Online Appendix with an Additional Example for Subjective Auxiliary State for Coarse-Grained Concurrency

Ruy Ley-Wild Aleksandar Nanevski
IMDEA Software Institute
{ruy.leywild, aleks.nanevski}@imdea.org

1. Buffer Library and Producer/Consumer Client

To illustrate separate verification of a library and its client, we adapt another classic example due to Owicki and Gries. The library is a one-place buffer with blocking put and get operations.[1] The client is a pair of producer and consumer each with a private array of data that they concurrently communicate through the buffer. The producer iterates over its array, putting the elements into the buffer, one by one. The consumer populates its private array in order by getting from the buffer. The goal is to verify that at the end, the producer and consumer arrays have identical contents.

1.1 Buffer Library

The buffer library BufferLib (Figure 1, left) provides the functionality of a one-place buffer with operations $\mathsf{put}(v)$ for adding an element v to the buffer and $\mathsf{get}()$ for removing the contents of the buffer. Both operations block until they succeed: put loops until the buffer is empty so it can place v, and get loops until there's an element to remove.

Library's Parameters. To achieve separate verification of the library and client, we verify the library parametrized with respect to client-specific information for verification. For example, it is the client who decides on the type of values stored in the buffer, and the kind of auxiliary information that should be tracked by the shared resource. The buffer is oblivious to that kind of information; it just provides the basic functionality of inserting into and removing from the buffer. As with PCMs, dependent records suffice to describe the structure of information exchanged between the buffer and client.

The buffer has three parameters: the type A of values to be held in the buffer (i.e., an instance of parametric polymorphism), a PCM $\mathbb{U}$ for the auxiliary contributions the client may want to track with the buffer (i.e., an instance of ad hoc polymorphism), and a client-specific invariant ClientInv (i.e., an abstract predicate, generalizing [**?**, **?**]). Thus, BufferLib is a function of the above arguments, and produces a module that defines an invariant for the shared state and the put and get operations, which are themselves parametric in the client's auxiliaries.

Library's Invariant. The buffer owns two shared pointers full and cell protected by the lock lk. When the buffer is empty, the heap is described by $\mathsf{full} \mapsto \mathsf{false} * \mathsf{cell} \mapsto ()$. When the buffer is full with value v, the heap is described by $\mathsf{full} \mapsto \mathsf{true} * \mathsf{cell} \mapsto v$.

ClientInv extends the library's invariant with some constraint to relate the client's auxiliary contribution (of type $\mathbb{U}$) to the buffer's contents. In the interest of abstraction, it's desirable to hide the buffer's private heap from the client's invariant. Since the buffer holds at most one element v of type A, the buffer's contents are exposed with the algebraic representation of the type option A; it has two possible values: None for the empty buffer, and Some v for the full buffer with v. Thus, ClientInv is a predicate over a $\mathbb{U}$ value for the client's auxiliary contribution and an option A value that represents the buffer's contents.

The combined library/client invariant Inv takes an auxiliary contribution α argument, and produces a separation logic formula (i.e., a predicate over heaps). The invariant combines the library's representation invariant (that the shared heap satisfies the footprint for full and cell described above) and additionally requires the client invariant to hold of the auxiliary and the corresponding representation (as an option). Note the separation of concerns: the library's invariant is independent of the client's auxiliary, and the client's invariant is independent of the library's heap.

Library's Code. Just as the invariant Inv includes client-specific constraints, the put and get operations must also have parameters for the client to update its auxiliary contribution and relate its own pre- and postconditions to the invariant. That is, when the client calls a function func, it may want to pass an update function Φ to change its own auxiliary. Moreover, it may have a client-specific precondition ClientPre and postcondition ClientPost that it wants to relate to its invariant ClientInv. In CSL, the rule for the function can be given schematically as follows (in SCST, the specification can be given directly in the language):

$$\frac{\mathsf{ClientInv}\ \alpha\ repn \wedge \mathsf{ClientPre}\ \alpha \rightarrow \mathsf{ClientInv}\ (\Phi(\alpha))\ repn' \wedge \mathsf{ClientPost}\ (\Phi(\alpha))\ r}{\{\mathsf{ClientPre}\ \alpha\}\mathsf{func}(\langle\Phi\rangle)\{\mathsf{ClientPost}\ (\Phi(\alpha))\ r \wedge r = \cdots\}}$$

