as long as we use $*$ to ensure that there is no interference with the memory manager's data structure.

For example,

$$
\begin{array}{lcl}
 & \{RI_{mm} * RI_{buf}\} & \\
 & \{RI_{mm} * RI_{buf} * \texttt{emp} * \texttt{emp}\} & \\
\{\texttt{emp}\} & & \{\texttt{emp}\} \\
\texttt{alloc}(x); & \| & \texttt{get}(y); \\
\{x \mapsto -,-\} & & \{y \mapsto -,-\} \\
\texttt{put}(x); & & \texttt{free}(y); \\
\{\texttt{emp}\} & & \{\texttt{emp}\} \\
 & \{RI_{mm} * RI_{buf} * \texttt{emp} * \texttt{emp}\} & \\
 & \{RI_{mm} * RI_{buf}\} &
\end{array}
$$

In this code, a pointer is first transferred out of the *mm* resource into the lefthand user process. It then gets sent into the *buf* resource, from where it taken out by the righthand process and promptly returned to *mm*.

6 More on the Buffer

This section deals with some of the finer points to do with the buffer example; namely, with how much memory is transferred with a pointer, and how much is present on termination.

6.1 Ownership is in the Eye of the Asserter

Transfer of ownership is not something that is determined operationally. Whether we transfer the storage associated with a single address, a segment, a linked list, or a tree depends on what we want to prove. Of course, what we can prove is constrained by code outside `region` statements, and in the program that used the buffer code

$$\begin{array}{lll} x := \texttt{cons}(-); & \| & \texttt{get}(y); \\ \texttt{put}(x); & & \texttt{dispose}(y); \end{array}$$

our hand was forced by the $\texttt{dispose}(y)$ statement in the second parallel process.

We have already seen this issue in the buffer example that used our home-grown memory management. There we transferred binary cons cells rather than unary cells, without canging the buffer code.

As an extreme case, if we dispose in the first rather than the second process then we have to ensure that no storage transfers with the pointer's value.

$$\begin{array}{lll} x := \texttt{cons}(-); & \| & \texttt{get}(y); \\ \texttt{put}(x); & & \\ \texttt{dispose}(x) & & \end{array}$$

We can prove this program, but to do so we need a different resource invariant.

$$RI: \quad (\mathit{full} \wedge \texttt{emp}) \vee (\neg \mathit{full} \wedge \texttt{emp}).$$

The use of `emp` in the left disjunct prevents the storage associated with x from flowing into the resource invariant.

This new resource invariant leads to different specifications of `get` and `put`.

$$\{x \mapsto -\}\texttt{put}(x)\{x \mapsto -\}$$
$$\{\texttt{emp}\}\texttt{get}(x)\{\texttt{emp}\}.$$

With the new resource invariant these specifications can be proven using the rule for `region` statements. We omit the details, and just give the proof figure for the parallel composition.

$$\begin{array}{l} \qquad\quad \{RI\} \\ \qquad\quad \{RI * \texttt{emp} * \texttt{emp}\} \\ \begin{array}{lll} \{\texttt{emp}\} & & \{\texttt{emp}\} \\ x := \texttt{cons}(-); & \| & \texttt{get}(y); \\ \{x \mapsto -\} & & \{\texttt{emp}\} \\ \texttt{put}(x); & & \\ \{x \mapsto -\} & & \\ \texttt{dispose}(x); & & \\ \{\texttt{emp}\} & & \end{array} \\ \qquad\quad \{RI * \texttt{emp} * \texttt{emp}\} \\ \qquad\quad \{RI\} \end{array}$$

We have seen that memory ownership can either transfer with a pointer's value, or stay located in the sending process, depending on what we want to prove. A final point is that it is not possible for the ownership to go both ways. For example, we cannot find a resource invariant that lets us prove

$$
\begin{array}{lcl}
x := \texttt{cons}(-); & \quad\| \quad & \texttt{get}(y); \\
\texttt{put}(x); & & \texttt{dispose}(y); \\
\texttt{dispose}(x) & &
\end{array}
$$

as is fortunate.

