EUROPEAN COMMISSION
SEVENTH FRAMEWORK PROGRAMME
FP7-ICT-2011-C
Grant agreement no.: 308830

Deliverable D5.1

Progress report for WP5: Architecture-driven verification of complex software components

Project acronym	ADVENT
Project title	Architecture-driven verification of systems software
Funding scheme	FP7 FET Young Explorers
Scientific coordinator	Dr. Alexey Gotsman, IMDEA Software Institute, Alexey.Gotsman@imdea.org, +34 911 01 22 02

1 Summary

Work Package 5 in the ADVENT project is concerned with reasoning about key concurrent components, critical tightly-connected parts of software systems. We have made notable progress in this area, as described below.

- We developed a novel proof technique for establishing the *linearizability* (functional correctness) of concurrent queue algorithms. Concurrent queues are key components in many critical software systems, e.g., operating systems and web servers, and thus verifying their correctness is paramount. Our approach is unique as it reduces the challenging verification problem into proving certain *spatio-temporal architectural aspects* of concurrent queue algorithms. We have shown that any queue algorithm which correctly implements these aspects is indeed linearizable. This effort resulted in a paper in CONCUR'13 and in a verification tool (see D6.1).

- We developed CaReSL, the first program logic that enables the use of granularity abstraction for modular verification of higher-order concurrent programs. CaReSL is able to handle complicated fine-grained concurrent software artifacts which are parameterised using *higher order functions* enables to reuse the results of verifying, e.g., the work stealing mechanism in Cilk [3] and the implementation of concurrent iterators in `java.util.concurrency`. This allows reusing the results of verifying such complicated concurrent components in any context in which they are instantiated. Furthermore, its support for *granularity abstraction* allows hiding the intricacies of such complicated concurrent components and the spatio-temporal invariants which guarantee their correctness when reasoning about their clients. This effort resulted in a paper in ICFP'13.

- We introduced *Concurrency-Aware Linearizabilty (CAL)*, a novel correctness condition for concurrent data structures. We observed that certain key software components, e.g., the `Exchangers` in `java.util.concurrent.Exchanger`, cannot specified using the standard notion of linearizability. CAL rectifies this unfortunate situation by allowing a new class of specifications. This class paves the way for developing novel modular verification techniques for *truly* concurrent components, which is a subject that we are currently working on. In a way, CAL can be seen as a technology transfer from our work on inter-component modular reasoning (see D2.1) into the realm of intra-component verification: It allows to untie the tight connections between *parts of components* using a novel form of specification and thus to greatly simplify their verification. This effort has already resulted in a brief announcement in PODC'14.

- In addition to looking at concurrent systems, we have expended our efforts into verifying key components in *distributed systems* using the techniques developed in WP2. More specifically, we developed reasoning techniques for components implementing the conflict resolution mechanisms in eventually consistent replicated stores. Verifying such components is very challenging due to the highly concurrent nature of a replicated store, with multiple replicas simultaneously updating their object copies and exchanging messages. We addressed this challenge by proposing *replication-aware simulations* which allows *reducing* the verification of a distributed system comprised of multiple replicas into local reasoning about a single one by exploiting certain *architectural properties* of the agreement protocols used for synchronizing local updates to the replica's store with the ongoing stream of received messages pertaining to remote updates. Apart from proving correctness of replicated data type implementations, we have also devised techniques for proving the optimality of their resource usage using a novel method for proving lower bounds on the *worst-case metadata overhead* of replicated data types—the proportion of metadata relative to the client-observable content. Using our method, we proved that four of the implementations we verified have an optimal worst-case metadata overhead among all implementations satisfying the same specification. This effort resulted in a paper in POPL'14.

2 Verifying Concurrent Queue Algorithms ([CONCUR13])

Concurrent queues are key data structures in concurrent systems. Implementing efficient and correct queues (e.g., as found in `java.util.concurrency` and Intel's Threading Building Blocks library) is very challenging: programmers use intricate programming techniques to reduce synchronization, and hence make the queue more efficient. This makes intuitive understanding and formal reasoning very difficult.

In the context of concurrent data structures, linearizability [15] is the standard correctness requirement for concurrent data structure implementations. Intuitively, It amounts to showing that all methods are atomic and obey the high-level sequential specification of the data structure.

The standard way to prove linearizability is to use simulation: The verifier establishes an invariant relating the state of the implementation to the state of the specification, and identifies the linearization points, which when performed by the implementation change the state of the specification. (See, e.g., [1, 2, 4, 5, 8, 21, 25, 30, 31]). While for a number of concurrent algorithms, spotting the linearization points may be straightforward (and has even been automated to some extent [31]), in general specifying the linearization points can be very difficult. For instance, in implementations using a helping mechanism, they can lie in code not syntactically belonging to the thread and operation in question, and can even depend on future behavior. There are numerous examples in the literature, where this is the case; to mention only a few concurrent queues: the Herlihy and Wing queue [15], the optimistic queue [19], the elimination queue [22], the baskets queue [18], the flat-combining queue [13]. In our experience, using standard simulation-based techniques for verifying such complicated data structure as mentioned above is very difficult and results in unintuitive proofs.

We developed an alternative simpler way of proving linearizability for concurrent queue algorithms. We *reduced* the task of proving linearizability to establishing four relatively simple properties. In (loose) analogy to aspect-oriented programming, we called our approach "aspect-oriented" linearizability, because each of these four properties can be proved independently.

So what are these properties? A correct (i.e., linearizable) concurrent queue:

(1) must not allow dequeuing an element that was never enqueued;
(2) it must not allow the same element to be dequeued twice;
(3) it must not allow elements to be dequeued out of order; and
(4) it must correctly report whether the queue is empty or not.

Although similar properties were already mentioned by Herlihy and Wing [15], we are the first to prove that suitably formalized versions of these four properties are not only necessary, but also sufficient, conditions for linearizability with respect to the queue specification, at least for what we call *purely-blocking* implementations. This is a rather weak requirement satisfied by all non-blocking methods, as well as by possibly blocking methods, such as Herlihy and Wing `deq()` method, whose blocking executions do not modify the global state.

We implemented this new *reduction-based* approach as part of Cave [31], a verification tool based on separation logic, and used it to verify a linked-list version of Herlihy and Wing's queue. (See WP6 regarding automation.)

3 Reasoning about Parametrised libraries ([ICFP'13])

Modular programming and modular verification go hand in hand: Programmers use modularity in the design of their concurrent programs, to allow reusing individual program components which, in turn, requires reasoning about these components in relative isolation. Unfortunately, most existing advanced logics for concurrency, e.g., [6, 7, 9, 20, 24, 28, 29], are geared toward building proofs that reflect the data abstraction inherent in well-designed programs, ignore two crucial forms of modularity: *higher-order functions*, which are essential for building reusable components, and *granularity abstraction*, a key technique for hiding the intricacies of fine-grained concurrent data structures from the clients of those data structures. Reasoning about such complicated components is particularly challenging the correctness of these data structure is often based on certain spatio-temporal protocols which every thread must follow and hence requires temporal reasoning about the parts of the states they manipulate. Modular reasoning about these components is even more challenging as it requires hiding these spatio-temporal invariants which guarantee the correctness of the data structures when reasoning about their clients.

We introduced CaReSL, the first logic to support the use of granularity abstraction for modular verification of higher-order concurrent programs. Our logic allows to reason about complicated concurrent components and libraries which are parameterised using higher order functions. The use of higher order functions is quite common in system software, for example it is used by the work stealing mechanism in Cilk [3] and the implementation of concurrent iterators in `java.util.concurrency`, just to name a few.

In contrast to existing logics, CaReSL takes full advantage of the architectural aspects overlooked by existing works, namely, higher-order functions and granularity abstraction. To do so, it builds on ideas from two distinct lines of research: Kripke logical relations and Concurrent separation logics.

Kripke logical relation are a fundamental technique for proving refinement in languages with higher-order functions, polymorphism, and recursive types. Logical relations explain observable behaviour in terms of the logical interpretation of each type. For example, two functions of type $\tau \to \tau'$ are logically related if giving them related inputs of type τ implies that they will produce related results of type τ'. Kripke logical relations are defined relative to a "possible world" that describes the relationship between the internal, hidden state of a program and that of its specification.

Concurrent separation logics. Separation logic is a Hoare logic in which assertions are understood relative to some portion of the heap, with the separating conjunction $P * Q$ dividing up the heap between assertions P and Q. Most concurrent separation logics also add some form of protocols (e.g., rely-guarantee), which facilitate compositional reasoning by constraining the potential interference between threads operating concurrently on some shared portion of the heap.

CaReSL attempts to unify these techniques. So, for example, the logic does not include refinement as a primitive notion. Instead, refinement is derived from Hoare-style reasoning: to prove that e refines e', one merely proves a particular Hoare triple about e, allowing the tools of concurrent separation logic to be exploited in the proof of logical relatedness. In the other direction, CaReSL is a modal logic with the possible worlds of Kripke logical relations, which can in turn be used to express the shared-state protocols of concurrent separation logics.

To demonstrate its effectiveness, we used CaReSL to construct the first formal proof of correctness for Hendler et al.'s "flat combining" algorithm [13]. Flat combining provides a generic way to turn a sequential ADT into a relatively efficient concurrent one, by having certain threads perform the combined actions requested by a whole bunch of other threads. The flat combining algorithm is interesting not only because it itself is a rather sophisticated higher-order function, but also because it is modularly assembled from other higher-order components, including a fine-grained stack with concurrent iterator. We therefore take the opportunity to verify flat combining in a modular way that mirrors the modular structure of its implementation.

4 Verifying Concurrency-Aware Concurrent Data Structures ([PODC'14])

Linearizability [16] is the main correctness condition for concurrent data structures. It is a natural and important property as in many realistic programming models linearizability amounts to observational refinement [10,11]. Intuitively, a concurrent object is linearizable if in every execution each operation seems to take effect instantaneously between its invocation and response, and the resulting sequence of (seemingly instantaneous) operations respects a given sequential specification.

Unfortunately, for some concurrent objects it is *impossible* to provide a sequential specification: their behaviour in the presence of concurrent operations is *observably different* from their behaviour in the sequential setting. For example, `Exchanger` objects as found, e.g., in `java.util.concurrent.Exchanger`, serve as a synchronization point at which threads can pair up and *atomically swap* elements. (Note that elements can be swapped only when the Exchanger is used concurrently by several threads.) `Exchangers` are useful in applications such as genetic algorithms and pipeline designs, and are embedded in practice in thread-pool implementations as well as other higher-level data structures [26,27].

We refer to objects whose behavior when used by multiple threads is observationally different from their behavior in the sequential setting as *Concurrency-Aware Concurrent Objects (CA-objects)*. For such objects the traditional notion of linearizability is simply not expressive enough to allow for describing all desired behaviours without introducing undesired ones. (E.g., allowing threads to swap elements without a counterpart thread running concurrently.) As a result, CA-objects are not given a formal specification. The lack of formal specifications is problematic as it prevents modular proofs.

To rectify the aforementioned unfortunate situation, we propose *concurrency-Aware Linearizabilty (CAL)* a new correctness condition which addresses the aforementioned problem. CAL enables programmers to provide natural and intuitive specifications for (certain kinds of) CA-objects. Technically, CAL is an extension of linearizabilty where *Concurrency-Aware specifications*, that describe concurrency-dependent behaviours, are allowed. Sequential specifications are a special case of concurrency-aware ones which indicate that concurrency does not lead to observably different behaviours.

CAL it is targeted to serve as the basis for modular verification of complicated CA-Objects: Currently, correctness proofs of concurrent objects that utilize `Exchanger`-like objects are neither modular nor reusable. For example, the correctness proof of the HSY-stack [14] mixes reasoning about the implementation of an (`Exchanger`-like) elimination array with its particular usage by the stack. Our goal in this work is to develop reasoning techniques that modularize proofs of this kind of data structures by separating the reasoning about the concurrency-aware (often generic) subcomponents from the reasoning about the higher level data structure.

The introduction of CAL can be seen as a technology transfer from our work on inter-component modular reasoning (see D2.1) into the realm of intra-component verification. More specifically, inspired by our previous work on the Hindsight lemma [23], we are currently developing a proof technique which partitions the verification of CA-objects into three stages: Firstly, we show that every thread manipulates the state according to a simple spatio-temporal protocol. Secondly, we show that following these protocols ensures the existence of a global state in which *all* concurrent operations can be linearized simultaneously. Thirdly, we use a lightweight temporal reasoning to show that all threads agree on that joined linearization point, thus obviating the need to reason about linearization points in the body of other threads. (See discussion in Section 3.). The unique aspect of our approach is that it uses CAL to decouple the reasoning about multiple entangled parts of such complicated concurrent components in a way which was not possible before as it enables hiding the intricacies used in the implementation of one *part* of the data structure when reasoning about its other parts.

5 Reasoning about Replicated Data Types ([POPL'14])

We have proposed techniques for reasoning about implementations of replicated data types—components used for conflict resolution in eventually consistent replicated stores that we specified in WP2. This is described in Sections 5-6 of the attached [POPL'14] paper.

First, we have proposed a method for proving the correctness of replicated data type implementations with respect to the specifications in WP2 and applied it to seven existing implementations of the four data types, including those with nontrivial optimizations: last-writer-wins register, counter, multi-value register and observed-remove set. Reasoning about the implementations is difficult due to the highly concurrent nature of a replicated store, with multiple replicas simultaneously updating their object copies and exchanging messages. We addressed this challenge by proposing *replication-aware simulations*. Like classical simulations from data refinement [17], these associate a concrete state of an implementation with its abstract description—structures on events, in our case. To combat the complexity of replication, they consider the state of an object at a single replica or a message in transit separately and associate it with abstract descriptions of only those events that led to it. Verifying an implementation then requires only reasoning about an instance of its code running at a single replica.

Here, however, we have to deal with another challenge: code at a single replica can access both the state of an object and a message at the same time, e.g., when updating the former upon receiving the latter. To reason about such code, we often need to rely on certain *agreement properties* correlating the abstract descriptions of the message and the object state. Establishing these properties requires global reasoning. Fortunately, we have found that agreement properties needed to prove realistic implementations depend only on basic facts about their messaging behavior and can thus be established once for broad classes of data types. Then a particular implementation within such a class can be verified by reasoning purely locally. By carefully structuring reasoning in this way, we achieved easy and intuitive proofs of data type implementations.

Apart from proving correctness of replicated data type implementations, we have also devised techniques for proving the optimality of their resource usage. This is useful since replicated data type designers strive to optimize their implementations; knowing that one is optimal can help guide such efforts in the most promising direction. For most data types we studied, the primary optimization target is the size of the metadata needed to resolve conflicts or handle network failures.

To establish optimality of metadata size, we developed a novel method for proving lower bounds on the *worst-case metadata overhead* of replicated data types—the proportion of metadata relative to the client-observable content. The main idea is to find a large family of executions of an arbitrary correct implementation such that, given the results of data type operations from a certain fixed point in any of the executions, we can recover the previous execution history. This implies that, across executions, the states at this point are distinct and thus must have some minimal size. Using our method, we proved that four of the implementations we verified have an optimal worst-case metadata overhead among all implementations satisfying the same specification.

References

[1] P. Abdulla, F. Haziza, L. Holk, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.

[2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *19th International Conference on Computer Aided Verification (CAV)*, pages 477–490, 2007.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[4] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *Electron. Notes Theor. Comput. Sci.*, 137(2):93–110, July 2005.

[5] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In *Proceedings of the 17th International Conference on Formal Methods*, FM, pages 323–337, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010 Object-Oriented Programming*, pages 504–528, 2010.

[7] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 259–270, New York, NY, USA, 2011. ACM.

[8] S. Doherty and M. Moir. Nonblocking algorithms and backward simulation. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC'09, pages 274–288, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[10] I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379 – 4398, 2010. European Symposium on Programming 2009 - ESOP 2009.

[11] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP*, pages 453–465, 2011.

[12] N. Hemed and N. Rinetzky. Brief announcement: Contention-aware linearizability. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2014.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[14] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *In SPAA04: Symposium on Parallelism in Algorithms and Architectures*, pages 206–215. ACM, 2004.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3), 1990.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[17] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1972.

[18] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In *Proceedings of the 11th International Conference on Principles of Distributed Systems*, OPODIS'07, pages 401–414, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. In *In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[20] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[21] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 321–337, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[23] P. W. OHearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight, 2010.

[24] F. Pottier. Hiding local state in direct style: A higher-order anti-frame rule. In *LICS*, pages 331–340, 2008.

[25] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 243–259, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] W. N. Scherer III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. *SCOOL 05*, page 83, 2005.

[27] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156. ACM, 2006.

[28] K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 169–188, Berlin, Heidelberg, 2013. Springer-Verlag.

[29] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

[30] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 335–348, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] V. Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 450–464, Berlin, Heidelberg, 2010. Springer-Verlag.

List of Attached Papers

[CONCUR'13] Thomas Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR'13: Concurrency Theory*, pages 242–256, 2013.

[ICFP'13] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP'13: ACM International Conference on Functional Programming*, pages 377–390, 2013.

[PODC'14] Nir Hemed and Noam Rinetzky. Brief announcement: Concurrency-aware linearizability. In *PODC'14: ACM Symposium on Principles of Distributed Computing*, 2014.

[POPL'14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *POPL'14: ACM Symposium on Principles of Programming Languages, San Diego, CA, USA, pages 271-284. ACM, 2014.*, 2014.

Aspect-Oriented Linearizability Proofs

Thomas A. Henzinger[1], Ali Sezgin[1], and Viktor Vafeiadis[2]

[1] IST Austria {tah,asezgin}@ist.ac.at
[2] MPI-SWS viktor@mpi-sws.org

Abstract. Linearizability of concurrent data structures is usually proved by monolithic simulation arguments relying on identifying the so-called linearization points. Regrettably, such proofs, whether manual or automatic, are often complicated and scale poorly to advanced non-blocking concurrency patterns, such as helping and optimistic updates.

In response, we propose a more modular way of checking linearizability of concurrent queue algorithms that does not involve identifying linearization points. We reduce the task of proving linearizability with respect to the queue specification to establishing four basic properties, each of which can be proved independently by much simpler arguments. As a demonstration of our approach, we verify the Herlihy and Wing queue, an algorithm that is challenging to verify by a simulation proof.