The library's function conceptually changes the representation from $repn$ to $repn'$ and when it completes, the client's update function Φ is used to change the client's contribution from α to $\Phi(\alpha)$. Thus it is necessary to show that the client's invariant holds of the updated buffer (i.e., with a new representation) and the client's new auxiliary. Thus, the client has the proof obligation in the premise: if the invariant holds of the old buffer representation ($\mathsf{ClientInv}\ \alpha\ repn$) and the precondition holds ($\mathsf{ClientPre}\ \alpha$), then after the update the invariant holds of the new buffer representation ($\mathsf{ClientInv}\ (\Phi(\alpha))\ repn'$) and the postcondition holds ($\mathsf{ClientPost}\ (\Phi(\alpha))\ r$, where r is the result).[2]

Thus, each function has parameters for the client's update function Φ, pre- and postconditions, and a proof of the pre/post/invariant implication. In particular, we first define the interfaces PutSpecs and GetSpecs to package the above information, and put and get functions take as an argument a module of the corresponding interface. Here we take advantage of the factthat the dependent type theory gives a unified language to package code and proofs. In both

[1] The original version had an N-place buffer, we use $N = 1$ for simplicity.

[2] A similar pattern appears in [**?**] for threading the client's *auxiliary code* through the library, but in that setting auxiliary state is global, and the considered language contains no modularity constructs.

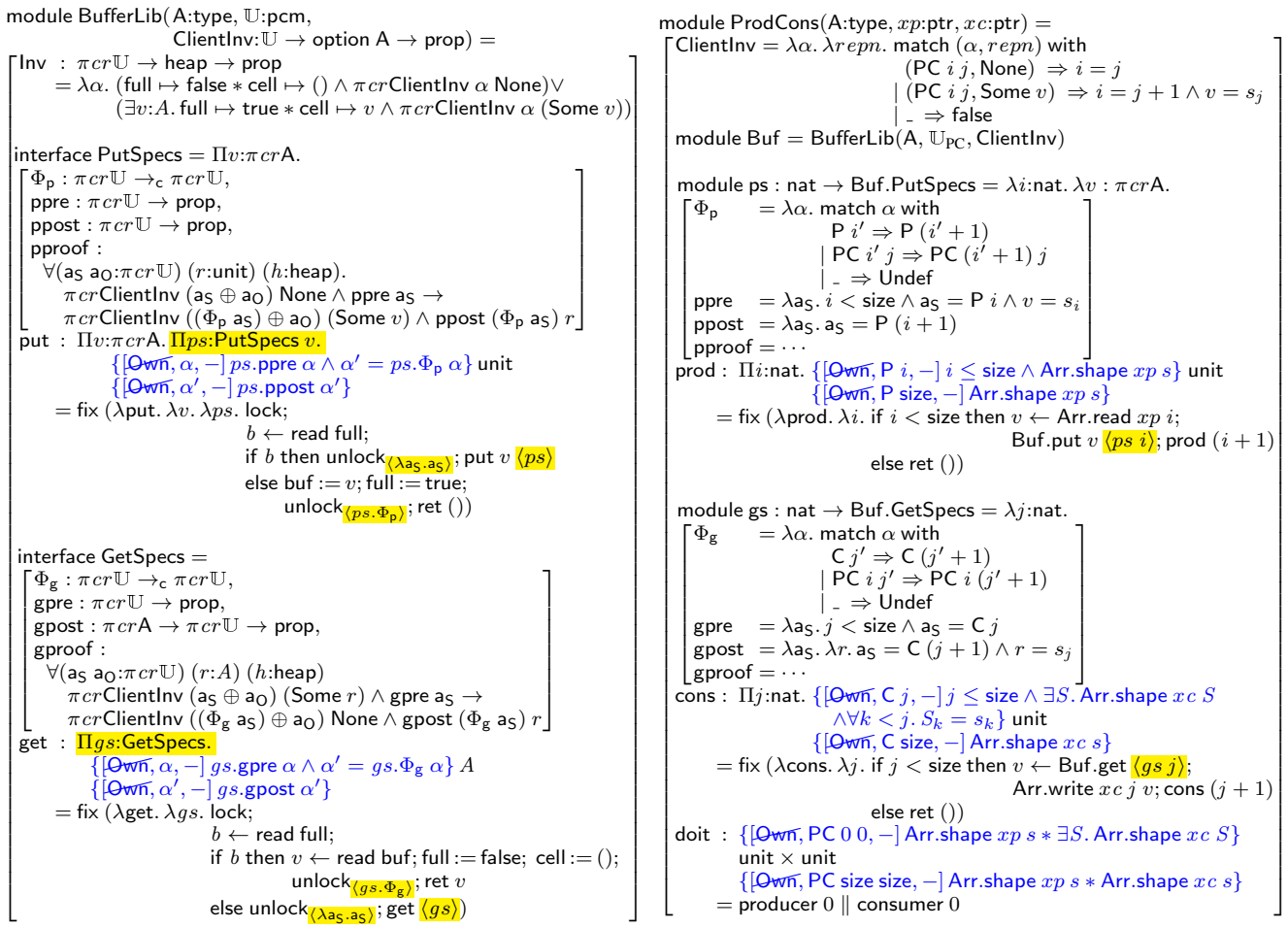

Figure 1. A buffer library (left) and its producer/consumer client (right).