6.2 Avoiding Memory Leak using Auxiliary Variables

Returning to the end of Section 4, we proved there a weaker property than might have been expected. That is, we only obtained RI in the postcondition, and the resource invariant in that section was a disjunction, one of whose components included $c \mapsto -$. But, given the way the program works, we know the heap will be empty on termination; there is a small memory leak in the specification. To fix this, we would like to get `emp` in the postcondition.

The problem is that we have lost the information that the buffer is empty when we exit the critical region of the second process. And we cannot include $\neg \mathit{full}$ in the postcondition, because this would violate the variable restriction that is essential for the soundness of the `region` rule.

To handle the problem of memory leak, we first verify a more complex program that uses additional variables to record control information. Then we delete those variables [8] to infer a more exact property for the original program.

The new program has two additional variables, *start* and *finish*, and begins with the following initialization sequence and `resource` statement.

$$
\begin{array}{l}
\mathit{full} := false;\ \mathit{start} := \texttt{true};\ \mathit{finish} := \texttt{false} \\
\texttt{resource}\ \mathit{buf}(c, \mathit{full}, \mathit{start}, \mathit{finish}).
\end{array}
$$

We assign to the extra variables within the critical regions for buffer management.

$$
\begin{array}{lll}
\texttt{put}'(x) & \stackrel{\Delta}{=} & \texttt{region}\ \mathit{buf}\ \texttt{when}\ \neg\mathit{full}\ \texttt{do} \\
& & \qquad c := x;\ \mathit{full} := \texttt{true};\ \mathit{start} := \texttt{false} \\
& & \texttt{endregion}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{get}'(y) & \stackrel{\Delta}{=} & \texttt{region}\ \mathit{buf}\ \texttt{when}\ \mathit{full}\ \texttt{do} \\
& & \qquad y := c;\ \mathit{full} := \texttt{false};\ \mathit{finish} := \texttt{true} \\
& & \texttt{endregion};
\end{array}
$$

The new resource invariant is

$$
\begin{array}{rl}
& (\neg\mathit{full} \wedge \mathit{start} \wedge \neg\mathit{finish} \wedge \texttt{emp}) \\
RI: & \vee(\mathit{full} \wedge \neg\mathit{start} \wedge \neg\mathit{finish} \wedge c \mapsto -) \\
& \vee(\neg\mathit{full} \wedge \neg\mathit{start} \wedge \mathit{finish} \wedge \texttt{emp}).
\end{array}
$$

The three disjuncts correspond to what will be three control points: before `put'` begins; after `put'` ends and before `get'` begins; after `get'` ends.

Here is the reasoning for the body of the critical region in `put'`

$$\begin{array}{l}
\{(RI * (x \mapsto - \wedge \mathit{start})) \wedge \neg \mathit{full}\} \\
\{(\neg \mathit{full} \wedge \mathit{start} \wedge \neg \mathit{finish} \wedge \mathtt{emp}) * x \mapsto -\} \\
\quad c := x;\ \mathit{full} := \mathtt{true};\ \mathit{start} := \mathtt{false} \\
\{(\mathit{full} \wedge \neg \mathit{start} \wedge \neg \mathit{finish} \wedge \mathtt{emp}) * c \mapsto -\} \\
\{(\mathit{full} \wedge \neg \mathit{start} \wedge \neg \mathit{finish} \wedge c \mapsto -) * (\mathtt{emp} \wedge \neg \mathit{start})\} \\
\{RI * (\mathtt{emp} \wedge \neg \mathit{start})\}
\end{array}$$

and from the `region` rule we obtain

$$\{x \mapsto - \wedge \mathit{start}\}\mathtt{put}'(x)\{\mathtt{emp} \wedge \neg \mathit{start}\}.$$