1 Introduction

Linearizability [8] is widely accepted as the standard correctness requirement for concurrent data structure implementations. It amounts to showing that all methods are atomic and obey the high-level sequential specification of the data structure. For example, an unbounded queue must support the following two methods: *enqueue*, which extends the queue by appending one element to its end, and *dequeue*, which removes and returns the first element of the queue.

The standard way to prove that a concurrent queue implementation is linearizable is to prove an invariant which relates the state of the implementation to the state of the specification. A well-established approach (e.g. [1–5, 11, 13–15]) is to identify the linearization points, which when performed by the implementation change the state of the specification, and to then construct a forward or backward simulation.

While for a number of concurrent algorithms, spotting the linearization points may be straightforward (and has even been automated to some extent [15]), in general specifying the linearization points can be very difficult. Due to helping, they can lie in code not syntactically belonging to the thread and operation in question, and can also depend on future behavior. There are numerous examples in the literature, where this is the case. To mention only a few concurrent queues: the Herlihy and Wing queue [8], the optimistic queue [10], the elimination queue [12], the baskets queue [9], the flat-combining queue [6].

```
1: var $q.back : int \leftarrow 0$
2: var $q.items$ : array of $val$
         $\leftarrow \{$NULL, NULL, ...$\}$

3: procedure enq($x : val$)
4:     $\langle i \leftarrow$ INC($q.back$)$\rangle$          $\triangleright E_1$
5:     $\langle q.items[i] \leftarrow x\rangle$          $\triangleright E_2$

6: procedure deq() : $val$
7:     while true do
8:         $\langle range \leftarrow q.back - 1\rangle$                 $\triangleright D_1$
9:         for $i = 0$ to $range$ do
10:            $\langle x \leftarrow$ SWAP($q.items[i]$, NULL)$\rangle$   $\triangleright D_2$
11:            if $x \neq$ NULL then return $x$
```

Fig. 1. Herlihy and Wing queue [8].

The HW Queue. In this paper, we focus on the Herlihy and Wing queue [8] (henceforth, HW queue for short) that illustrates nicely the difficulties encountered when defining a simulation relation based on linearization points. The code is given in Fig. 1. The queue is represented as a pre-allocated unbounded array, *q.items*, initially filled with NULLs, and a marker, *q.back*, pointing to the end of the used part of the array. Enqueuing an element is done in two steps: the marker to the end of the array is incremented (E_1), thereby reserving a slot for storing the element, and then the element is stored at the reserved slot (E_2). Dequeue is more complex: it reads the marker (D_1), and then searches from the beginning of the array up to the marker to see if it contains a non-NULL element. It removes and returns the first such element it finds (D_2). If no element is found, dequeue starts again afresh. Each of the four statements surrounded by $\langle\rangle$ brackets and annotated by E_i or D_i for $i = 1, 2$ is assumed to execute in isolation.

Consider the following execution fragment, where $\cdot$ denotes context switches between concurrent threads,

$$(t : E_1) \cdot (u : E_1) \cdot (v : D_1, D_2) \cdot (u : E_2) \cdot (t : E_2) \cdot (w : D_1)$$

which have threads t and u executing enqueue instances, v and w executing dequeue instances. At the end of this fragment, v is ready to dequeue the element enqueued by u, and w is ready to dequeue the element enqueued by t. In order to define a simulation relation from this interleaving sequence to a valid sequential queue behavior, where operations happen in isolation, we have to pick the linearization points for the two completed enqueue instances. The difficulty lies in the fact that no matter which statements are chosen as the linearization points for the two enqueue instances, there is always an extension to the fragment inconsistent with the particular choice of linearization points. For instance, if we pick $(t : E_1)$ as the linearization point for t, then the extension

$$(v : D_2, \mathbf{return}) \cdot (z : D_1, D_2, \mathbf{return})$$

requiring u's element be enqueued before that of t's, will be inconsistent. If on the other hand, any statement which makes u linearize before t, then the extension

$$(w : D_2, \mathbf{return}) \cdot (z : D_1, D_2, D_2, \mathbf{return})$$

requiring the reverse order of enqueueing will be inconsistent. This shows not only that finding the correct linearization sequence can be challenging, but also that the simulation proofs will require to reason about the entire state of the system, as the local state of one thread can affect the linearization of another.

Our Contribution. In our experience, this and similar tricks for reducing synchronization among threads so as to achieve better performance, make concurrent algorithms extremely difficult to reason about when one is constrained to establishing a simulation relation. However, if two methods overlap in time, then the only thing enforced by linearizability is that their effects are observed in *some* and same order by all threads. For instance, in the example given above, the simple answer for the particular ordering between the linearization points of the enqueue instances of t and u, is that it does not matter! As long as enqueue instances overlap, their values can be dequeued in any order.

Building on this main observation, our contribution is to simplify linearizability proofs by modularizing them. We reduce the task of proving linearizability to establishing four relatively simple properties, each of which may be reasoned about independently. In (loose) analogy to aspect-oriented programming, we are proposing "aspect-oriented" linearizability proofs for concurrent queues, where each of these four properties will be proved independently.

So what are these properties? A correct (i.e., linearizable) concurrent queue:
(1) must not allow dequeuing an element that was never enqueued;
(2) it must not allow the same element to be dequeued twice;
(3) it must never reorder enqueued elements; and
(4) it must correctly report whether the queue is empty or not.

Although similar properties were already mentioned by Herlihy and Wing [8], we for the first time prove that suitably formalized versions of these four properties are not only necessary, but also sufficient, conditions for linearizability with respect to the queue specification, at least for what we call *purely-blocking* implementations. This is a rather weak requirement satisfied by all non-blocking methods, as well as by possibly blocking methods, such as HW `deq`() method, whose blocking executions do not modify the global state.

The rest of the paper is structured as follows: §2 recalls the definition of linearizability in terms of execution histories; §3 formalizes the aforementioned four properties, and proves that they are necessary and sufficient conditions for proving linearizability of queues; §4 returns to the HW queue example and presents a detailed manual proof of its correctness; and §5 explains how the bulk of this proof was also performed automatically by an adaptation of CAVE [15]. Finally, in §6 we discuss related work, and in §7 we conclude.

2 Technical Background

In this section, we introduce common notations that will be used throughout the paper and recall the definition of linearizability.

Histories, Linearizability. For any function f from A to B and $A' \subseteq A$, let $f(A') \overset{\text{def}}{=} \{f(a) \mid a \in A'\}$. Given two sequences x and y, let $x \cdot y$ denote their concatenation, and let $x \sim_{\text{perm}} y$ hold if one is a permutation of the other.

A *data structure* $\mathcal{D}$ is a pair $(D, \Sigma_{\mathcal{D}})$, where D is the *data domain* and $\Sigma_{\mathcal{D}}$ is the *method alphabet.* An *event* of $\mathcal{D}$ is a triple (m, d_i, d_o), for some $m \in \Sigma_{\mathcal{D}}$, $d_1, d_2 \in D$. Intuitively, (m, d_i, d_o) denotes the application of method m with input argument d_i returning the output value d_o. A sequence over events of $\mathcal{D}$ is called a *behavior.* The *semantics* of data structure $\mathcal{D}$ is a set of behaviors, called *legal* behaviors.

Each event $a = (m, d_i, d_o)$ generates two *actions*: the *invocation* of a, written as $inv(a)$, and the *response* of a, written as $res(a)$. We will also use $m_i(d_i)$ and $m_r(d_o)$ to denote the invocation and the response actions, respectively. When a particular method m does not have an input (resp., output) parameter, we will write $(m, \bot, x)$ (resp., $(m, x, \bot)$), and $m_i()$ (resp., $m_r()$) for the corresponding invocation (resp., response) action.

In this paper, a *history* of $\mathcal{D}$ is a sequence of invocation and response actions of $\mathcal{D}$. We will assume the existence of an implicit identifier in each history c that uniquely pairs each invocation with its corresponding response action, if the latter also occurs in c. A history c is *well-formed* if every response action occurs after its associated invocation action in c. We will consider only well-formed histories. An event is *completed* in c, if both of its invocation and response actions occur in c. An event is *pending* in c, if only its invocation occurs in c. We define *remPending*(c) to be the sub-sequence of c where all pending events have been removed. An event e precedes another event e' in c, written $e \prec_c e'$, if the response of e occurs before the invocation of e' in c. For event e, *Before*(e, c) denotes the set of all events that precede e in c. Similarly, *After*(e, c) denotes the set of all events that are preceded by e in c. Formally,

$$\mathit{Before}(e,c) \overset{\text{def}}{=} \{e' \mid e' \prec_c e\} \qquad \text{and} \qquad \mathit{After}(e,c) \overset{\text{def}}{=} \{e' \mid e \prec_c e'\}\,.$$

History c is called *complete* if it does not have any pending events. For a possibly incomplete history c, a *completion* of c, written $\hat{c}$, is a (well-formed) complete history such that $\hat{c} = \mathit{remPending}(c \cdot c')$ where c' contains only response events. Let *Compl*(c) denote the set of all completions of c.

A history is called *sequential* if all invocations in c are immediately followed by their matching responses, with the possible exception of the very last action which can only be the invocation of a pending event. We identify complete sequential histories with behaviors of $\mathcal{D}$ by mapping each consecutive pair of matching actions in the former to its event constructing the latter. A sequential history s is a *linearization* of a history c, if there exists $\hat{c} \in \mathit{Compl}(c)$ such that $\hat{c} \sim_{\text{perm}} s$ and whenever $e \prec_{\hat{c}} e'$ we have $e \prec_s e'$.

Definition 1 (Linearizability [8]). *A set of histories C is linearizable with respect to a data structure $\mathcal{D}$, if for any $c \in C$, there exists a linearization of c which is a legal behavior of $\mathcal{D}$.*

Queues. The method alphabet Σ_Q of a queue is the set $\{\texttt{enq}, \texttt{deq}\}$. We will take the data domain to be the set of natural numbers, $\mathbb{N}$, and a distinguished symbol `NULL` not in $\mathbb{N}$. Events are written as $\texttt{enq}(x)$, short for $(\texttt{enq}, x, \bot)$, and $\texttt{deq}(x)$, short for $(\mathtt{deq}, \bot, x)$. Events with $\mathtt{enq}$ are called *enqueue* events, and those with $\mathtt{deq}$ are called *dequeue* events.

Let c be a history. $Enq(c)$ denotes the set of all enqueue events invoked (and not necessarily completed) in c. Similarly, $Deq(c)$ denotes the set of all dequeue events invoked in c. A set $A \subseteq Enq(c) \cup Deq(c)$ is *closed under* $\prec_c$ if $a \in A$ and $b \prec_c a$, then $b \in A$.

For an $\mathtt{enq}$ event e in c, $Val_c(e)$ denotes the value to be inserted by e in c. Formally, $Val_c(\mathtt{enq}(x)) = x$. Similarly, for a completed $\mathtt{deq}$ event d in c, $Val_c(d)$ denotes the value removed by d in c. Formally, $Val_c(\mathtt{deq}(x)) = x$. For a pending $\mathtt{deq}$ event, $Val_c(\mathtt{deq}(x))$ is undefined.

We will use a labelled transition system, LTS_Q, to define the queue semantics. The states of LTS_Q are sequences over $\mathbb{N}$, the initial state is the empty sequence ε. There is a transition from q to q' with action a, written $q \xrightarrow{a} q'$, if (i) $a = \mathtt{enq}(x)$ and $q' = q \cdot x$, or (ii) $a = \mathtt{deq}(x)$ and $q = x' \cdot q'$, or (iii) $a = \mathtt{deq}(\mathtt{NULL})$ and $q = q' = \varepsilon$. A queue is *partial* if the last transition ($\mathtt{NULL}$ returning dequeue event) is not allowed.

A *run* of LTS_Q is an alternating sequence $q_0 l_1 q_1 \dots l_n q_n$ of states and queue events such that for all $1 \leq i \leq n$, we have $q_{i-1} \xrightarrow{l_i} q_i$. The trace of a run is the sequence $l_1 \dots l_n$ of the events occurring on the run. A queue behavior b is *legal* iff there is a run of LTS_Q with trace b.

We find it useful to express the queue semantics in an alternative formulation.

Definition 2. *A queue behavior b has a* sequential witness *if there is a total mapping μ_{seq} from $Deq(b)$ to $Enq(b) \cup \{\bot\}$ such that*

- $\mu_{\text{seq}}(d) = e$ *implies* $Val_b(d) = Val_b(e)$,
- $\mu_{\text{seq}}(d) = \bot$ *iff* $Val_b(d) = \mathtt{NULL}$,
- $\mu_{\text{seq}}(d) = \mu_{\text{seq}}(d') \neq \bot$ *implies* $d = d'$,
- $e \prec_b e'$ *and there exists* d' *with* $\mu_{\text{seq}}(d') = e'$ *imply* $\mu_{\text{seq}}^{-1}(e) \prec_b d'$,
- $\mu_{\text{seq}}(d) = \bot$ *implies that*
  $|\{e \in Enq(b) \mid e \prec_b d\}| = |\{d' \in Deq(b) \mid d' \prec_b d \land \mu_{\text{seq}}(d') \neq \bot\}|$.

Proposition 1. *A queue behavior b is legal iff b has a sequential witness.*

Proof (Sketch). If b is legal, then, by definition, it has a run r in LTS_Q with trace b. Let d be a dequeue event occurring in b. Then there is a transition $q \xrightarrow{d} q'$ in r. If $d = \mathtt{deq}(x)$ for some $x \in \mathbb{N}$, then set $\mu_{\text{seq}}(d) = e$ where e is the enqueue event $\mathtt{enq}(x)$ which has inserted x into the state sequence. If $d = \mathtt{deq}(\mathtt{NULL})$, then set $\mu_{\text{seq}}(d) = \bot$. Then, it is easy to check that μ_{seq} satisfies all the conditions of being a sequential witness for b.

For the other direction, let μ_{seq} be a sequential witness for b. We observe that i) an element x is in state q iff an enqueue event $\mathtt{enq}(x)$ has happened on the prefix of the run ending at q and the dequeue event with $\mu_{\text{seq}}(d) = e$ has not happened on the same prefix, ii) for any two enqueue events e, e' with $e \prec_b e'$, $Val_b(e)$ occurs in a state before $Val_b(e')$, iii) the relative ordering of inserted elements in a state does not change as long as both are in the state, iv) each enqueue event inserts exactly one element to the state, v) each dequeue event $\texttt{deq}(x)$ with $x \neq \texttt{NULL}$ removes exactly one element from the state, and vi) the dequeue event $\texttt{deq}(\texttt{NULL})$ does not change the state. Then, by induction on the length of b, we show that b has a run in LTS_Q. $\square$

3 Conditions for Queue Linearizability

3.1 Generic Necessary and Sufficient Conditions

We start by reducing the problem of checking linearizability of a given history, c, with respect to the queue specification to finding a mapping from its dequeue events to its enqueue events satisfying certain conditions. Intuitively, we map each dequeue event to the enqueue event whose value the dequeue removed, or to nothing if the dequeue event returns NULL. We say that the mapping is *safe* if it pairs each deq event with a proper enq event, implying that elements are inserted exactly once and removed at most once. A safe mapping is *ordered* if it additionally respects precedence induced by c. Finally, an ordered mapping is a *linearizability witness* if all NULL returning deq events see at least one state where the queue is logically empty. Below, we formalize these notions.

Definition 3 (Safe Mapping). *A mapping Match from $Deq(c)$ to $Enq(c) \cup \{\bot\}$ is* safe *for c if*
(1) for all $d \in Deq(c)$, if $Match(d) \neq \bot$, then $Val_c(d) = Val_c(Match(d))$;
(2) for all $d \in Deq(c)$, $Match(d) = \bot$ iff $Val_c(d) = \texttt{NULL}$; and
(3) for all $d, d' \in Deq(c)$, if $Match(d) = Match(d') \neq \bot$, then $d = d'$.

Definition 4 (Ordered Mapping). *A safe mapping Match for c is* ordered *if*
(1) for all $d \in Deq(c)$, we have $d \not\prec_c Match(d)$; and
(2) for all $d, d' \in Deq(c)$, if $Match(d) \prec_c Match(d')$, then $d' \not\prec_c d$.

Definition 5 (Linearization Witness). *An ordered mapping Match for c is a* linearization witness *if for any $d \in Deq(c)$ with $Val_c(d) = \texttt{NULL}$, there exists a subset $D' \subseteq Deq(c)$ such that $Match(D')$ is closed under $\prec_c$ and $D' \cap After(d, c) = \emptyset$ and $Before(d, c) \cap Enq(c) \subseteq Match(D')$.*

The main result of this section is stated below.

Theorem 1. *A set of histories C is linearizable with respect to queue iff every $c \in C$ has a completion $\hat{c} \in Compl(c)$ that has a linearization witness.*

Proof. ($\Rightarrow$) If $c \in C$ is linearizable with respect to queue, then there is a linearization s of c which is a legal queue behavior. By Prop. 1, s has a sequential witness μ_{seq}. The mapping μ_{seq} satisfies the conditions of a linearization witness since all $\prec_c$ orderings are preserved in s.

($\Leftarrow$) Pick a $c \in C$ and let $\hat{c} \in Compl(c)$ be its completion that has a linearization witness *Match*. Let $<$ be some arbitrary total order on the events of $\hat{c}$. We construct the linearization of $\hat{c}$ inductively as follows:

Let c' be the prefix of $\hat{c}$ that has been processed, and let s' be the resulting sequential history. All events in s' are *placed.* Events that are not placed but are pending after c' are called *candidate.* We extend c' until the first response action that happens after c' in $\hat{c}$. Formally, let $c' \cdot c_e \cdot a_r$ be a prefix of $\hat{c}$ such that c_e contains only invocation actions and a_r is a response action. Let A denote the set of all candidate events after $c' \cdot c_e \cdot a_r$. The new s' is obtained by appending some $a \in A$ as the next event if
(1) a is an enqueue event, and there does not exist another enqueue event e such that $\mathit{Match}^{-1}(e) \prec_{\hat{c}} \mathit{Match}^{-1}(a)$ and e is not placed in s'; or
(2) a is a dequeue event with $\mathit{Val}_{\hat{c}}(a) \neq \texttt{NULL}$, $\mathit{Match}(a)$ is placed in s', and there does not exist another dequeue event d such that $\mathit{Match}(d) \prec_{\hat{c}} \mathit{Match}(a)$ and d is not placed in s'; or
(3) a is a dequeue event with $\mathit{Val}_{\hat{c}}(a) = \texttt{NULL}$ and the number of enqueue events in s' is equal to the number of dequeue events d with $\mathit{Val}_{\hat{c}}(d) \neq \texttt{NULL}$ in s'.
In case, where both first and second conditions are satisfied, the candidate element minimal with respect to $<$ is appended to s'. This iteration is repeated until there are no candidate events that satisfy any of the conditions, at which point the inductive step ends with setting c' to $c' \cdot c_e \cdot a_r$. The existence of *Match* guarantees that such a sequence can be constructed. The constructed sequence s has *Match* also as a sequential witness, completing the proof. □

3.2 Necessary and Sufficient Conditions for Complete Histories

We now focus on complete histories, namely ones with no pending events. We observe that their linearizability violations can always be manifested in terms of the dequeued values. Intuitively, the possible violations are:

(VFresh) A dequeue event returning a value not inserted by any enqueue event.
(VRepet) Two dequeue events returning the value inserted by the same enqueue event.
(VOrd) Two ordered dequeue events returning values inserted by enqueue events in the inverse order.
(VWit) A dequeue event returning NULL even though the queue is never logically empty during the execution of the dequeue event.