interfaces, the Φ function has type $\mathbb{U} \to_c \mathbb{U}$, and the pre- and post-conditions are predicates over an auxiliary $\mathbb{U}$. A module matching the interface of PutSpecs is a function of the value v being inserted (hence the Πv:A argument in the interface definition); the implication pproof has an initial representation $repn = \mathsf{None}$ and a final representation $repn' = \mathsf{Some}\ v$ corresponding to successfully finding the buffer empty and then filling it with v. A module matching the interface of GetSpecs doesn't have an argument; the implication has an initial representation $repn = \mathsf{Some}\ r$ and a final representation $repn' = \mathsf{None}$ corresponding to successfully finding r in the buffer and then emptying it; note that the initial value r in the buffer is also the result, the quantification over r indicating that the implication must work for any possible contents found in the buffer.

The definition of put has two arguments: the value v to put into the buffer and a module ps matching the PutSpecs interface, the latter is shaded because it's auxiliary to the verification but not needed for execution. The type exposes that the auxiliary contribution changes from α to $\alpha' = ps.\Phi_{\mathsf{p}}(\alpha)$, where $ps.\Phi_{\mathsf{p}}$ projects the Φ_{p} component from the ps module. Observe that the local contribution is combined with the environment's contribution when passed to the client invariant, but the pre and post only depend on the local contribution. The code is a recursive definition that loops until it successfully enters the critical section and finds the buffer empty, so it can place the contents in cell and update the full flag, then exit the critical section and update the client's auxiliary (it passes $ps.\Phi_{\mathsf{p}}$ as an argument to unlock); if the buffer is full, then the code unlocks with the same local contribution and retries with a recursive call. Although the proof of the implication (i.e., ps.pproof) doesn't appear in the code, it's used to typecheck the successful unlock.

The definition of get only has one argument: a module gs matching the GetSpecs interface. The code is a recursive definition that loops until it successfully finds a value in the buffer to remove.

1.2 Buffer Client

The client (Figure 1, right) consists of a producer and a consumer that can be verified against the buffer's specification without relying on its implementation (i.e., heap invariant and code).

The producer has an array with base pointer xp of some size which contains the elements of a sequence s (i.e., its heap contains $xp + i \mapsto s_i$ for each index i). Likewise, the consumer has an array with base pointer xc of the same size, but its initial contents are arbitrary. The producer iteratively puts each element from its private array into the shared buffer. Concurrently, the consumer iteratively gets from the shared buffer to populate its private array. The intended postcondition is for the producer and consumer to have arrays with identical contents.

The heap predicate Arr.shape $x\ s$ represents the contents of the array:

$$\mathsf{Arr.shape}\ x\ s \ \widehat{=}\ (x + 0 \mapsto s_0) * \cdots * (x + \mathsf{size} - 1 \mapsto s_{\mathsf{size}-1})$$

The ith index of an array can be read with $\mathsf{Arr.read}\ x\ i$ and written value v with $\mathsf{Arr.write}\ x\ i\ v$.

Client's PCM. Intuitively, the producer (resp., consumer) needs an auxiliary number i' (resp., j') to relate its iteration index i (resp., j) and loop invariant to the client invariant embedded in the buffer. Thus, the PCM for the producer and consumer's contributions is defined by the following *inductive type* and associated operations:

$$\begin{array}{rcl} \mathbb{U}_{\mathsf{PC}} & = & \mathsf{N} \mid \mathsf{P}\ i \mid \mathsf{C}\ j \mid \mathsf{PC}\ i\ j \mid \mathsf{Undef} \\ x \oplus y & = & \begin{cases} x & \text{if } y = \mathsf{N} \\ y & \text{if } x = \mathsf{N} \\ \mathsf{PC}\ i\ j & \text{if } (x,y) = (\mathsf{P}\ i, \mathsf{C}\ j) \text{ or } (\mathsf{C}\ j, \mathsf{P}\ i) \\ \mathsf{Undef} & \text{otherwise} \end{cases} \\ \mathbb{0} & = & \mathsf{N} \end{array}$$

The $\mathbb{U}_{\mathsf{PC}}$ type has five distinct possible values that describe the thread behavior on the shared buffer: (1) $\mathsf{P}\ i$ is a producer that has advanced to the index i in its private array; (2) $\mathsf{C}\ j$ is a consumer that has advanced to the index j in its private array; (3) $\mathsf{PC}\ i\ j$ is a thread that both produces and consumes; (4) N is a thread that neither produces nor consumes, though it can enter the critical section; (5) Undef indicates an inconsistent behavior (e.g., two threads trying to produce).