For the critical region in `get'` we have

$$\begin{array}{l}
\{(RI * (\mathtt{emp} \wedge \neg \mathit{finish})) \wedge \mathit{full}\} \\
\{\mathit{full} \wedge \neg \mathit{start} \wedge \neg \mathit{finish} \wedge c \mapsto -\} \\
\quad y := c;\ \mathit{full} := \mathtt{false};\ \mathit{finish} := \mathtt{true} \\
\{\neg \mathit{full} \wedge \neg \mathit{start} \wedge \mathit{finish} \wedge y \mapsto -\} \\
\{(\neg \mathit{full} \wedge \neg \mathit{start} \wedge \mathit{finish} \wedge \mathtt{emp}) * (y \mapsto -) * (\mathtt{emp} \wedge \mathit{finish})\} \\
\{RI * (y \mapsto -) * (\mathtt{emp} \wedge \mathit{finish})\}
\end{array}$$

and thus

$$\{\mathtt{emp} \wedge \neg \mathit{finish}\}\mathtt{get}'(y)\{(y \mapsto -) * (\mathtt{emp} \wedge \mathit{finish})\}$$

Now, we can reason as follows

$$\begin{array}{lcl}
 & \{RI \wedge \mathit{start} \wedge \neg \mathit{finish}\} & \\
 & \{RI * (\mathtt{emp} \wedge \mathit{start}) * (\mathtt{emp} \wedge \neg \mathit{finish})\} & \\
\{\mathtt{emp} \wedge \mathit{start}\} & & \{\mathtt{emp} \wedge \neg \mathit{finish}\} \\
x := \mathtt{cons}(-); & \| & \mathtt{get}(y); \\
\{x \mapsto - \wedge \mathit{start}\} & & \{y \mapsto - * (\mathtt{emp} \wedge \neg \mathit{finish})\} \\
\mathtt{put}(x); & & \mathtt{dispose}(y) \\
\{\mathtt{emp} \wedge \neg \mathit{start}\} & & \{\mathtt{emp} \wedge \mathit{finish}\} \\
 & \{RI * (\mathtt{emp} \wedge \neg \mathit{start}) * (\mathtt{emp} \wedge \mathit{finish})\} & \\
 & \{RI \wedge \neg \mathit{start} \wedge \mathit{finish}\} & \\
 & \{\mathtt{emp}\} &
\end{array}$$

In the second-last formula, since $\neg \mathit{start} \wedge \mathit{finish}$ holds we know that the third of the disjuncts in RI must be true; this is what justifies the conclusion of `emp`.

Next, the initialization sequence establishes $\{RI \wedge \mathit{start} \wedge \neg \mathit{finish}\}$, so if we denote by `prog'` the initialization sequence and resource declaration of this section, followed by the parallel composition, we have shown

$$\{\mathtt{emp}\}\ \mathtt{prog'}\ \{\mathtt{emp}\}.$$

The variables *start* and *finish* occur only in statements of the form $\mathit{start} := E$ or $\mathit{finish} := E$. They thus do not affect the flow of control, and are auxiliary variables in the sense of [8]. Further, neither the pre nor postcondition of `prog'` mentions *start* or *finish*. Therefore, we can remove all statements involving these variables in `prog'`, as well as their positions in the resource declaration, and we obtain

$$\{\texttt{emp}\}\,\texttt{prog}\,\{\texttt{emp}\}.$$

The resulting program, `prog`, consists of the initialization, resource declaration and parallel statement from Section 4. We have finally proven the exact property we were after.

References

[1] G. Andrews. *Concurrent programming: principles and practice.* Benjamin/Cummings, 1991.

[2] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques.* Academic Press, 1972.

[3] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[4] J. H. Howard. Proving monitors. *Communications of the ACM*, 19(5):273–279, 1976.

[5] S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 36–49, London, January 2001.

[6] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice Hall, 1988. Second edition.

[7] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic: CSL 2001*, LNCS. Springer-Verlag, 2001. To appear.

[8] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM*, 19(5):279–285, 1976.

[9] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[10] J. C. Reynolds. Lectures on reasoning about shared mutable data structure. *IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic.* Tandil, Argentina, September 2000.