We have the following result which ties the above violation types to linearizable queues.

Proposition 2. *A complete history c has a linearization which is a legal queue behavior iff it has none of the* VFresh*,* VRepet*,* VOrd*,* VWit *violations.*

Proof (Sketch). First, note that as c has no pending events, $\mathit{Compl}(c) = \{c\}$. If c has a linearization which is a legal queue behavior, then by Theorem 1, c has a linearization witness *Match*, and so none of the violations can happen. As *Match* is safe, (VFresh) and (VRepet) cannot happen; as it is ordered, (VOrd) cannot occur; and as it is a linearization witness, likewise (VWit) cannot happen. Similarly, in the other direction, the absence of all the violations ensures the existence of a linearizability witness. □

We remark that none of the violations mentions the possibility of an element inserted by an enqueue being lost forever. This is intentional, as such histories are ruled out by the following proposition.

Proposition 3. *Given an infinite sequence of complete histories $c_1, c_2, \dots$ not containing any of the violations above, where for every i, c_i is a prefix of c_{i+1}, and the number of dequeue events in c_i is less than that of c_{i+1}, if c_1 contains an enqueue event* $\mathtt{enq}(x)$*, then exists some c_j containing* $\mathtt{deq}(x)$*.*

Proof. We prove this by contradiction. If there is no $\mathtt{deq}(x)$ event, then $\mathtt{enq}(x)$ is always in the queue, and so, from the absence of VWit violations, none of the dequeue events following $\mathtt{enq}(x)$ can return `NULL`. Also, since dequeue events cannot return values that were not previously enqueued (VFresh) and cannot return the same value multiple times (VRepet), and since the number of dequeue events is increasing, then there must also be new enqueue events. However, only finitely many of those are not preceded by $\mathtt{enq}(x)$ which completes in c_1. This means that eventually one dequeue event has to return an element inserted by $\mathtt{enq}(y)$ such that $\mathtt{enq}(x) \prec_{c_j} \mathtt{enq}(y)$, which is VOrd. □

For checking purposes, we find it useful to re-state the third violation as the following equivalent proof obligation.

(POrd) For any enqueue events e_1 and e_2 with $e_1 \prec_c e_2$ and $Val_c(e_1) \neq Val_c(e_2)$, a dequeue event cannot return $Val_c(e_2)$ if $Val_c(e_1)$ is never removed in c.

Thus, we need an invariant which specifies all those executions satisfying the premise of POrd, and prove that such an execution cannot end with a dequeue event (in the sense that no other method is preceded by that dequeue event) returning the value of e_2.

3.3 Necessary and Sufficient Conditions for Purely-Blocking Queues

There is a subtle complication in the statement of Theorem 1. The witness mapping is chosen relative to some completion of the concurrent history under consideration. However, because implementations may become blocked, such completions may actually never be reached. This means that one cannot reason about the correctness of a queue implementation by considering only reachable states. What we would ideally like to do is to claim that if the implementation violates linearizability, then there is a finite complete history of the implementation which has no witness. In other words, if the implementation contains an incomplete history with no witness, then that execution is the prefix of a complete history of the implementation.

Let C be the set of all possible execution histories of a library implementation. We call a library implementation *completable* iff for every history $c \in C$, we have $Compl(c) \cap C \neq \emptyset$. For completable implementations, it suffices to consider only complete executions.

Theorem 2. *A completable queue implementation is linearizable iff all its complete histories have none of the* VFresh*,* VRepet*,* VOrd *and* VWit *violations.*

Proof. ($\Rightarrow$) If some complete history has a violation, by Prop. 2, it has no linearization, contradicting the assumption that the implementation is linearizable.

($\Leftarrow$) Consider an arbitrary history c of the implementation. As the implementation is completable, there exists a completion $\hat{c} \in \mathit{Compl}(c)$ that is a valid history of the implementation. From our assumptions, $\hat{c}$ cannot have a violation, and so by Prop. 2, $\hat{c}$ has a linearization, and therefore so does c. □

Since it may not be obvious how to easily prove that an implementation is completable, we introduce the stronger notion of purely-blocking implementations, that is straightforward to check. We say that an implementation is *purely-blocking* when at any reachable state, any pending method, if run in isolation will terminate or its entire execution does not modify the global state.

Proposition 4. *Every purely-blocking implementation is completable.*

Proof. Given a history $c \in C$, we will construct $\hat{c} \in \mathit{Compl}(c) \cap C$. We fix a total order of pending events, and consider them in that order. For a pending method e, if running it in isolation terminates, then extend c with the corresponding response for e. Otherwise, the execution of e does not modify any global state and so can be removed from the history without affecting its realizability. □

We remark that our new notion of purely-blocking is a strictly weaker requirement than the standard non-blocking notions: *obstruction-freedom*, which requires all pending methods to terminate when run in isolation, as well as the stronger notions of lock-freedom and wait-freedom. (See [7] for an in depth exposition of these three notions.)

4 Manually Verifying the Herlihy-Wing Queue

Let us return to the HW queue presented in §1 and prove its correctness manually following our aspect-oriented approach.

First, observe that HW queue is purely-blocking: `enq`() always terminates, and `deq`() can update the global state only by reading $x \neq$ `NULL` at E_2, in which case it immediately terminates. So from Prop. 4 and Theorem 2, it suffices to show that it does not have any of the four violations. The last one, VWit, is trivial as the HW `deq`() never returns `NULL`. So, we are left with three violations whose absence we have to verify: VFresh, VRepet, and VOrd.

Intuitively, there are no VFresh violations because `deq`() can return only a value that has been stored inside the *q.items* array. The only assignments to *q.items* are E_1 and D_2: the former can only happen by an `enq`(x), which puts x into the array; the latter assigns `NULL`.

Likewise, there are no VRepet violations because whenever in an arbitrary history two calls to `deq`() return the same x, then at least twice there was an element of the *q.items* array holding the value x and was updated to `NULL`

```
procedure deq($v : val$)
  while true do
    $\langle range \leftarrow q.back - 1 \rangle$
    for $i = 0$ to $range$ do
      $\left( \begin{array}{l} \left\langle \begin{array}{l} x \leftarrow q.items[i]; \\ \mathsf{assume}(x = v \wedge x \neq \mathtt{NULL}); \\ q.items[i] \leftarrow \mathtt{NULL} \end{array} \right\rangle ; \\ \textbf{return } x \end{array} \right) \sqcup \left\langle \begin{array}{l} x \leftarrow q.items[i]; \\ \mathsf{assume}(x = \mathtt{NULL}); \\ q.items[i] \leftarrow \mathtt{NULL} \end{array} \right\rangle$
```

Fig. 2. The HW dequeue method instrumented with the prophecy variable v guessing its return value, where $\sqcup$ stands for non-deterministic choice.

by the `SWAP` instruction at D_2. Therefore, at least two assignments of the form $q.items[\_] \leftarrow x$ happened; i.e. there were at least two $\mathsf{enq}(x)$ events in the history.

We move on to the more challenging third condition, VOrd. We actually consider its equivalent reformulation, POrd. Fix a value v_2 and consider a history c where every method call enqueuing v_2 is preceded by some method call enqueuing some different value v_1 and there are no $\mathsf{deq}()$ calls returning v_1 (there may be arbitrarily many concurrent $\mathsf{enq}()$ and $\mathsf{deq}()$ calls enqueuing or dequeuing other values). The goal is to show that in this history, no $\mathsf{deq}()$ return v_2.

Let us suppose there is a dequeue d returning v_2, and try to derive a contradiction. For d to return v_2, it must have read $range \geq i_2$ such that $q.items[i_2] = v_2$. So, d must have read $q.back$ at D_1 after $\mathsf{enq}(v_2)$ incremented it at E_1.

Since, $\mathsf{enq}(v_1) \prec_c \mathsf{enq}(v_2)$, it follows that $\mathsf{enq}(v_2)$ will have read a larger value of $q.back$ at E_1 than $\mathsf{enq}(v_1)$. So, in particular, once $\mathsf{enq}(v_1)$ finishes, the following assertion will hold:

$$\exists i_1 < q.back.\ q.items[i_1] = v_1 \wedge (\forall j < i_1.\ q.items[j] \neq v_2) \qquad (*)$$

Note that since, by assumption, v_1 can never be dequeued, and any later $\mathsf{enq}(v_2)$ can only affect the $q.items$ array at indexes larger than i_1, $(*)$ is an invariant.

Given this invariant, however, it is impossible for d to return v_2, as in its loop it will necessarily first have encountered v_1.

5 Automation

As can be seen from our previous informal argument, establishing absence of VFresh and VRepet violations was relatively straightforward, whereas proving POrd was somewhat more involved. Therefore, in this section, we will focus on automating the proof of the third property, POrd. Towards the end of the section, we will discuss the automatic verification of the absence of VWit violations for queue implementations, where `deq` may return `NULL`.

Prophetic Instrumentation of Dequeues. Our proof technique relies heavily on instrumenting the $\mathsf{deq}()$ function with a prophecy variable 'guessing' the value that will be returned when calling it. Essentially, we construct a method, $\mathsf{deq}(v)$, such that the set of traces of $\bigsqcup_{x\in\mathbb{N}\cup\{\mathtt{NULL}\}} \mathtt{deq}(x)$ is equal to the set of traces of $\mathtt{deq}()$, where $\sqcup$ stands for non-deterministic choice. Figure 2 shows the resulting automatically-generated instrumented definition of $\mathtt{deq}(v)$ for the HW queue.

Our implementation of the instrumentation performs a sequence of simple rewrites, each of which does not affect the set of traces produced:

$$\begin{aligned}
\textbf{return } E &\rightsquigarrow \mathsf{assume}(v = E); \textbf{return } E\\
\textbf{if } B \textbf{ then } C \textbf{ else } C' &\rightsquigarrow (\mathsf{assume}(B); C) \sqcup (\mathsf{assume}(\neg B); C')\\
C; \mathsf{assume}(B) &\rightsquigarrow \mathsf{assume}(B); C \qquad \text{provided } \mathit{fv}(B) \subseteq \mathit{Locals} \setminus \mathit{writes}(C)\\
C; (C_1 \sqcup C_2) &\leftrightsquigarrow (C; C_1) \sqcup (C; C_2)\\
(C_1 \sqcup C_2); C &\leftrightsquigarrow (C_1; C) \sqcup (C_2; C)
\end{aligned}$$

In general, the goal of applying these rewrite rules is to bring the introduced $\mathsf{assume}(v = E)$ statements as early as possible without unduly duplicating code.

Proving Absence of VOrd *Violations.* It turns out that our automated technique for proving POrd also establishes absence of VFresh violations as a side-effect. We reduce the problem of proving absence of VFresh and VOrd violations to the problem of checking non-termination of non-deterministic programs with an unbounded number of threads. The reduction exploits the instrumented $\mathtt{deq}(v)$ definition: $\mathtt{deq}()$ cannot return a result x in an execution precisely if $\mathtt{deq}(x)$ cannot terminate in that same execution.

Theorem 3. *A completable queue implementation has no* VFresh *and* VOrd *violations iff for all $n \in \mathbb{N}$ and forall v_1 and v_2 such that $v_1 \neq v_2$, the program*[3]

$$\mathit{Prg} \overset{\text{def}}{=} b \leftarrow \mathsf{false};\ (\mathtt{deq}(v_2) \parallel \overbrace{C \parallel \ldots \parallel C}^{n\ times})$$

does not terminate, where

$$C \overset{\text{def}}{=} (\mathtt{enq}(v_1); b \leftarrow \mathsf{true}) \sqcup (\mathsf{assume}(b); \mathtt{enq}(v_2)) \sqcup \bigsqcup_{x\neq v_2} \mathtt{enq}(x) \sqcup \bigsqcup_{x\neq v_1} \mathtt{deq}(x)\,.$$

Proof. ($\Rightarrow$) We argue by contradiction. Consider a terminating history c of *Prg*. If $\mathtt{enq}(v_2)$ is not invoked in c, then as there are no VFresh violations, we know that no $\mathtt{deq}()$ in c can return v_2, contradicting our assumption that c is a terminating history of *Prg*. Otherwise, if $\mathtt{enq}(v_2)$ is invoked in c, then at some earlier point $\mathsf{assume}(b)$ was executed, and since initially b was set to false, this means that $b \leftarrow \mathsf{true}$ was executed and therefore $\mathtt{enq}(v_1) \prec_c \mathtt{enq}(v_2)$. Consequently, from

[3] For simplicity, we assume that the methods cannot distinguish the thread in which they are running (i.e., they do not use thread-local storage or thread identifiers). Handling thread identifiers properly is not difficult: we have to record a set of thread identifiers that are not currently in use. Before each method invocation, we have to atomically pick and remove an identifier from that set, and on returning from the method, we have to add the current identifier back the set of unused identifiers.

POrd, if there is $\texttt{deq}()$ in c returns v_2, there must be a $\texttt{deq}()$ in c that can be completed to return v_1, contradicting our assumption that c is a terminating history of *Prg*.

($\Leftarrow$) We have two properties to prove. For VFresh, it suffices to consider the restricted parallel context that never chooses to execute the first two of the non-deterministic choices. In this restricted context, namely one that never enqueues v_2, $\texttt{deq}(v_2)$ does not terminate, and so $\texttt{deq}()$ cannot return v_2. For VOrd, consider a history in which every $\texttt{enq}(v_2)$ happens after some enqueue of a different value, say $\texttt{enq}(v_1)$, and in which there is no $\texttt{deq}(v_1)$. Such a history can easily be produced by the unbounded parallel composition of C, and so $\texttt{deq}(v_2)$ also does not terminate, as required. □

To prove non-termination, we essentially prove the partial-correctness Hoare triple, $\{\mathsf{true}\}$ *Prg* $\{\mathsf{false}\}$. Given a sound program logic, the only way for such a triple to hold is for the program to always diverge.

Implementation within CAVE. To prove such triples, we have midly adapted the implementation of CAVE [15], a sound but incomplete thread-modular concurrent program verifier that can handle dynamically allocated linked list data structures, fine-grained concurrency. The tool takes as its input a program consisting of some initialization code and a number of concurrent methods, which are all executed in parallel an unbounded number of times each. When successful, it produces a proof in RGSep that the program has no memory errors and none of its assertions are violated at runtime. Internally, it performs RGSep action inference [16] with a rich shape-value abstract domain [14] that can remember invariants of the form that value v_1 is inside a linked list. CAVE also has a way of proving linearizability by a brute-force search for linearization points (see [15] for details), but this is not applicable to the HW queue and therefore irrelevant for our purposes.

The main modifications we had to perform to the tool were: (1) to add code that instruments $\texttt{deq}()$ methods with a prophecy argument guessing its return value, thereby generating $\texttt{deq}(v)$; (2) to improve the abstraction function so that it can remember properties of the form $v_2 \notin X$, which are needed to express the $(*)$ invariant of the proof in §4; and (3) to add some glue code that constructs the *Prg* verification condition and runs the underlying prover to verify it.

As CAVE does not support arrays (it only supports linked lists), we gave the tool a linked-list version of the HW queue, for which it successfully verified that there are no VFresh and VOrd violations.

Showing Absence of VWit *Violations.* Here, we have to show that any dequeue event cannot return empty if it never goes through a state where the queue is logically empty. This in turn means that we have to express non-emptiness using only the actions of the history (and not referring to the linearization point or the gluing invariant which relates the concrete states of the implementation to the abstract states of the queue). For the following let us fix a (complete) concurrent history c and a dequeue of interest d which returns $\texttt{NULL}$ and does not precede any other event in c.

Let c' be some prefix of c and let $e \in \mathit{Enq}(c')$ be a complete enqueue event in c'. We will call e *alive* after c' if there is no completion of c' in which the dequeue event $\mathtt{deq}(\mathit{Val}_{c'}(e))$ occurs. Let d_i denote the dequeue event which removes the element inserted by the enqueue event e_i; that is, $d_i = \mathtt{deq}(\mathit{Val}_c(e_i))$. A sequence $e_0 e_1 \dots e_n$ of enqueue events in $\mathit{Enq}(c)$ is *covering* for d in c if the following holds:

- e_0 is alive at c' where c' is the maximal prefix of c such that $d \notin \mathit{Deq}(c')$.
- For all $i \in [1, n]$, e_i starts before d completes.
- For all $i \in [1, n]$, we have $e_i \prec_c d_{i-1}$.
- e_n is alive at c.

Note that all d_i must exist by the third condition and that d_n does not exist by the last condition. Then, the sequence is covering for d if d_0 does not start before d starts, and every enqueue event e_i completes before the dequeue event d_{i-1} starts. Intuitively, this means that at every state visited during the execution of d, the queue contains at least one element. The property corresponding to the last violation (VWit) then becomes the following:

(PWit) A dequeue event d cannot return NULL if there is a covering for d.

We will actually re-state the same property in a simpler way by making the following observation.

Proposition 5. *There is a covering for d in c iff at every prefix c' of c such that d is running, there is at least one alive enqueue event.*

Then, we can alternatively state PWit as follows:

(PWit′) A dequeue event d cannot return NULL if for every prefix c' at which d is pending there exists an alive enqueue event.

Note that, POrd can also be stated in terms of alive enqueue events.