Client's Parameters. The client module has a parameter A for the array contents' type and the base array pointers xp and xc. The verification of the producer and consumer can be done generically for arrays with any type of content. We assume there is a finite function s that describes the contents of the producer's array, which is used in the functions' specifications to relate the value inserted by the producer to the value removed by the consumer.

Client's Invariant. The client defines its invariant $\mathsf{ClientInv}$ as follows: the collective auxiliary contribution must have exactly one producer and consumer (hence $\alpha = \mathsf{a_S} \oplus \mathsf{a_O}$ is of the form $\mathsf{PC}\ i\ j$); if the buffer's representation is initially None, then the producer and consumer should have the same index ($i = j$); if the buffer's representation is initially $\mathsf{Some}\ v$, then the producer has put but the consumer has yet to get, so their indices are off by one ($i = j + 1$) and the contents of the buffer match the jth element of the producer's array ($xp + j \mapsto s_j$). Note that the client invariant only depends on the auxiliary state and the abstract representation of the buffer; it does not know the shape of the buffer's heap.

Thus, the buffer library can be instantiated as Buf by passing the necessary arguments to $\mathsf{BufferLib}$: the type A of the contents, the PCM $\mathbb{U}_{\mathsf{PC}}$, and the client invariant $\mathsf{ClientInv}$.

Producer. The producer defines a module ps matching the buffer's expected interface $\mathsf{PutSpecs}$. It's a function of the producer's iteration index i so that we can instantiate it at every iteration of the producer's loop. It also has an argument value v to match the argument required by $\mathsf{PutSpecs}$. Since the producer is put's client, it must define the necessary Φ_{p} function, pre- and postconditions, and prove the implication. The Φ_{p} function increment's the producer's contribution i by 1; the function has an analogous behavior on $\mathsf{PC}\ i\ j = \mathsf{P}\ i \oplus \mathsf{C}\ j$ to satisfy unlock's locality requirement (*cf.* Section **??**); for all the other values it's Undef. ppre checks that the index i is within bounds, the local auxiliary contribution is the behavior of a producer ($\mathsf{a_S} = \mathsf{P}\ i$), and that the value being inserted is the ith element of s. ppost checks that the producer's index went up by 1, as to be expected from the Φ_{p} function. The pproof witnesses the implication described above; it is elided for reasons of space, but we emphasize that both proofs and programs can be expressed in the language.

Next, the type of prod says that it has an argument i, and if the initial auxiliary contribution is $\mathsf{P}\ i$ and the index is within the array's bounds, then after the program runs the auxiliary contribution will be $\mathsf{P}\ \mathsf{size}$ and the array will remain in the heap. The code is a recursive definition that terminates if the index is out of bounds, otherwise it reads a value v from the private array xp and puts it into the shared buffer, and finally loops on the next index. The ps module is instantiated with the index i and passed as auxiliary data to typecheck the put call.

Consumer. Similarly, the consumer defines a module gs matching the interface $\mathsf{GetSpecs}$ and the code cons for populating the consumer's private array. The Φ_{g} function increments the consumer's index j by 1. Whereas the producer's *pre*condition related the client's invariant to the value inserted, the consumer's *post*condition relates the invariant to the value removed. Thus the consumer's gpre requires the consumer's index j to be within bounds, while gpost ensures that the index has incremented by 1 and result r of get is the jth element of the sequence s.

The type of cons specifies as loop invariant that the consumer's private array contains some sequence S which agrees with the sequence s up to, but not including, index j. The precondition requires the iteration index j to be at most size, the postcondition ensures that at the end the consumer's contribution is size and its array agrees with s. The code for cons is a recursive loop over argument j that gets from the shared buffer and writes into the jth entry in its private array for all entries less than size. The gs module is instantiated with the iteration index j in order to typecheck the get call (i.e., to verify that the pre/post/invariant implication holds).

Tying it all together. Finally, we can define the top-level program doit that runs the producer and consumer in parallel with initial iteration indices $i = 0 = j$. The precondition shows that the local auxiliary contribution is that of a producer and consumer at index 0 ($\mathsf{PC}\ 0\ 0$) and there are two arrays: one that represents the sequence s and another with some unknown sequence S. Since the local auxiliary and heap can be split to satisfy the $\mathsf{producer}$'s and $\mathsf{consumer}$'s preconditions, the parallel composition rule allows us to conclude the postcondition with both producer and consumer at the end of the array ($\mathsf{PC}\ \mathsf{size}\ \mathsf{size}$) and both arrays representing s.