(POrd′) For any enqueue events e_1 and e_2 with $e_1 \prec_c e_2$ and $\mathit{Val}_c(e_1) \neq \mathit{Val}_c(e_2)$, a dequeue event cannot return $\mathit{Val}_c(e_2)$ if e_1 is alive at c.

6 Related Work

Linearizability was first introduced by Herlihy and Wing [8], who also presented the HW queue as an example whose linearizability cannot be proved by a simple forward simulation where each method performs its effects instantaneously at some point during its execution. The problem is, as we have seen, that neither of E_1 or E_2 can be given as the (unique) linearization point of `enq` events, because the way in which two concurrent enqueues are ordered may depend on not-yet-completed concurrent `deq` events. In other words, one cannot simply define a mapping from the concrete HW queue states to the queue specification states. Nevertheless, Herlihy and Wing do not dismiss the linearization point technique completely, as we do, but instead construct a proof where they map concrete states to non-empty sets of specification states.

This mapping of concrete states to non-empty sets of abstract states is closely related to the method of *backward simulations*, employed by a number of manual proof efforts [3, 5, 13], and which Schellhorn et al. [13] recently showed to be a complete proof method for verifying linearizability. Similar to forward simulation proofs, backward simulation proofs, are monolithic in the sense that they prove linearizability directly by one big proof. Sadly, they are also not very intuitive and as a result often difficult to come up with. For instance, although the definition of their backward simulation relation for the HW queue is four lines long, Schellhorn et al. [13] devote two full pages to explain it.

As a result, most work on automatically verifying linearizability (e.g. [2, 14, 15, 1]) has relied on the simpler technique of forward simulations, even though it is known to be incomplete. The programmer is typically required to annotate each method with its linearization points and then the verifier uses some kind of shape analysis that automatically constructs the simulation relation. This approach seems to work well for simple concurrent algorithms such as the Treiber stack and the Michael and Scott queues, where finding the linearization points may be automated by brute-force search [15], but cannot handle more challenging examples such as the ones mentioned in the introduction.

Among this line of work, the most closely related one to this paper is the recent work by Abdulla et al. [1], who verify linearizability of stack and queue algorithms using observer automata that report specification violations such as our VOrd. Their approach, however, still requires users to annotate methods with linearization points, because checker automata are synchronized with the linearization points of the implementation.

We would also like to point out that the use of forward simulations is not limited to automated verifications of linearizability. Several manual verification also used forward simulations (e.g. [4, 3]).

To the best of our knowledge, there exist only two earlier published proofs of the HW queue: (1) the original pencil-and-paper proof by Herlihy and Wing [8], and (2) a mechanized backward simulation proof by Schellhorn et al. [13].

Both proofs are manually constructed. In comparison, our new proof is simpler, more modular, and largely automatically generated.[4] This is largely due to the fact that we have decomposed the goal of proving linearizability into proving four simpler properties, which can be proved independently. This may allow one to adapt the HW queue algorithm, e.g. by checking emptiness of the queue and allowing `deq` to return `NULL`, and affecting only the proof of absence of VWit violations without affecting the correctness arguments of the other properties.

Our violation conditions are arguably closer to what programmers have in mind when discussing concurrent data structures. Informal specifications written by programmers and bug reports do not mention that some method is not linearizable, but rather things like that values were dequeued in the wrong order.

[4] We say 'largely' because we have not yet automated the verification of the absence of VRepet violations, which requires a simple counting argument, nor the (admittedly trivial) proof that the HW queue is purely-blocking. We intend to implement these in the near future.

7 Conclusion

We have presented a new method for checking linearizability of concurrent queues. Instead of searching for the linearization points and doing a monolithic simulation proof, we verify four simple properties whose conjunction is equivalent to linearizability with respect to the atomic queue specification. By decomposing linearizability proofs in this way, we obtained a much simpler correctness proof of the Herlihy and Wing queue [8], and one which can be produced automatically.

We conjecture that our new property-oriented approach to linearizability proofs will be equally applicable to other kinds of concurrent shared data structures, such as stacks, sets, and maps. In the future, we would like to build tools that will automate this kind of reasoning for such data structures.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS'13. pp. 324–338. Springer (2013)
2. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV (2007)
3. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. ENTCS 137(2), 93–110 (2005)
4. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: FM'11. pp. 323–337. Springer (2011)
5. Doherty, S., Moir, M.: Nonblocking algorithms and backward simulation. In: Keidar, I. (ed.) DISC. LNCS, vol. 5805, pp. 274–288. Springer (2009)
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA '10. pp. 355–364. ACM (2010)
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc. (2008)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. pp. 463–492 (1990)
9. Hoffman, M., Shalev, O., Shavit, N.: The baskets queue. In: OPODIS'07. pp. 401–414. Springer (2007)
10. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. In: DISC '04. pp. 117–131. Springer-Berlin (2004)
11. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: FM '09. pp. 321–337. Springer (2009)
12. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA '05. pp. 253–262. ACM (2005)
13. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: CAV'12. pp. 243–259. Springer (2012)
14. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI. LNCS, vol. 5403, pp. 335–348. Springer (2009)
15. Vafeiadis, V.: Automatically proving linearizability. In: CAV'10. pp. 450–464. Springer (2010)
16. Vafeiadis, V.: RGSep action inference. In: Barthe, G., Hermenegildo, M.V. (eds.) VMCAI. LNCS, vol. 5944, pp. 345–361. Springer (2010)

Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency

Aaron Turon
MPI-SWS
turon@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Abstract

Modular programming and modular verification go hand in hand, but most existing logics for concurrency ignore two crucial forms of modularity: *higher-order functions*, which are essential for building reusable components, and *granularity abstraction*, a key technique for hiding the intricacies of fine-grained concurrent data structures from the clients of those data structures. In this paper, we present CaReSL, the first logic to support the use of granularity abstraction for modular verification of higher-order concurrent programs. After motivating the features of CaReSL through a variety of illustrative examples, we demonstrate its effectiveness by using it to tackle a significant case study: the first formal proof of (partial) correctness for Hendler *et al.*'s "flat combining" algorithm.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Contextual refinement; higher-order functions; fine-grained concurrency; separation logic; Kripke logical relations.

1. Introduction

Over the past decade, a number of Hoare logics have been developed to cope with the complexities of concurrent programming [20, 32, 7, 3, 18, 4, 27]. Unsurprisingly, the name of the game in these logics is improving support for *modular* reasoning along a variety of dimensions. Concurrent Abstract Predicates (CAP) [3], for example, utilizes a deft combination of *separation logic* [26] (for spatially-modular reasoning about resources and ownership), *rely-guarantee* [15] (for thread-modular reasoning in the presence of interference), and *abstract predicates* [21] (for hiding invariants about a module's private data structures from its clients).

At the same time, of course, the importance of modularity is not restricted to verification. Programmers use modularity in the *design* of their concurrent programs, precisely to enable reasoning about individual program components in relative isolation. And indeed, certain aspects of advanced concurrency logics, such as their aforementioned use of abstract predicates, are geared toward building proofs that reflect the data abstraction inherent in well-designed programs.

ICFP '13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09...$15.00.
http://dx.doi.org/10.1145/2500365.2500600

We contend, however, that existing concurrency logics are not exploiting the modular design of sophisticated concurrent programs to full effect. In particular, we observe that there are two crucial dimensions of modular concurrent programming that existing logics provide no way to leverage in the construction of proofs: namely, **higher-order functions** and **granularity abstraction**.

Higher-order concurrency Higher-order functional abstraction is of course one of the most basic hammers in the modern programmer's toolkit for writing reusable and modular code. Moreover, a number of concurrent programming patterns rely on it: work stealing [2], Concurrent ML-style events [25], concurrent iterators [16], parallel evaluation strategies [29], and monadic approaches to concurrent programming [10], just to name a few.

Yet, verification of higher-order concurrent programs remains a largely unexplored topic. To our knowledge, only a few existing logics can handle higher-order concurrent programs [27, 17, 14], but these logics are limited in other ways—in particular, they do not presently support verification of "fine-grained" concurrent ADTs. This leads us directly to the second limitation we observe of the state of the art, concerning granularity abstraction.

Granularity abstraction via contextual refinement An easy way to adapt a sequential mutable data structure for concurrent access is to employ *coarse-grained* synchronization: use a single global lock, and instrument each of the operations on the data structure so that they acquire the lock before they begin and release it after they complete. On the other hand, more sophisticated implementations of concurrent data structures employ *fine-grained* synchronization: they protect different parts of a data structure with different locks, or avoid locking altogether, so that threads can access disjoint pieces of the data structure in parallel.

There may seem at first glance to be a fundamental tradeoff here. Fine-grained synchronization enables parallelism, but makes the data structures that use it very tricky to reason about directly, due to their complex internal coordination between threads. Coarse-grained synchronization sequentializes access to the data structure, which is bad for parallelism but perfect for client-side reasoning, since it enables clients to reason about concurrent accesses as if each operation takes effect atomically.

Fortunately, modular programming comes to the rescue. In particular, so long as tricky uses of fine-grained synchronization are confined to the hidden state of a carefully crafted ADT, it is possible to prove that the fine-grained implementation of the ADT is a *contextual refinement* of some coarse-grained implementation. Contextual refinement means that, assuming clients only access the ADT through its abstract interface (so that the state really is hidden), every behavior that clients can observe of the fine-grained implementation is also observable of the coarse-grained one. Thus, clients can pretend, for the purpose of simplifying their own verification, that they are using the coarse-grained version, yet be sure that their code will still be correct when linked with the more efficient fine-grained version. This is what we call *granularity abstraction*. (Note: granularity abstraction is similar to *atomicity abstraction* [19], but is more general in that, as we will see in the iterator example later in this section, it applies even if the target of the abstraction is only *somewhat* coarse-grained.)

To illustrate this point more concretely, let us consider a simple motivating example of reasoning about Treiber's stack [28]. (We will in fact use this very example as part of a larger case study later in the paper.) Treiber's stack is a fine-grained implementation of a concurrent stack ADT. Instead of requiring concurrently executing push and pop operations to contend for a global lock on the whole stack (as a coarse-grained implementation would), Treiber's implementation allows them to race to access the head of the stack using compare-and-set (CAS). (The implementation of Treiber's stack is shown in Figure 9, and discussed in detail in §3.3.)

Now, the reader may expect that stacks should admit a canonical, principal specification (spec, for short), perhaps something like the following "precise" spec which tracks the exact contents s of the stack using the abstract predicate $\mathsf{Con}(s)$:

$$\{\mathsf{Con}(s)\}\ \mathsf{push}(x)\ \{\mathsf{Con}(x :: s)\}$$
$$\{\mathsf{Con}(s)\}\ \mathsf{pop}()\ \{\mathsf{ret}.\ (\mathsf{ret} = \mathsf{none} \land s = \mathsf{nil} \land \mathsf{Con}(s)) \lor (\exists x, s'.\mathsf{ret} = \mathsf{some}(x) \land s = x :: s' \land \mathsf{Con}(s'))\}$$

The trouble with using this spec in a concurrent setting is that knowledge about the exact contents of the stack is not stable under interference from other threads. As a result, some concurrency logics prohibit this spec altogether. Others permit the spec, but force the $\mathsf{Con}(s)$ predicate to be treated as a resource that only one thread can own at a time, thus effectively preventing any concurrent access to the stack and defeating the point of using Treiber's stack in the first place!

We want instead to capture the idea that the client threads of a data structure interact with it according to some (application-specific) protocol. Take, for example, the following "per-item" spec, which abstracts away from the LIFO nature of the stack and instead imposes an item-level protocol:

$$\begin{array}{ll} & \forall x.\ \{p(x)\}\ \mathsf{push}(x)\ \{\mathbf{true}\} \\ \land & \{\mathbf{true}\}\ \mathsf{pop}()\ \{\mathsf{ret}.\ \mathsf{ret} = \mathsf{none} \lor (\exists x.\ \mathsf{ret} = \mathsf{some}(x) \land p(x))\}\end{array}$$

Given an arbitrary predicate p of the client's choice, this per-item spec asserts that the stack contains only elements that satisfy the predicate p. It is pleasantly simple, and sufficient for the purposes of the case study we present later in the paper. It should be clear, however, that this "per-item" spec is far from a canonical or principal specification of stacks: the same spec would also be satisfied, for instance, by queues.

This brings us to our key point: different clients have different needs, and so we may want to verify the stack ADT against a range of different Hoare-style specs, but we do not want to have to re-verify *Treiber's implementation* each time. Ideally, we would like to modularly decompose the proof effort into two parts:

1. Prove that Treiber's implementation is a contextual refinement of a coarse-grained, lock-based implementation, which serves as a simple *reference implementation* of the stack ADT.
2. Use Hoare-style reasoning to verify that this reference implementation satisfies the various specs of interest to clients.

This decomposition engenders a clean separation of concerns, confining the difficulty of reasoning about Treiber's particular implementation[1] to the proof of refinement, and simplifying the verification of the stack ADT against different client specs.

The story we have just told is, *in principle*, nothing new. The ability to prove granularity abstraction via contextual refinement has been accepted in the concurrency literature as a useful correctness criterion for tricky concurrent data structures, precisely because of the modular decomposition of proof effort that it ought to facilitate [13, 8, 9]. Unfortunately, despite the utility of such a modular decomposition in theory, *no existing concurrency logic* actually supports it in practice. In particular, very few systems support proofs of contextual refinement at all, and those few that do support refinement proofs—such as the recent work of Liang and Feng [18]—do not provide a means of composing refinement with client-side Hoare-style verification in a unified logic.

Granularity abstraction for higher-order functions Although supporting granularity abstraction is already a challenge for first-order concurrent programs, it becomes even more interesting for higher-order concurrent programs.

Suppose, for instance, we wished to add a higher-order iterator, iter, to the concurrent stack ADT. (Such concurrent iterators are already commonplace [16].) Adding iter to Treiber's implementation is trivial: $\mathsf{iter}(f)$ will (without acquiring any lock) just read the current head pointer of the stack and apply f to each element of the stack accessible from it. But how do we specify the behavior of this iterator? What reference implementation should we use in proving granularity abstraction? Unlike for push and pop, it does not make sense for the reference implementation of $\mathsf{iter}(f)$ to be "maximally" coarse-grained (*i.e.,* to execute entirely within a critical section) because that will not correspond to the reality of the implementation we just described. In particular, the Treiber implementation does *not* freeze modifications to the stack while f is being applied to each element, so it does not contextually refine the maximally coarse-grained implementation of iteration.

Rather, what the Treiber iterator guarantees is that f will be applied to all the elements that were accessible from some node that was the head of the stack *at some point in time*. A clean way to specify this behavior is via a reference implementation that (1) acquires the lock, (2) takes a "snapshot" (*i.e.,* makes a copy) of the stack, (3) releases the lock, and (4) iterates f over the snapshot. (For the code of this reference implementation, see Figure 9.) What makes this reference implementation so intriguing is that it is only *somewhat* coarse-grained, yet as we will show later in the paper, it is nonetheless quite useful as a target for granularity abstraction.

This example demonstrates the flexibility of refinement, a flexibility which has not heretofore been tested, since no one has previously applied granularity abstraction to higher-order ADTs.

CaReSL: A logic for higher-order concurrency

In this paper, we present CaReSL (pronounced "carousel"), the first logic to support the use of granularity abstraction for modular verification of higher-order concurrent programs.

In providing both refinement and Hoare-style reasoning, CaReSL builds on ideas from two distinct lines of research:

- *Kripke logical relations* [22, 1, 5]. Logical relations are a fundamental technique for proving refinement in languages with *e.g.,* higher-order functions, polymorphism, and recursive types. Logical relations explain observable behavior in terms of the logical interpretation of each type. For example, two functions of type $\tau \to \tau'$ are logically related if giving them related inputs of type τ *implies* that they will produce related results of type τ'. *Kripke* logical relations are defined relative to a "possible world" that describes the relationship between the internal, hidden state of a program and that of its spec.
- *Concurrent separation logics* [20, 32, 3]. Separation logic is a reimagining of Hoare logic in which assertions are understood relative to some *portion* of the heap, with the separating conjunction $P * Q$ dividing up the heap between assertions P and Q. The motivation for separation is *local* reasoning: pre- and post-conditions need only mention the relevant portion of the

[1] Treiber's stack is actually one of the easiest fine-grained data structures to reason about, but our general line of argument applies equally well to more sophisticated implementations, such as the HSY elimination stack [12].

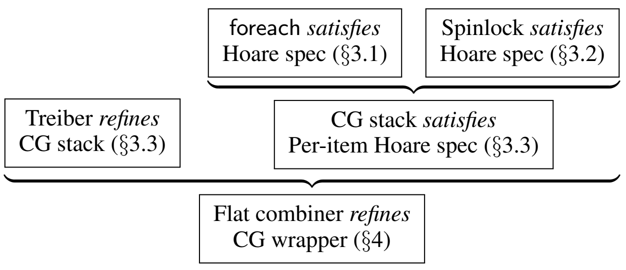

foreach *satisfies* Hoare spec (§3.1)
Spinlock *satisfies* Hoare spec (§3.2)
Treiber *refines* CG stack (§3.3)
CG stack *satisfies* Per-item Hoare spec (§3.3)
Flat combiner *refines* CG wrapper (§4)

Figure 1. The structure of the case study—and paper

heap, and the rest of the heap can be presumed invariant. Most concurrent separation logics also add some form of protocols (*e.g.,* rely-guarantee), which facilitate compositional reasoning by constraining the potential interference between threads operating concurrently on some *shared* portion of the heap.

Rather than a combination of these approaches, CaReSL is an attempt to unify them. So, for example, the logic does *not* include refinement as a primitive notion. Instead, refinement is derived from Hoare-style reasoning: to prove that e refines e', one merely proves a particular Hoare triple about e, allowing the tools of concurrent separation logic to be exploited in the proof of logical relatedness. In the other direction, CaReSL is a modal logic with the possible worlds of Kripke logical relations, which can in turn be used to express the shared-state protocols of concurrent separation logics. The unification yields surprising new techniques, such as the use of modal *necessity* ($\Box$) to hide resources that an ADT's client would otherwise have to thread through their reasoning (§3.2).[2]

The semantics of CaReSL is derived directly from the model of Turon *et al.* [31], who presented the first formal proofs of refinement for fine-grained data structures in an ML-like setting (although they did not use it to reason about higher-order examples). Compared with Turon *et al.*'s work, which was purely semantic, CaReSL provides a *syntactic* theory for carrying out refinement proofs at a much higher level of abstraction. This proof theory, in turn, is inspired by Dreyer *et al.*'s LADR [5], a modal logic for reasoning about contextual equivalence of (sequential) stateful ML programs. CaReSL exemplifies how the key ideas of LADR are just as relevant to—and arguably even more compelling when adapted to—the concurrent setting.

To demonstrate the effectiveness of CaReSL, we use it to tackle a significant case study: namely, the first formal proof of (partial) correctness for Hendler *et al.*'s "flat combining" algorithm [11]. Flat combining provides a generic way to turn a sequential ADT into a relatively efficient concurrent one, by having certain threads perform—in one fell swoop—the combined actions requested by a whole bunch of other threads. The flat combining algorithm is interesting not only because it itself is a rather sophisticated higher-order function, but also because it is modularly assembled from other higher-order components, including a fine-grained stack with concurrent iterator. We therefore take the opportunity to verify flat combining in a modular way that mirrors the modular structure of its implementation.

Figure 1 illustrates the high-level structure of our proof, which involves an intertwined application of refinement and Hoare-style verification. For example, we prove that Treiber's stack refines the coarse-grained (CG) reference implementation of stacks; we then use Hoare-style reasoning to prove that the reference implementation satisfies the per-item spec; and finally we rely on the per-item spec in the proof that the flat combining algorithm refines an even higher-level abstraction. Every component of the case study involves higher-order code and therefore higher-order specifications.

[2] Previously, such hiding required the subtle *anti-frame* rule [24].

Syntax

Val $\quad v ::= () \mid \textbf{true} \mid \textbf{false} \mid (v, v) \mid \textbf{inj}_i\ v \mid \textbf{rec}\ f(x).e \mid \Lambda.e \mid \ell$

Exp $\quad e ::= v \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid e\ e \mid e\ \_ \mid (e, e) \mid \textbf{prj}_i\ e \mid \textbf{inj}_i\ e$
$\quad \mid\ \textbf{case}(e, \textbf{inj}_1\ x \Rightarrow e, \textbf{inj}_2\ y \Rightarrow e) \mid \textbf{new}\ e \mid \textbf{get}\ e \mid e := e$
$\quad \mid\ \textbf{CAS}(e, e, e) \mid \textbf{newLcl} \mid \textbf{getLcl}(e) \mid \textbf{setLcl}(e, e) \mid \textbf{fork}\ e$

CType $\quad \sigma ::= \textbf{B} \mid \textbf{ref}\ \tau \mid \textbf{refLcl}\ \tau \mid \mu\alpha.\sigma$

Type $\quad \tau ::= \sigma \mid \textbf{1} \mid \alpha \mid \tau \times \tau \mid \tau + \tau \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \tau \to \tau$

ECtx $\quad K ::= [\,] \mid \textbf{if}\ K\ \textbf{then}\ e\ \textbf{else}\ e \mid K\ e \mid v\ K \mid \cdots$

Typing contexts

$$\Delta ::= \cdot \mid \Delta, \alpha \qquad \Gamma ::= \cdot \mid \Gamma, x : \tau \qquad \Omega ::= \Delta; \Gamma$$

Well-typed expressions $\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{\Omega \vdash e : \textbf{ref}\ \sigma \qquad \Omega \vdash e_o : \sigma \qquad \Omega \vdash e_n : \sigma}{\Omega \vdash \textbf{CAS}(e, e_o, e_n) : \textbf{B}} \qquad \frac{\Omega \vdash e : \forall\alpha.\tau}{\Omega \vdash e\ \_ : \tau[\tau'/\alpha]}$$

Machine configurations

TLV $\quad L \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \text{Value} \qquad$ TPool $\quad \mathcal{E} \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \text{Expression}$

Heaps $\quad h \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} (\text{Val} \cup \text{TLV}) \qquad$ Config $\quad \varsigma ::= h; \mathcal{E}$

Thread-local lookup $\quad \lfloor L \rfloor(i) \triangleq \begin{cases} \text{none} & i \notin \text{dom}(L) \\ \text{some}(L(i)) & \text{otherwise} \end{cases}$

Pure reduction $\boxed{e \overset{\text{pure}}{\to} e'}$

$$(\textbf{rec}\ f(x).e)\ v \overset{\text{pure}}{\to} e[\textbf{rec}\ f(x).e/f, v/x] \qquad (\Lambda.e)\ \_ \overset{\text{pure}}{\to} e$$

Per-thread reduction $\boxed{h; e \overset{i}{\to} h'; e'}$

$$\begin{array}{rcll}
h; e & \overset{i}{\to} & h; e' & \text{when } e \overset{\text{pure}}{\to} e' \\
h; \textbf{newLcl} & \overset{i}{\to} & h \uplus [\ell \mapsto \emptyset]; \ell & \\
h \uplus [\ell \mapsto L]; \textbf{setLcl}(\ell, v) & \overset{i}{\to} & h \uplus [\ell \mapsto L[i := v]]; () & \\
h; \textbf{getLcl}(\ell) & \overset{i}{\to} & h; v & \text{when } h(\ell) = L,\ \lfloor L \rfloor(i) = v \\
h; \textbf{CAS}(\ell, v_o, v_n) & \overset{i}{\to} & h; \textbf{false} & \text{when } h(\ell) \neq v_o \\
h \uplus [\ell \mapsto v_o]; \textbf{CAS}(\ell, v_o, v_n) & \overset{i}{\to} & h \uplus [\ell \mapsto v_n]; \textbf{true} &
\end{array}$$

General reduction $\boxed{h; \mathcal{E} \to h'; \mathcal{E}'}$

$$\frac{h; e \overset{i}{\to} h'; e'}{h; \mathcal{E} \uplus [i \mapsto K[e]] \to h'; \mathcal{E} \uplus [i \mapsto K[e']]}$$

$$h; \mathcal{E} \uplus [i \mapsto K[\textbf{fork}\ e]] \to h; \mathcal{E} \uplus [i \mapsto K[()]] \uplus [j \mapsto e]$$

Figure 2. The calculus: $F^{\mu!}$ with **fork**, **CAS**, and thread-local refs

2. The programming model

The programming language for our program logic is sketched in Figure 2; the full details are in the appendix [30]. It has a standard core: the polymorphic lambda calculus with sums, products, references, and equi-recursive types. We elide all type annotations in terms, but polymorphism is nevertheless introduced and eliminated by explicit type abstraction ($\Lambda.e$) and application ($e\ \_$). To this standard core, we add concurrency (through a **fork** construct), atomic compare-and-set (**CAS**) and thread-local references (type **refLcl** τ).

While normal references provide shared state through which threads can communicate, thread-local references allow multiple threads to access disjoint values in memory (a different one for each thread) using a common location. Formally, each thread-local reference has an associated *thread ID-indexed* table L, which is initially empty. Thus for **refLcl** τ, the **getLcl** operation returns an "option" of type $1+\tau$, reflecting whether the thread-local reference has been initialized for the thread that invoked it.[3] Thread-local references are commonly used when a single data structure needs mutable, thread-local storage for an unbounded number of threads; the flat combining algorithm presented in §4 embodies this technique.

We define a small-step, call-by-value operational semantics, using evaluation contexts K to specify a left-to-right evaluation order within each thread. The $\overset{\text{pure}}{\to}$ relation gives the reductions that do not interact with the heap, while $\xrightarrow{i}$ gives general, single-thread reductions for the thread with ID i. These reductions are then lifted to general reduction $\to$ of *machine configurations* ς, which consist of a heap h together with an ID-indexed pool $\mathcal{E}$ of threads.

The type system imposes two important restrictions. First, recursive types $\mu\alpha.\tau$ must be *guarded*, meaning that all free occurrences of α in τ must appear under a non-μ type constructor. Second, because **CAS** exposes physical equality comparison of word-sized values at the hardware level, it can only be applied to references of *comparable type* σ, which we take to be base types and locations.

If $\Omega \vdash e_{\mathrm{I}} : \tau$ and $\Omega \vdash e_{\mathrm{S}} : \tau$, we say e_{I} *contextually refines* e_{S}, written $\Omega \models e_{\mathrm{I}} \preceq e_{\mathrm{S}} : \tau$, if, for every pair of thread IDs i and j,

$$\begin{aligned}\forall C : (\Omega,\tau) \rightsquigarrow (\emptyset, \mathsf{B}).\ \forall n.\forall \mathcal{E}_{\mathrm{I}}.\ \ &\emptyset; [i \mapsto C[e_{\mathrm{I}}]] \to^* h_{\mathrm{I}}; [i \mapsto v] \uplus \mathcal{E}_{\mathrm{I}} \\ \implies \exists \mathcal{E}_{\mathrm{S}}.\ \ &\emptyset; [j \mapsto C[e_{\mathrm{S}}]] \to^* h_{\mathrm{S}}; [j \mapsto v] \uplus \mathcal{E}_{\mathrm{S}}\end{aligned}$$

where $C : (\Omega,\tau) \rightsquigarrow (\Omega',\tau')$ is the standard typing judgment for contexts (guaranteeing that $\Omega \vdash e : \tau$ implies $\Omega' \vdash C[e] : \tau'$). This definition resembles the standard one for sequential languages, but take note that only the termination behavior of the main thread is observed; as with most languages in practice, we assume that once the main thread has finished, any forked threads are aborted.

3. CaReSL

While CaReSL ultimately provides mixed refinement and Hoare-style reasoning about higher-order concurrent programs, it is easiest to understand by first considering less powerful "sublogics", and then gradually incorporating more powerful features:

- We begin with a core logic that provides separation-logic reasoning for sequential (but higher-order) programs (§3.1). In this core logic, both Hoare triples and heap assertions live in a single syntactic class: they are propositions. Thus propositions include *e.g.,* implications between Hoare triples, which are ubiquitous when reasoning about higher-order code. Propositions also include first- and second-order quantification, guarded recursion, separating conjunction, and modal necessitation—all standard concepts whose interplay we explain in §3.1.
- We then add concurrency, leading to a concurrent separation logic for higher-order programs (§3.2). The new concept needed to deal with concurrency is that of a *protocol* governing some shared region of the heap. We express protocols through transition systems whose states provide an abstraction of the heap region's actual state. Threads can gain or lose *roles* in the protocol, which determine the transitions those (and other) threads are permitted to take. Traditional rely/guarantee reasoning can be recovered as a particular, restricted way of using protocols.
- Finally, we show how to reason about refinement, yielding a **C**oncurrent **a**nd **Re**fined **S**eparation **L**ogic (**CaReSL**) for higher-order programs (§3.3). As explained in the introduction, CaReSL does not simply add refinement as a primitive notion on top of a concurrent separation logic. Instead, CaReSL treats refinement as a derived notion, expressed as a particular Hoare-style specification. In order to do so, however, CaReSL does require one further extension: the notion of "spec resources", first introduced by Turon *et al.* [31], which allow pieces of a program configuration (heap and threads) to be manipulated as logical resources within Hoare-style proofs.

While CaReSL is the main contribution of the paper, its sublogics are of independent interest. We explain them by way of idiomatic higher-order examples, each of which serves as a component of the case study in §4.

3.1 Core logic: Sequential higher-order programs

To motivate the basic ingredients of core CaReSL, we begin with the quintessential higher-order stateful combinator, foreach:

$$\begin{array}{l}\mathsf{foreach}\ :\ \forall\alpha.(\alpha \to \mathbf{1}) \to \mathsf{list}(\alpha) \to \mathbf{1} \qquad \mathsf{list}(\alpha) \triangleq \mu\beta.\mathbf{1} + (\alpha \times \beta) \\ \mathsf{foreach} \triangleq \Lambda.\ \lambda f.\ \textbf{rec}\ \mathsf{loop}(l).\ \textbf{case}\ l\ \textbf{of}\ \mathsf{none} \qquad\quad \Rightarrow () \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \mathsf{some}(x,n) \Rightarrow f(x);\ \mathsf{loop}(n)\end{array}$$

A specification of foreach should explain that foreach $f\ l$ "lifts" the imperative behavior of f (which works on elements) to l (a list of elements). But to do so, it needs to *quantify* over the unknown behavior of an unknown function in a way that can be lifted to lists.

One possibility is to characterize f by means of a Hoare triple, while ultimately quantifying over the pre- and post-conditions of that triple. Suppose that $p(x)$ and $q(x)$ are predicates on values x, and that r is an arbitrary proposition, such that

$$\forall x.\ \{p(x) * r\}\ f(x)\ \{q(x) * r\}$$

The idea is that when f is applied to an element x of the list, it may assume both that $p(x)$ holds and that an invariant r holds of a disjoint portion of the heap. If $f(x)$ transforms $p(x)$ to $q(x)$ while maintaining the invariant, then foreach lifts that behavior to an entire list—regardless of what the predicates actually are. To capture the assumption that $p(x)$ holds of each element of a list l, we need something like the following recursive predicate:[4]

$$\mathrm{Map}_p(l) \triangleq l = \mathsf{none} \vee (\exists x, n.\ l = \mathsf{some}(x,n) * p(x) * \mathrm{Map}_p(n))$$

Note that, thanks to the use of the separating conjunction $*$, for each x in l there is a *disjoint region* of the heap satisfying $p(x)$.

These ideas lead to the following spec for foreach:

$$\begin{aligned}&\forall p,q,r.\ \forall f.\ (\forall x.\{p(x) * r\}\ f(x)\ \{q(x) * r\}) \\ &\qquad\qquad \Rightarrow \forall l.\ \Big\{\mathrm{Map}_p(l) * r\Big\}\ \mathsf{foreach}\ \_\ f\ l\ \Big\{\mathrm{Map}_q(l) * r\Big\}\end{aligned}$$

Formalizing this sketch will require a logic with a number of basic features: we need to be able to mix Hoare triples with other kinds of connectives (*e.g.,* implication), to quantify over both values (*e.g.,* f and x) and predicates (*e.g.,* p and q), and to define recursive predicates (*e.g.,* Map). With these goals in mind, we now explore CaReSL's design more systematically.

Syntax Figure 3 presents core CaReSL—the fragment of the logic appropriate for reasoning about sequential, higher-order code. CaReSL is a multi-sorted second-order logic, meaning that its syntax is stratified into the following three layers.

First, there are *terms* M, N, which come in a variety of *sorts* Σ, including the values and expressions of the language, as well as thread-local storage (with operations $\lfloor - \rfloor$, $\uplus$ and $:=$). The judgment $\mathcal{X} \vdash M : \Sigma$ gives the sort of a term (where $\mathcal{X}$ is a variable context giving the sorts of term and predicate variables).

Second, there are *propositions* P, Q, R, which include the connectives of multi-sorted first-order logic (*e.g.,* $\forall X \in \Sigma.P$) and second-order logic (*e.g.,* $\exists p \in \mathbb{P}(\Sigma).P$) with their standard meanings. The judgment $\mathcal{X} \vdash P : \mathbb{B}$ asserts that the proposition P is well-sorted.

Third, there are *predicates* φ, ψ, which are just propositions parameterized by a term: a predicate of sort $\mathbb{P}(\Sigma)$ can be introduced by $(X \in \Sigma).P$, which binds an unknown term X of sort Σ in P,

[3] Throughout, we let $\mathsf{none} \triangleq \mathbf{inj}_1\ ()$ and $\mathsf{some}(e) \triangleq \mathbf{inj}_2\ e$.

[4] Throughout, names of functions and types are written in lower case sans-serif, while predicate names are Capitalized Serif.

Syntax

Terms $M ::= X \mid (M, M) \mid n \mid e \mid M \uplus M \mid \lfloor M \rfloor(M) \mid M[M := M]$

Sorts $\Sigma ::= \mathbf{1} \mid \Sigma \times \Sigma \mid \text{Nat} \mid \text{Val} \mid \text{Exp} \mid \text{AtomicExp} \mid \text{LclStorage}$

Preds $\varphi ::= p \mid (X \in \Sigma).P \mid \mu p \in \mathbb{P}(\Sigma).\varphi$

Propositions P ::=

$\text{True} \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \exists X \in \Sigma.P \mid \forall X \in \Sigma.P \mid \triangleright P \mid$
$\varphi(M) \mid \exists p \in \mathbb{P}(\Sigma).P \mid \forall p \in \mathbb{P}(\Sigma).P \mid P * P \mid M \hookrightarrow_{\mathrm{I}} M \mid A$

Necessary (“always”) propositions A ::=

$\Box P \mid M \overset{\text{pure}}{\to} M \mid M{=}M \mid (\!|P|\!)\ M \Rrightarrow_{\mathrm{IS}} M\ (\!|\varphi|\!) \mid \{P\}\ M \Rrightarrow M\ \{\varphi\}$

Contexts

$\mathcal{X} ::= \cdot \mid \mathcal{X}, X : \Sigma \mid \mathcal{X}, p : \mathbb{P}(\Sigma)$ $\qquad \mathcal{P} ::= \cdot \mid \mathcal{P}, P$ $\qquad \mathcal{C} ::= \mathcal{X}; \mathcal{P}$

Well-sorted terms $\boxed{\mathcal{X} \vdash M : \Sigma}$

$$\frac{}{\mathcal{X}, X : \Sigma \vdash X : \Sigma} \qquad \frac{\mathcal{X} \vdash M : \text{LclStorage} \qquad \mathcal{X} \vdash N : \text{Nat}}{\mathcal{X} \vdash \lfloor M \rfloor(N) : \text{Val}} \qquad \ldots$$

Well-sorted props. $\boxed{\mathcal{X} \vdash P : \mathbb{B}}$ ***Well-sorted preds.*** $\boxed{\mathcal{X} \vdash \varphi : \mathbb{P}(\Sigma)}$

Figure 3. Core CaReSL: Syntax

and eliminated by $\varphi(M)$ (also written $M \in \varphi$) which substitutes M for the parameter of φ. Because sorts include unit and products, predicates can express *relations* of arbitrary arity. The judgment $\mathcal{X} \vdash \varphi : \mathbb{P}(\Sigma)$ asserts that φ is a well-sorted predicate over terms of sort Σ.

In general, term variables are written X. But to avoid cluttering our rules and proofs with sort annotations, we use variables x, y, z for sort Val and i, j, k, ℓ for sort Nat. We abuse notation, writing *e.g.,* v, e or L to stand for a term of sort Val, Exp or LclStorage, respectively.

In addition to this standard logical setup, CaReSL adopts key connectives from separation logic. In general, these connectives will refer to a variety of “resources” that will be introduced as we go along. In *core* CaReSL, however, the only resource is the heap. The *points-to* assertion $\ell \hookrightarrow_{\mathrm{I}} v$ holds of any heap containing a location ℓ that points to the value v. (Ignore the I subscript for now; we will return to it in §3.3). The *separating conjunction* $P * Q$ holds if the currently-owned resources (here, a portion of the heap) can be split into two disjoint parts satisfying P and Q respectively.

While the truth of some propositions (*e.g.,* $\ell \hookrightarrow_{\mathrm{I}} v$) is *contingent* on the presence of certain resources, others (*e.g.,* $M = N$) are *necessary*: if they hold, they do so regardless of the currently-owned resources, and therefore will continue to hold in any future state. (Such propositions are called “pure” in separation logic parlance.) The syntactic subcategory A of necessary propositions includes claims about term equality, about the operational semantics ($M \overset{\text{pure}}{\to} N$), and Hoare triples. Arbitrary propositions can be made necessary via the $\Box$ (“always”) modality, where $\Box P$ holds if P holds for all possible resources. As we will see shortly, necessary propositions play by special rules: they can move freely through Hoare triples and separating conjunctions.

Ultimately, CaReSL distinguishes between triples about general expressions e and those about *atomic* expressions a (which execute in a single step). Since this distinction is motivated by concurrency, we postpone its explanation to §3.2. We include the distinction syntactically in core CaReSL, but it can be safely ignored for now.

Atomic expressions a have the following grammar:

$\textbf{new}\ v \mid \textbf{get}\ v \mid v\ := \ v \mid \textbf{CAS}(v, v, v) \mid \textbf{newLcl} \mid \textbf{getLcl}(v) \mid \textbf{setLcl}(v, v)$

The propositions for atomic triples[5] $(\!|P|\!)\ i \Rrightarrow_{\mathrm{IS}} a\ (\!|\varphi|\!)$ and general triples $\{P\}\ i \Rrightarrow e\ \{\varphi\}$ are both parameterized by a *thread ID* i; the expression may access thread-local storage, in which case its behavior is ID-dependent. (When the thread ID doesn't matter, we

[5] Again, ignore the IS subscript until §3.3.

Logical axioms and rules

$$\frac{\mathcal{X}; A \vdash P}{\mathcal{X}; \mathcal{P}, A \vdash \Box P}\ \Box\mathrm{I} \qquad \frac{\mathcal{C} \vdash P}{\mathcal{C} \vdash \triangleright P}\ \textsc{Mono} \qquad \frac{\mathcal{C}, \triangleright P \vdash P}{\mathcal{C} \vdash P}\ \textsc{Löb}$$

$$\begin{array}{rcl} \Box(P \Rightarrow Q) & \Rightarrow & \Box P \Rightarrow \Box Q \\ A & \Rightarrow & \Box A \\ \Box P & \Rightarrow & P \end{array} \qquad \begin{array}{c} A * P \Leftrightarrow A \wedge P \\ \text{True} * P \Leftrightarrow P \\ \triangleright(P * Q) \Leftrightarrow \triangleright P * \triangleright Q \end{array}$$

$$\frac{\mathcal{C}, p : \mathbb{P}(\Sigma) \vdash P}{\mathcal{C} \vdash \forall p \in \mathbb{P}(\Sigma).\ P}\ \forall_2\mathrm{I} \qquad \begin{array}{r} M \in (X \in \Sigma).P \Leftrightarrow P[M/X] \\ M \in (\mu p.\varphi) \Leftrightarrow M \in \varphi[\mu p.\varphi/p] \end{array}$$

Figure 4. Core CaReSL: Selected logical axioms and rules

write $\{P\}\ e\ \{\varphi\}$ as short for $\forall i.\ \{P\}\ i \Rrightarrow e\ \{\varphi\}$.) In addition, since we are working with an expression language (as opposed to a command language), postconditions are *predicates* over the return value of the expression, rather than simply propositions about the final state of the heap. But when an expression returns unit, we often abuse notation and write a proposition instead.

In order to support recursive assertions, the logic includes *guarded recursion* ($\mu p \in \mathbb{P}(\Sigma).\ P$), which entails the following tradeoff. On the one hand, guarded recursion allows occurrences of the recursive predicate p to appear negatively, which is crucial for modelling recursive types (§3.3) but is usually prohibited for lack of monotonicity. On the other hand, the recursion is “guarded” in that references to p in P must appear under the $\triangleright$ modality. A proposition $\triangleright Q$ represents the present knowledge that Q holds *later*, *i.e.,* after at least one step of computation. Guarded recursion supports a coinductive style of reasoning: to prove P one can assume $\triangleright P$, but this assumption is only useful after at least one step of computation. As we explain in §5, our use of guarded recursion descends from a line of work on *step-indexed* logical relations, but the interaction with Hoare triples is a novelty of CaReSL.

Proof rules The main judgment of CaReSL is written $\mathcal{C} \vdash P$, where $\mathcal{C} = \mathcal{X}; \mathcal{P}$ is a *combined* context of annotated term/predicate variables $\mathcal{X}$ and propositional assumptions $\mathcal{P}$. The meaning is the usual one: for any way of instantiating the variables in $\mathcal{X}$, if the hypotheses $\mathcal{P}$ are true for a given resource (*i.e.,* for the moment, a given heap), then P is true of the same resource. We implicitly assume $\mathcal{X} \vdash Q : \mathbb{B}$ for every proposition Q in $\mathcal{P}$ and likewise that $\mathcal{X} \vdash P : \mathbb{B}$ holds.

A few basic logical axioms and proof rules for core CaReSL are shown in Figure 4. Axioms hold in an arbitrary well-sorted context. The axioms include all the standard ones for a multi-sorted second-order logic (we show only $\forall_2$I), as well as several characterizing separating conjunction and the $\Box$ and $\triangleright$ modalities. We'll just mention the highlights:

- Because True is a unit for separating conjunction (and every proposition implies True), propositions are *affine*: we can “throw away” resources, because $(P * Q) \Rightarrow (\text{True} * Q) \Rightarrow Q$.
- The two conjunctions $*$ and $\wedge$ are identical if at least one of their operands is a necessary proposition. Consequently, necessary propositions can be “freely copied”: $A \Rightarrow A * A$.
- The Löb rule provides a coinductive reasoning principle: to prove P, you may assume P—but only under the $\triangleright$ modality, which guards use of the assumption until at least one step of computation has taken place. On the other hand, Mono says that any proposition can be *weakened* to one that is guarded. (Both Mono and Löb are inherited from LADR [5].) We will momentarily see how $\triangleright$ is eliminated in Hoare-style reasoning.
- The $\triangleright$ modality distributes over $*$, as it does with most other connectives, except implication and recursion.

Atomic Hoare logic (where $\textsc{is} ::= \textsc{i} \mid \textsc{s}$ as explained in §3.3)

$$\begin{array}{rll}
\llparenthesis \mathrm{True} \rrparenthesis & i \Mapsto_{\textsc{is}} \textbf{new } v & \llparenthesis \mathsf{ret}.\ \mathsf{ret} \hookrightarrow_{\textsc{is}} v \rrparenthesis \\
\llparenthesis v \hookrightarrow_{\textsc{is}} v' \rrparenthesis & i \Mapsto_{\textsc{is}} \textbf{get } v & \llparenthesis \mathsf{ret}.\ \mathsf{ret} = v' * v \hookrightarrow_{\textsc{is}} v' \rrparenthesis \\
\llparenthesis v \hookrightarrow_{\textsc{is}} - \rrparenthesis & i \Mapsto_{\textsc{is}} v \ := \ v' & \llparenthesis \mathsf{ret}.\ \mathsf{ret} = () * v \hookrightarrow_{\textsc{is}} v' \rrparenthesis \\
\llparenthesis \mathrm{True} \rrparenthesis & i \Mapsto_{\textsc{is}} \textbf{newLcl} & \llparenthesis \mathsf{ret}.\ \mathsf{ret} \hookrightarrow_{\textsc{is}} \emptyset \rrparenthesis \\
\llparenthesis v \hookrightarrow_{\textsc{is}} L \rrparenthesis & i \Mapsto_{\textsc{is}} \textbf{getLcl}(v) & \llparenthesis \mathsf{ret}.\ \mathsf{ret} = \lfloor L \rfloor (i) * v \hookrightarrow_{\textsc{is}} L \rrparenthesis \\
\llparenthesis v \hookrightarrow_{\textsc{is}} L \rrparenthesis & i \Mapsto_{\textsc{is}} \textbf{setLcl}(v, v') & \llparenthesis \mathsf{ret}.\ \mathsf{ret} = () * v \hookrightarrow_{\textsc{is}} L[i := v'] \rrparenthesis \\
\llparenthesis v \hookrightarrow_{\textsc{is}} v' \rrparenthesis & i \Mapsto_{\textsc{is}} & \llparenthesis \mathsf{ret}.\ (v' = v_o * \mathsf{ret} = \textbf{true} * v \hookrightarrow_{\textsc{is}} v_n) \\
& \quad \textbf{CAS}(v, v_o, v_n) & \quad \vee \ (v' \neq v_o * \mathsf{ret} = \textbf{false} * v \hookrightarrow_{\textsc{is}} v') \rrparenthesis
\end{array}$$

$$\frac{\mathcal{X}; P \vdash P' \qquad \mathcal{X}; \mathcal{P} \vdash \llparenthesis P' \rrparenthesis\ i \Mapsto_{\textsc{is}} a\ \llparenthesis x.\ Q' \rrparenthesis \qquad \mathcal{X}, x; Q' \vdash Q}{\mathcal{X}; \mathcal{P} \vdash \llparenthesis P \rrparenthesis\ i \Mapsto_{\textsc{is}} a\ \llparenthesis x.\ Q \rrparenthesis}\ \textsc{ACsq}$$

$$\frac{\mathcal{C} \vdash \llparenthesis P \rrparenthesis\ i \Mapsto_{\textsc{is}} a\ \llparenthesis x.\ Q \rrparenthesis}{\mathcal{C} \vdash \llparenthesis P * R \rrparenthesis\ i \Mapsto_{\textsc{is}} a\ \llparenthesis x.\ Q * R \rrparenthesis}\ \textsc{AFrame} \qquad (\textsc{ADisj}, \textsc{AEx} \text{ elided})$$

General Hoare logic

$$\frac{\mathcal{C} \vdash \llparenthesis P \rrparenthesis\ i \Mapsto_{\textsc{i}} a\ \llparenthesis Q \rrparenthesis}{\mathcal{C} \vdash \{\triangleright P\}\ i \Mapsto a\ \{Q\}}\ \textsc{Private} \qquad \frac{\mathcal{C} \vdash e \overset{\text{pure}}{\to} e' \quad \mathcal{C} \vdash \{P\}\ i \Mapsto e'\ \{\varphi\}}{\mathcal{C} \vdash \{\triangleright P\}\ i \Mapsto e\ \{\varphi\}}\ \textsc{Pure}$$

$$\frac{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ Q\} \qquad \mathcal{C}, x \vdash \{Q\}\ i \Mapsto K[x]\ \{R\}}{\mathcal{C} \vdash \{P\}\ i \Mapsto K[e]\ \{R\}}\ \textsc{Bind}$$

$$\mathcal{C} \vdash \{\mathrm{True}\}\ i \Mapsto v\ \{x.\ x = v\}\ \textsc{Ret} \qquad (\textsc{Csq}, \textsc{Frame}, \textsc{Disj}, \textsc{Ex} \text{ elided})$$

$$\frac{\mathcal{C} \vdash A \quad \mathcal{C} \vdash \{P * A\}\ i \Mapsto e\ \{\varphi\}}{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{\varphi\}}\ \textsc{AIn} \qquad \frac{\mathcal{C}, A \vdash \{P\}\ i \Mapsto e\ \{\varphi\}}{\mathcal{C} \vdash \{P * A\}\ i \Mapsto e\ \{\varphi\}}\ \textsc{AOut}$$

Derivable rules

$$\frac{\mathcal{C}, f, \forall x. \{\triangleright P\}\ i \Mapsto f\ x\ \{\varphi\} \vdash \forall x.\ \{P\}\ i \Mapsto e\ \{\varphi\}}{\mathcal{C} \vdash \forall x.\ \{\triangleright P\}\ i \Mapsto (\textbf{rec } f(x).e)\ x\ \{\varphi\}}\ \textsc{Rec}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ \exists y.\ (x = \textbf{inj}_1\ y * \triangleright Q_1) \vee (x = \textbf{inj}_2\ y * \triangleright Q_2)\} \\ \forall k \in \{1, 2\} : \ \mathcal{C}, x, y \vdash \{x = \textbf{inj}_k\ y * Q_k\}\ i \Mapsto e_k\ \{\varphi\}\end{array}}{\mathcal{C} \vdash \{P\}\ i \Mapsto \textbf{case}(e, \textbf{inj}_1\ y \Rightarrow e_1, \textbf{inj}_2\ y \Rightarrow e_2)\ \{\varphi\}}$$

Figure 5. Core CaReSL: Hoare logic

The rules for atomic triples (Figure 5) are formulated in the standard style of separation logic. They transcribe the operational semantics of atomic expressions, mentioning only the part of the heap relevant to the expression. Atomic Hoare logic supports the usual rules—consequence, framing, disjunction, $\exists$-elimination—with one important exception: it does not support a sequencing rule, since a sequence of atomic expressions is not atomic.

All atomic expressions take exactly one step to execute, and in so doing allow us to peel off a layer of the $\triangleright$ modality. To cut down on clutter, the precondition in an atomic triple is *implicitly* understood as being under one $\triangleright$ modality. The rule Private, which lifts an atomic triple to a general one, makes this implicit assumption explicit. (In core CaReSL, *all* resources are private; §3.2 adds shared resources.) To handle pure reduction steps (like β-reduction), the Pure rule appeals directly to the operational semantics using the necessary proposition $e \overset{\text{pure}}{\to} e'$. The rest of the rules for general Hoare triples are mostly standard; we show only the nonstandard rules in Figure 5.

In an expression language, the monadic nature of Hoare logic becomes visible: the Bind rule replaces the usual rule of sequencing, while Ret is used to inject a value into a Hoare triple.

We also have a rule allowing necessary propositions to move freely from the proof context into preconditions (AIn), and vice versa (AOut). In general, any contingent assumptions like $x \hookrightarrow_{\textsc{i}} 3$ need to be given explicitly in the precondition of a Hoare triple, because the truth of such statements can change over time; the triple says that it is only usable at times when its precondition holds. But in the specific case of necessary propositions, we can do better: we know that if such a proposition happens to be true now, it will be true forever, and so it does not need to be given explicitly in the precondition. As we will see in §3.2, these rules will allow us to completely hide pieces of state that are known to always obey a certain protocol.

Finally, using Löb and Pure, together with standard Hoare rules, we can derive specialized rules for constructs like recursive functions and pattern matching—the two derived rules in Figure 5.

Verifying **foreach** Having seen core CaReSL in detail, we can now return to the foreach example. First, we need to rewrite our sketch of the specification more formally. The Map predicate needs to employ guarded recursion:

$$\begin{aligned}\mathrm{Map}_\varphi \triangleq \mu m \in \mathbb{P}(\mathrm{Val}).\ (l \in \mathrm{Val}).\\ l = \mathsf{none} \vee (\exists x, n.\ l = \mathsf{some}(x, n) * \varphi(x) * \triangleright m(n))\end{aligned}$$

while the foreach spec should be annotated with sorts:

$$\begin{aligned}&\forall p, q \in \mathbb{P}(\mathrm{Val}). \forall r \in \mathbb{P}(\mathbf{1}).\ \forall f.\ (\forall x. \{p(x) * r\}\ f(x)\ \{q(x) * r\}) \\ &\quad \Rightarrow \quad \forall l.\ \Big\{\mathrm{Map}_p(l) * r\Big\}\ \mathsf{foreach}\ \_\ f\ l\ \Big\{\mathrm{Map}_q(l) * r\Big\}\end{aligned}$$

To prove foreach correct, we use a kind of Hoare "proof outline", annotating each program point with a proposition:

$$\mathsf{foreach} \triangleq \Lambda.\ \lambda f.\ \textbf{rec}\ \mathsf{loop}(l).$$

$$\boxed{\begin{array}{ll}\textbf{Prop context:} & \textbf{Variables: } f, i, p, q, r, l, \mathsf{loop} \\ \multicolumn{2}{l}{\forall x.\ \{p(x) * r\}\ i \Mapsto f(x)\ \{q(x) * r\}} \\ \multicolumn{2}{l}{\forall n.\ \Big\{\triangleright(\mathrm{Map}_p(n) * r)\Big\}\ i \Mapsto \mathsf{loop}\ n\ \Big\{\mathrm{Map}_q(n) * r\Big\}}\end{array}}$$

$$\begin{array}{l}
\{\mathrm{Map}_p(l) * r\} \\
\textbf{case}\ l\ \textbf{of}\ \mathsf{none} \Rightarrow \{l = \mathsf{none} * \mathrm{Map}_p(l) * r\}\ \ ()\ \ \{\mathrm{Map}_q(l) * r\} \\
\qquad\quad \mid \mathsf{some}(x, n) \Rightarrow \{l = \mathsf{some}(x, n) * \mathrm{Map}_p(l) * r\} \\
\qquad\qquad\qquad\qquad\qquad \{l = \mathsf{some}(x, n) * p(x) * \triangleright \mathrm{Map}_p(n) * r\} \\
\qquad\qquad\qquad\qquad\qquad f(x); \\
\qquad\qquad\qquad\qquad\qquad \{l = \mathsf{some}(x, n) * q(x) * \triangleright \mathrm{Map}_p(n) * r\} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{loop}(n) \\
\qquad\qquad\qquad\qquad\qquad \{l = \mathsf{some}(x, n) * q(x) * \mathrm{Map}_q(n) * r\} \\
\qquad\qquad\qquad\qquad\qquad \{\mathrm{Map}_q(l) * r\}
\end{array}$$

Proof outlines implicitly apply the Hoare logic rule corresponding to each language construct: for recursive functions and pattern matching we use the derived rules shown earlier; for an atomic expression we use the corresponding axiom; when going under a Λ or λx we use the Pure rule and prove a triple about the function as applied to _ or x respectively. The frame rule is applied implicitly as needed, while uses of consequence are shown by writing a sequence of assertions (each one implying the next). We do not explicitly write the thread ID in a proof outline, but it is always clear from context (and, for our examples, always the variable i).

We supplement traditional proof outlines with boxed *context assertions* spelling out an extension to both the variable context $\mathcal{X}$ and proposition context $\mathcal{P}$. Extending the context with new variables introduces a universal quantification (using *e.g.,* the rule $\forall_2$I in Figure 4), while adding propositions introduces an implication. We use the AIn rule implicitly to bring necessary consequences of the proof context into the Hoare-style outline.

The proof for foreach is quite straightforward: the initial case analysis on the input list allows us to expand the definition of the Map predicate, which for the nonempty case gives us the necessary knowledge to execute f on an element of the list; note the heavy use of the frame rule to add invariant propositions. The use of $\triangleright$ in Map is neatly dispatched by the use of the Rec rule for foreach.

3.2 Adding concurrency: Protocols for shared state

To reason about concurrency, we need to reason about the protocols governing shared (and often hidden) state. Take, for example, the following higher-order spinlock:

$$
\begin{array}{lcl}
\mathsf{tryAcq} & \triangleq & \lambda x.\ \mathbf{CAS}(x, \mathbf{false}, \mathbf{true}) \\
\mathsf{acq} & \triangleq & \mathbf{rec}\ \mathsf{loop}(x).\ \mathbf{if}\ \mathsf{tryAcq}(x)\ \mathbf{then}\ ()\ \mathbf{else}\ \mathsf{loop}(x) \\
\mathsf{rel} & \triangleq & \lambda x.\ x\ :=\ \mathbf{false} \\
\mathsf{mkSync} & : & \mathbf{1} \to \forall\alpha.\forall\beta.(\alpha \to \beta) \to (\alpha \to \beta) \\
\mathsf{mkSync} & \triangleq & \lambda().\ \mathbf{let}\ \mathsf{lock} = \mathbf{new\ false\ in} \\
& & \qquad \Lambda.\Lambda.\lambda f.\lambda x.\ \mathsf{acq}(\mathsf{lock}); \mathbf{let}\ z = f(x)\ \mathbf{in}\ \mathsf{rel}(\mathsf{lock});\ z
\end{array}
$$

Each invocation of mkSync creates a new *wrapper* that closes over a fresh, hidden lock. The wrapper can then be used to add simple synchronization to an arbitrary function. There are, of course, a variety of ways to use synchronization, but a particularly common one is to protect access to some shared resource characterized by an invariant p—an idea that leads to the following specification:

$$
\begin{array}{l}
\forall p \in \mathbb{P}(\mathbf{1}).\ \{p\}\ \mathsf{mkSync}\ ()\ \left\{s.\ \mathrm{Syncer}_p(s)\right\} \\
\quad \textit{where}\ \ \mathrm{Syncer}_p \triangleq (s \in \mathrm{Val}).\ \ \Box\forall f, x, q, r.\ \ \{p * q\}\ f\ x\ \{z.\ p * r(z)\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow\ \{q\}\ s\ \_\ \_\ f\ x\ \{z.\ r(z)\}
\end{array}
$$

The spec says that when mkSync() is executed, the client can choose some invariant resource p, giving up control over the resource in exchange for a synchronization wrapper s. When s is later applied to a function f, it provides f with exclusive access to the resource p (seemingly out of thin air), which f can use however it pleases, so long as the invariant is reestablished on completion.

Intuitively, the reason that mkSync satisfies its specification is that the lock itself is *hidden*: all access to it is mediated through the wrapper s, which the client can only apply to invariant-preserving functions. Hiding enables mkSync to maintain an internal *protocol* in which, whenever the lock is free, the invariant p holds. To express this protocol, as well as the more sophisticated protocols needed for fine-grained concurrency (§3.3, §4), CaReSL provides a syntactic account of the semantic protocols of Turon *et al.* [31].

Protocols A protocol π governs a shared resource abstractly, by means of a set of protocol states (S) equipped with a transition relation ($\rightsquigarrow$). Each state $s \in S$ has an associated proposition $\varphi(s)$ giving its concrete interpretation in terms of *e.g.,* heap resources. The idea is then that any thread is allowed to update the shared resource, so long as at each atomic step those concrete updates correspond to a permitted abstract transition.

In general, although all threads must abide by a given protocol, not all of them play the same role within it. For example, a protocol governing a lock might have two states, Locked and Unlocked, with transitions in both directions, but we usually want to allow only the thread that acquired the lock to be allowed to release it (as is the case in the mkSync example). To allow threads to take on or relinquish roles dynamically, protocols employ *tokens* that individual threads may own. By taking a transition, a thread may "earn" a token from the protocol; conversely, certain transitions require a thread to "pay" by giving up a previously-earned token. For the locking protocol, we use a single token *Lock*:

$$\begin{tikzcd}
\boxed{\mathsf{Unlocked}; \mathit{Lock}} \arrow[r, bend left] & \boxed{\mathsf{Locked}} \arrow[l, bend left]
\end{tikzcd}$$

In the Unlocked state, the protocol itself owns the *Lock*, which we show by annotating the state with the token. A thread that transitions from Unlocked to Locked then earns (takes ownership of) the *Lock*. Conversely, to transition back to the Unlocked state, a thread must transfer the *Lock* back to the protocol—a move only possible for the thread that owns the *Lock*.

Formally, a protocol $\pi = (S, \rightsquigarrow, \mathcal{T}, \varphi)$ is given by a transition system (S and $\rightsquigarrow$), a function $\mathcal{T}$ giving the set of tokens owned by the protocol at each state, and a predicate φ on states giving their interpretations. To be able to talk about states and tokens in the logic, we add sorts State and TokSet, for which we will use the metavariables s and T respectively; see Figure 6. We leave the grammar of terms for these sorts open-ended, implicitly extending it as needed for particular transition systems.[6] Thus, we can give an interpretation $\mathrm{LockInterp}_p$ for the lock protocol that is appropriate for an instance of mkSync protecting an invariant p:

Protocols

$$
\begin{array}{ll}
\text{Protocol}\ \ \pi ::= (S, \rightsquigarrow, \mathcal{T}, \varphi) \quad \text{where} & S \subseteq \text{State}, \ \ \rightsquigarrow\ \subseteq S \times S, \\
& \mathcal{T} \in S \to \text{TokSet}, \ \ \varphi\ \text{token-pure}
\end{array}
$$

$$
\begin{array}{rcl}
(s; T) \rightsquigarrow_\pi (s'; T') & \triangleq & s\ (\pi.\rightsquigarrow)\ s' \ \wedge\ \pi.\mathcal{T}(s) \uplus T = \pi.\mathcal{T}(s') \uplus T' \\
\text{frame}_\pi(s; T) & \triangleq & (s;\ \text{AllTokens} - (\pi.\mathcal{T}(s) \uplus T))
\end{array}
$$

$$
\begin{array}{rcl}
(s; T)\ \sqsubseteq^{\text{guar}}_\pi\ (s'; T') & \triangleq & (s; T) \rightsquigarrow^*_\pi (s'; T') \\
(s; T)\ \sqsubseteq^{\text{rely}}_\pi\ (s'; T') & \triangleq & \text{frame}_\pi(s; T) \rightsquigarrow^*_\pi \text{frame}_\pi(s'; T')
\end{array}
$$

Syntax

$$
\begin{array}{ll}
\text{Sort} & \Sigma ::= \cdots \mid \text{State} \mid \text{TokSet} \\
\text{Prop} & P ::= \cdots \mid \boxed{M}^{M}_{\pi} \mid \mathit{Tid}(M) \\
\text{AbProp} & A ::= \cdots \mid M \sqsubseteq^{\text{rely}}_\pi M \mid M \sqsubseteq^{\text{guar}}_\pi M \mid \text{TokPure}(P)
\end{array}
$$

Hoare logic (*where* $\pi[\![b]\!] \triangleq \exists s.b = (s, -) \wedge \pi.\varphi(s)$)

$$
\frac{\mathcal{C} \vdash \{P\}\ i \Rrightarrow e\ \{x.\ Q * \pi[\![b]\!]\}}{\mathcal{C} \vdash \{P\}\ i \Rrightarrow e\ \left\{x.\ Q * \exists n.\ \boxed{b}^{n}_{\pi}\right\}}\ \textsc{NewIsl}
$$

$$
\frac{\mathcal{C} \vdash \forall b \overset{\text{rely}}{\sqsupseteq_\pi} b_0.\ (\!|\pi[\![b]\!] * P|\!)\ i \Rrightarrow_{\mathrm{I}} a\ (\!|x.\ \exists b' \overset{\text{guar}}{\sqsupseteq_\pi} b.\ \pi[\![b']\!] * Q|\!)}{\mathcal{C} \vdash \left\{\boxed{b_0}^{n}_{\pi} * \triangleright P\right\}\ i \Rrightarrow a\ \left\{x.\ \exists b'.\ \boxed{b'}^{n}_{\pi} * Q\right\}}\ \textsc{UpdIsl}
$$

$$
\frac{\mathcal{C} \vdash \{P * \mathit{Tid}(j)\}\ j \Rrightarrow e\ \{\mathsf{ret.\ ret} = ()\}}{\mathcal{C} \vdash \{P\}\ i \Rrightarrow \mathbf{fork}\ e\ \{\mathsf{ret.\ ret} = ()\}}\ \textsc{Fork}
$$

Logical axioms and rules

$$
\begin{array}{l}
\boxed{s_1; T_1}^{n}_{\pi} * \boxed{s_2; T_2}^{n}_{\pi} \ \Leftrightarrow \qquad\qquad\qquad\qquad\qquad\qquad \textsc{SplitIsl} \\
\qquad \exists s.\ \boxed{s; T_1 \uplus T_2}^{n}_{\pi} \wedge (s; T_1) \overset{\text{rely}}{\sqsupseteq_\pi} (s_1; T_1) \wedge (s; T_2) \overset{\text{rely}}{\sqsupseteq_\pi} (s_2; T_2)
\end{array}
$$

Figure 6. CaReSL: Incorporating concurrency and protocols

$$\mathrm{LockInterp}_p \triangleq (s \in \mathsf{State}).\ s = \mathsf{Locked} \vee (s = \mathsf{Unlocked} * p)$$

The combination of transition systems and tokens gives rise to *token-sensitive transitions*. A transition from state/tokens $(s; T)$ to state/tokens $(s'; T')$ is permitted by π, written $(s; T) \rightsquigarrow_\pi (s'; T')$, so long as the *law of token conservation* holds: the disjoint union of the thread's tokens and the protocols tokens $T \uplus \pi.\mathcal{T}(s)$ before the transition must be the same as the disjoint union $T' \uplus \pi.\mathcal{T}(s')$ afterwards.[7] Token-sensitive transitions constrain both what a thread can do (the *guarantee* moves $\sqsubseteq^{\text{guar}}_\pi$ enabled by its tokens) and what its environment can do (the *rely* moves $\sqsubseteq^{\text{rely}}_\pi$ enabled by the tokens owned by other threads). See Figure 6.

Island assertions In CaReSL, all resources are either privately owned by a thread, or else governed by a shared protocol. When a heap assertion like $x \hookrightarrow_{\mathrm{I}} 3$ appears in the pre- or postcondition of a triple, it is understood as asserting private ownership over the corresponding portion of the heap; no other thread is allowed to access it. Thus the rules of core CaReSL are immediately sound in a concurrent setting: there is no interference to account for.

To talk about shared resources, CaReSL includes *island assertions* $\boxed{b}^{n}_{\pi}$ (similar to region assertions in RGSep/CAP). As the name suggests, each island is an independent region of the heap governed by its own laws (the protocol π). The number n is the "name" of the island, which is used to distinguish between multiple islands with the same protocol; we often leave off the name when

[6] To be completely formal, we could allow each transition system to come equipped with its own explicit grammar of states and tokens.

[7] We use notation like $\pi.\mathcal{T}$ to extract named components from a tuple.

it is existentially quantified. The term b (of sort State $\times$ TokSet) asserts private ownership of a set of tokens,[8] and acts as a "rely-lower bound" on the state of the protocol: the current state of the protocol is some $b' \sqsupseteq^{\text{rely}}_{\pi} b$. Thus, in CaReSL *every assertion about shared resources is automatically stable under interference*.

While a thread's island assertions cannot be invalidated by a change its environment makes, they *can* be invalidated by a change the thread itself makes. For example, if a thread owns *Lock* in Locked state of the mkSync protocol, the environment cannot change the state at all—but the thread itself can move to the Unlocked state. There is, however, a special class of island assertions which are "completely" stable, *i.e.,* that act as necessary propositions: island assertions that do not claim any tokens. To understand why, consider that when no tokens are owned, the $\sqsubseteq^{\text{rely}}_{\pi}$ relation *degenerates* to the underlying transition system of the protocol, which means it contains all possible moves that any thread can make. Formally, we have $\boxed{(M;\emptyset)}^{n}_{\pi} \Rightarrow \Box\, \boxed{(M;\emptyset)}^{n}_{\pi}$.

When island assertions *do* claim ownership over tokens (a form of resource), they can be meaningfully combined by separating conjunction; see the SPLITISL rule in Figure 6, which takes into account the fact that the assertions give only rely-lower bounds.[9]

There are two Hoare logic rules for working with islands (Figure 6). The rule NEWISL creates a new island starting with an initial state and token set bound b; the resources necessary to satisfy the protocol's initial state must be present as private resources, and afterwards will be shared. (The shorthand $\pi[\![b]\!]$ just gives the interpretation at the state in b.) Once an island is established, the only way to access the shared resources it governs is through the UPDISL rule, which can only be applied to an *atomic* expression. Starting from an initial bound b_0, the atomic expression a might be (instantaneously) executed in any rely-reachable b; for each such state, the expression is granted exclusive access to the shared resource, but in its single atomic step, it must make a guarantee move in the protocol. This rule reveals the important semantic difference between atomic and general triples: through the UPDISL rule, atomic triples gain access to the concrete resources owned by shared islands, while general triples only have access to island assertions.

Thread creation In a protocol, threads gain and lose roles (tokens) by making deliberate moves within the protocol. But there is also a role that every thread plays *automatically*: the role of being a thread with a certain ID. To support protocols that use thread IDs (such as the one in §4), CaReSL builds in a proposition *Tid*(j) that asserts the existence of a thread j, and acts as an *uncopyable* resource: *Tid*(j) $*$ *Tid*(j) $\Leftrightarrow$ False. The resource is introduced by the FORK rule, which also allows the parent thread to transfer an arbitrary P to the newly-forked thread. (The parent can keep resources for itself via the FRAME rule.) The trivial postconditions in FORK may seem alarming, but they reflect the fact that the language has no **join** mechanism: the *only* form of communication between threads is shared state, mediated by a shared protocol.

***Verifying* mkSync** The interpretations $\pi.\varphi(s)$ of protocol states in CaReSL inherit a limitation from their semantic treatment by Turon *et al.* [31]: they must be token-pure, *i.e.,* they cannot assert ownership of island tokens. The necessary proposition TokPure(P) can be used to assert that, for example, an unknown proposition is token-pure and thus safe to use in an interpretation, and we need such a restriction on p in mkSync:

$$\begin{aligned} &\forall p \in \mathbb{P}(\mathbf{1}).\ \text{TokPure}(p) \Rightarrow \{p\}\ i \Rrightarrow \mathsf{mkSync}\ ()\ \Big\{s.\ \text{Syncer}_p(s)\Big\} \\ &\quad \textit{where}\ \ \text{Syncer}_p \triangleq (s \in \text{Val}).\ \ \Box\forall f,x,q,r.\ \ \{p * q\}\ f\ x\ \{z.\ p * r(z)\} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow\ \ \{q\}\ s\ \_\ \_\ f\ x\ \{z.\ r(z)\} \end{aligned}$$

[8] An island assertion is only satisfied if the asserted token set is disjoint from the tokens owned by the protocol at the asserted state.

[9] We assume that terms M include disjoint union on token sets.

The hidden protocol LockProt$_p$ for mkSync just puts together the pieces we have already seen:

$$\text{LockProt}_p \triangleq (\{\mathsf{Locked}, \mathsf{Unlocked}\}, \leadsto, \mathcal{T}, \text{LockInterp}_p)$$

where $\leadsto$ and $\mathcal{T}$ are given by the transition system for locking shown above.

The high-level proof outline for mkSync is straightforward:

$$\begin{array}{l} \lambda().\ \boxed{\textbf{Prop context: } \text{TokPure}(p) \qquad \textbf{Variables: } p} \\ \quad \{p\}\ \ \textbf{let}\ \mathsf{lock} = \textbf{new false}\ \ \{p * \mathsf{lock} \hookrightarrow_{\text{I}} \textbf{false}\} \\ \quad \{\exists n.\ \boxed{\mathsf{Unlocked}; \emptyset}^{n}_{\text{LockProt}_p}\} \\ \quad \Lambda.\Lambda.\lambda f.\lambda x. \\ \qquad \boxed{\begin{array}{ll} \textbf{Prop context:} & \textbf{Variables: } f, x, q, r \\ \Box\ \boxed{\mathsf{Unlocked}; \emptyset}_{\text{LockProt}_p} & \{p * q\}\ f\ x\ \{z.\ p * r(z)\} \end{array}} \\ \qquad \begin{array}{ll} \{q\} & \{\boxed{\mathsf{Unlocked}; \emptyset}_{\text{LockProt}_p} * q\} \\ \mathsf{acq(lock)}; & \{\boxed{\mathsf{Locked}; \mathit{Lock}}_{\text{LockProt}_p} * p * q\} \\ \textbf{let}\ z = f(x) & \{\boxed{\mathsf{Locked}; \mathit{Lock}}_{\text{LockProt}_p} * p * r(z)\} \\ \mathsf{rel(lock)}; & \{\boxed{\mathsf{Unlocked}; \emptyset}_{\text{LockProt}_p} * r(z)\} \\ z & \{z.\ r(z)\} \end{array} \end{array}$$

After allocating the hidden lock, we move it into a fresh island using NEWISL, and then move that island assertion (wrapped with $\Box$) into the proof context (using AOUT) before reasoning about the returned wrapper function. In this way the protocol is completely hidden from the client, yet available to the closure we return to the client—mirroring the fact that lock is hidden from the client but available in the closure. Since the spec Syncer$_p(s)$ is a necessary proposition, the client may move the spec into its proof context, and thus freely use the synchronization wrapper s without threading *any* assertion about it through its proof. (Previous logics, like CAP, require at least that the client thread through an abstract predicate standing for the lock.)

When verifying the closure, we begin with a precondition q of the client's choice, but then apply AIN to load the hidden island assertion into the precondition. The subsequent lines use the locking protocol to acquire the resource p, execute the client's function f, and then return p to the protocol. The tokens in the island assertions, which say what the *thread* owns at each point, complement those (in $\pi.\mathcal{T}$) labelling the corresponding states in the protocol.

The triples for acq and rel must ultimately be proved by appeal to the UPDISL rule. For acq, the *bound* on the protocol is just $(\mathsf{Unlocked}; \emptyset)$, which means that the *actual* state might be either Locked or Unlocked. The **CAS** within acq will return:

- **true** if successful; the state must have been Unlocked. We make a guarantee move to $(\mathsf{Locked}; \mathit{Lock})$, gaining the token.
- **false** if it fails; the state must have been Locked. We do not change the state, and the acq function retries by calling loop.

On the other hand, for rel, the bound $(\mathsf{Locked}; \mathit{Lock})$ means that the protocol *must* be in state Locked: the environment cannot move to Unlocked because it does not own *Lock*. But since rel *does* own *Lock*, it can simply update the lock to **false**, corresponding to a guarantee move to the Unlocked state in the protocol.

3.3 Adding refinement: Reasoning about specification code

At this point, we have seen the fragment of CaReSL providing Hoare-style specs and proofs for higher-order concurrent programs, as exemplified (in a simple way) by the mkSync example. This section explains the other major component of the logic: higher-order granularity abstraction, exemplified (in a simple way) by the Treiber stack with iterator.

Syntax

$$\Sigma ::= \cdots \mid \text{EvalCtx} \qquad P ::= \cdots \mid M \hookrightarrow_{\mathrm{S}} M \mid M \Mapsto_{\mathrm{S}} M$$
$$M ::= \cdots \mid K \mid M[M] \qquad A ::= \cdots \mid P \rightarrow_{\mathrm{S}} P \mid M \bowtie M$$

Spec rewriting

$$\frac{\mathcal{C} \vdash e \overset{\text{pure}}{\to} e'}{\mathcal{C} \vdash (P * j \Mapsto_{\mathrm{S}} K[e]) \rightarrow_{\mathrm{S}} (P * j \Mapsto_{\mathrm{S}} K[e'])}\ \textsc{SPure}$$

$$\frac{\mathcal{C} \vdash \llparenthesis P \rrparenthesis\ j \Mapsto_{\mathrm{S}} a\ \llparenthesis x.\ Q \rrparenthesis}{\mathcal{C} \vdash (P * j \Mapsto_{\mathrm{S}} K[a]) \rightarrow_{\mathrm{S}} (\exists x.\ Q * j \Mapsto_{\mathrm{S}} K[x])}\ \textsc{SPrim}$$

$$\mathcal{C} \vdash (j \Mapsto_{\mathrm{S}} K[\textbf{fork}\ e]) \rightarrow_{\mathrm{S}} (j \Mapsto_{\mathrm{S}} K[()] * k \Mapsto_{\mathrm{S}} e)\ \textsc{SFork}$$

Hoare logic (AExecSpec elided)

$$\frac{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ Q\} \qquad \mathcal{C} \vdash Q \rightarrow_{\mathrm{S}} R}{\mathcal{C} \vdash \{P\}\ i \Mapsto e\ \{x.\ R\}}\ \textsc{ExecSpec}$$

$$\frac{\mathcal{C}, j \bowtie j' \vdash \left\{P * \mathit{Tid}(j) * j' \Mapsto_{\mathrm{S}} e_{\mathrm{S}}\right\}\ j \Mapsto e_{\mathrm{I}}\ \{x.\ x = ()\}}{\mathcal{C} \vdash \left\{P * i' \Mapsto_{\mathrm{S}} K[\textbf{fork}\ e_{\mathrm{S}}]\right\}\ i \Mapsto \textbf{fork}\ e_{\mathrm{I}}\ \left\{i' \Mapsto_{\mathrm{S}} K[()]\right\}}\ \textsc{ForkS}$$

Figure 7. CaReSL: Incorporating spec resources

Spec resources While it is possible to formulate a relational version of Hoare logic (with Hoare *quadruples* [33]), or to develop special-purpose logics for refinement [5], our goal with CaReSL is to support *both* standard Hoare-style reasoning and refinement reasoning in a single unified logic. In particular, we want a treatment of refinement that re-uses as much Hoare-style reasoning as possible. To this end, we adapt the idea of *specification resources*, first proposed by Turon *et al.* [31] as a way of proving refinement when threads engage in "cooperation" (a point we return to in §4). We will show how spec resources make it possible to state and prove refinements as an *entirely* derived concept on top of the Hoare logic we have already built up, in particular allowing protocols and triples to serve double-duty when reasoning about spec code.

To prove that e_{I} refines e_{S}, one must (intuitively) show that every behavior observable of e_{I} is observable of e_{S} as well. Our strategy is to encode these proof obligations into certain Hoare triples about the execution of e_{I}, but with pre- and postconditions instrumented with pieces of the corresponding spec state—both heap and code—which we treat as resources in CaReSL. These spec resources are entirely a fiction of the logic: they do not reflect anything about the *physical* state of e_{I}, but they must be used through logical rules that enforce the operational semantics for e_{S}.

There are two basic spec resource assertions: the *(spec) points-to* assertion $\ell \hookrightarrow_{\mathrm{S}} v$, which claims ownership of a fragment of the spec heap containing a location ℓ that points to the value v, and the *spec thread* assertion $i \Mapsto_{\mathrm{S}} e$, which claims ownership of a spec thread with ID i executing expression e. The separating conjunction $P * Q$ works in the usual way with these resources, dividing up the spec heap and thread pool between the propositions P and Q. We also add a final sort, EvalCtx, and terms K and $M[N]$ for expressing and combining them, which makes it possible to *abstract* over the evaluation context for some spec thread. (By convention, the variable κ is always of sort EvalCtx.)

In addition, CaReSL provides a necessary proposition $P \rightarrow_{\mathrm{S}} Q$, which says that the spec resources owned by P can, according to the operational semantics, take a step to those owned by Q. All other resources must be left invariant. The rule SPure lifts the pure stepping relation from the operational semantics directly. The rule SPrim, on the other hand, *re-uses* atomic triples to support reasoning about atomic spec expressions. (The subscript IS in the laws of atomic triples allows them to be used in *either* implementation mode I or spec mode S.) The ExecSpec Hoare rule (and identical AExecSpec rule for atomic triples) allows the specification to be

Relating expressions

$$(e_{\mathrm{I}}, e_{\mathrm{S}}) \downarrow \varphi \ \triangleq\ \forall (i \bowtie j), \kappa.$$
$$\{\mathit{Tid}(i) * j \Mapsto_{\mathrm{S}} \kappa[e_{\mathrm{S}}]\}\ i \Mapsto e_{\mathrm{I}}\ \{x_{\mathrm{I}}.\ \exists x_{\mathrm{S}}.\ \varphi(x_{\mathrm{I}}, x_{\mathrm{S}}) * \mathit{Tid}(i) * j \Mapsto_{\mathrm{S}} \kappa[x_{\mathrm{S}}]\}$$

Relating values

$$\llbracket \mathbf{1} \rrbracket \triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ x_{\mathrm{I}} = x_{\mathrm{S}} = () \qquad \llbracket \mu\alpha.\tau \rrbracket \triangleq \mu\alpha.\llbracket \tau \rrbracket$$
$$\llbracket \mathbf{B} \rrbracket \triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ (x_{\mathrm{I}} = x_{\mathrm{S}} = \textbf{true}) \vee (x_{\mathrm{I}} = x_{\mathrm{S}} = \textbf{false}) \qquad \llbracket \alpha \rrbracket \triangleq \alpha$$

$$\begin{aligned}
\llbracket \tau_1 \times \tau_2 \rrbracket &\triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ (\textbf{prj}_1\ x_{\mathrm{I}}, \textbf{prj}_1\ x_{\mathrm{S}}) \downarrow \llbracket \tau_1 \rrbracket \wedge (\textbf{prj}_2\ x_{\mathrm{I}}, \textbf{prj}_2\ x_{\mathrm{S}}) \downarrow \llbracket \tau_2 \rrbracket \\
\llbracket \tau_1 + \tau_2 \rrbracket &\triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\quad (\rhd \exists (y_{\mathrm{I}}, y_{\mathrm{S}}) \in \llbracket \tau_1 \rrbracket.\ x_{\mathrm{I}} = \textbf{inj}_1\ y_{\mathrm{I}} \wedge x_{\mathrm{S}} = \textbf{inj}_1\ y_{\mathrm{S}}) \\
&\qquad \vee (\rhd \exists (y_{\mathrm{I}}, y_{\mathrm{S}}) \in \llbracket \tau_2 \rrbracket.\ x_{\mathrm{I}} = \textbf{inj}_2\ y_{\mathrm{I}} \wedge x_{\mathrm{S}} = \textbf{inj}_2\ y_{\mathrm{S}}) \\
\llbracket \tau \to \tau' \rrbracket &\triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ \Box \forall y_{\mathrm{I}}, y_{\mathrm{S}}.(\rhd (y_{\mathrm{I}}, y_{\mathrm{S}}) \in \llbracket \tau \rrbracket) \Rightarrow (x_{\mathrm{I}}\ y_{\mathrm{I}}, x_{\mathrm{S}}\ y_{\mathrm{S}}) \downarrow \llbracket \tau' \rrbracket \\
\llbracket \forall \alpha.\tau \rrbracket &\triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ \Box \forall \alpha.\text{Type}(\alpha) \Rightarrow (x_{\mathrm{I}}\ \_, x_{\mathrm{S}}\ \_) \downarrow \llbracket \tau \rrbracket \\
\llbracket \textbf{ref}\ \tau \rrbracket &\triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ \text{Inv}(\exists (y_{\mathrm{I}}, y_{\mathrm{S}}) \in \llbracket \tau \rrbracket.\ x_{\mathrm{I}} \hookrightarrow_{\mathrm{I}} y_{\mathrm{I}} * x_{\mathrm{S}} \hookrightarrow_{\mathrm{S}} y_{\mathrm{S}}) \\
\llbracket \textbf{refLcl}\ \tau \rrbracket &\triangleq (x_{\mathrm{I}}, x_{\mathrm{S}}).\ \text{Inv}\begin{pmatrix} \exists L_{\mathrm{I}}, L_{\mathrm{S}}.\ x_{\mathrm{I}} \hookrightarrow_{\mathrm{I}} L_{\mathrm{I}} * x_{\mathrm{S}} \hookrightarrow_{\mathrm{S}} L_{\mathrm{S}} * \\ \quad \forall (i \bowtie j).\ (\lfloor L_{\mathrm{I}} \rfloor (i), \lfloor L_{\mathrm{S}} \rfloor (j)) \in \llbracket \mathbf{1} + \tau \rrbracket \end{pmatrix}
\end{aligned}$$

Invariant protocols $\quad \text{Inv}(P) \triangleq \exists n.\ \boxed{0; \emptyset}^{n}_{(\{0\}, \emptyset, [0 \mapsto \emptyset], (s).P)}$

Type interpretations $\quad \text{Type}(\varphi) \triangleq \Box \forall (x_{\mathrm{I}}, x_{\mathrm{S}}) \in \varphi.\ \Box (x_{\mathrm{I}}, x_{\mathrm{S}}) \in \varphi$

Logical relatedness* (i.e., *refinement)

$$\overline{\alpha}; \overline{x : \tau} \vdash e_{\mathrm{I}} \preceq e_{\mathrm{S}} : \tau \ \triangleq$$
$$\overline{\alpha}, \overline{x_{\mathrm{I}}, x_{\mathrm{S}}}; \overline{\text{Type}(\alpha)}, \overline{(x_{\mathrm{I}}, x_{\mathrm{S}}) \in \llbracket \tau \rrbracket} \vdash (e_{\mathrm{I}}\overline{[x_{\mathrm{I}}/x]}, e_{\mathrm{S}}\overline{[x_{\mathrm{S}}/x]}) \downarrow \llbracket \tau \rrbracket$$

Figure 8. Encoding refinement in CaReSL

"executed" within any postcondition, *i.e.,* at any point in a proof. Since ExecSpec can be applied repeatedly, a single step of some implementation code—*e.g.,* an atomic expression—can correspond to an arbitrary number of spec steps.

Putting these pieces together, a triple like

$$\{j \Mapsto_{\mathrm{S}} e_{\mathrm{S}}\}\ i \Mapsto e_{\mathrm{I}}\ \{x_{\mathrm{I}}.\ \exists x_{\mathrm{S}}.\ j \Mapsto_{\mathrm{S}} x_{\mathrm{S}} * \varphi(x_{\mathrm{I}}, x_{\mathrm{S}})\}$$

says that if e_{I} produces some value x_{I}, then e_{S} can be executed to produce some value x_{S} such that $\varphi(x_{\mathrm{I}}, x_{\mathrm{S}})$ holds—exactly the kind of observational claim we set out to make.

Encoding refinement To give a full account of refinement, we also need to ensure that updates to public reference cells by the implementation are matched in lock-step by updates by the spec, which we will do by imposing a protocol governing public reference cells. And to account for thread-local references, we must also track a *correspondence* between implementation and specification thread IDs, which allows us to state invariants connecting the storage on either side. The (necessary) assertion $i \bowtie j$ asserts that the implementation thread i and spec thread j belong to the bijective correspondence, *i.e.,* $(i \bowtie j) \wedge (i' \bowtie j') \Rightarrow (i = i' \Leftrightarrow j = j')$. Correspondences are introduced using the rule ForkS, a variant of Fork that jointly creates fresh implementation and spec threads.

And that's all: using these ingredients, we can *encode* refinement by simply writing down a particular predicate in CaReSL.

The encoded predicate expresses a logical relation—a variant of the relation given semantically by Turon *et al.* [31]—following the approach first laid out by Plotkin and Abadi [23]. We define the relation in stages (see Figure 8).

First, we have the proposition $(e_{\mathrm{I}}, e_{\mathrm{S}}) \downarrow \varphi$, which is a more general version of the triple we suggested above: it includes assumptions about thread IDs, and permits compositional reasoning about specs by quantifying over an unknown evaluation context κ. The expressions do not begin with private ownership of any heap resources because, when proving refinement, we must assume that any pre-existing state is shared with the context. As we will see in a moment, this shared state is governed by an extremely liberal protocol; all we can assume about the context is that it is well-typed.

Second, we have the binary predicate $\llbracket \tau \rrbracket$, which is satisfied by $(v_{\mathrm{I}}, v_{\mathrm{S}})$ when the observations a context can make of v_{I} can

```
$\mathsf{stack}_S$  :  $\forall\alpha.\ (\alpha \to \mathbf{1}) \times (\mathbf{1} \to (1+\alpha)) \times ((\alpha \to \mathbf{1}) \to \mathbf{1})$
$\mathsf{stack}_S$  $\triangleq$  $\Lambda.$ let sync = mkSync(), $\mathsf{hd}_S$ = new (none)
  let push = $\lambda x.$ $\mathsf{hd}_S$ := some($x$, get $\mathsf{hd}_S$)
  let pop  = $\lambda()$. case get $\mathsf{hd}_S$ of none          $\Rightarrow$ none
                                    | some($x, n$) $\Rightarrow$ $\mathsf{hd}_S$ := $n$; some($x$)
  let snap = sync _ _ ($\lambda()$.get $\mathsf{hd}_S$)
  let iter  = $\lambda f.$ foreach _ $f$ (snap())
  in (sync _ _ push, sync _ _ pop, iter)

$\mathsf{stack}_I \triangleq \Lambda.$ let $\mathsf{hd}_I$ = new nil                  where nil $\triangleq$ new none,
  let push = rec try($x$).                                cons $e$ $\triangleq$ new some($e$)
    let $c$ = get $\mathsf{hd}_I$, $n$ = cons($x, c$)
    in if CAS($\mathsf{hd}_I, c, n$) then () else try($x$),
  let pop = rec try().
    let $c$ = get $\mathsf{hd}_I$ in case get $c$ of
        none $\Rightarrow$ none
      | some($d, n$) $\Rightarrow$ if CAS($\mathsf{hd}_I, c, n$) then some($d$) else try()
  let iter = $\lambda f.$  let rec loop($c$) = case get $c$ of
                          none $\Rightarrow$ ()   |   some($d, n$) $\Rightarrow$ $f(d)$; loop($n$)
                   in loop(get $\mathsf{hd}_I$)
  in (push, pop, iter)
```

Figure 9. Coarse- and fine-grained stacks, with iterators

also be made of v_S—the heart of the logical relation. It is defined by induction on the structure of τ, since the ways a value can be observed depend on its type:

For **ground types**, the context can observe the exact value, so only equal values are in the interpretation.

For **product types**, the context can project both sides of the product, so two values are related iff each of their projections are related; similarly for **sums**. The context can observe **functions** only by applying them, so one function is related to another iff, when applied to related values, they produce related results.

Recursive and **polymorphic** types are interpreted via guarded recursion and second-order quantification, respectively; we pun type variables α, β as predicate variables, which are implicitly assumed to be of sort $\mathbb{P}(\mathrm{Val} \times \mathrm{Val})$, and for quantification explicitly required to satisfy $\mathrm{Type}(\alpha)$. These two constraints guarantee that the relation $[\![\tau]\!]$ is a resource-insensitive, binary predicate on values. The resource-insensitivity reflects the fact that refinement between values must hold in an arbitrary context of observation/usage—including contexts that freely copy the values in question—which means that we can assume nothing about privately-owned resources.

The context can interact with values of **reference type** by either reading from or writing to them at any time. Thus, references assert the existence of a simple *invariant* protocol with a single state, whose interpretation says that related locations must continuously point to related values. For **thread-local references**, the statement is qualified by indexing the storage table by related thread IDs.

The uses of $\triangleright$ throughout are necessary to ensure that recursive predicate variables α only occur in guarded positions. Note that island assertions are implicitly guarded, as are uses of $(e_I, e_S) \downarrow \varphi$ where e_I is not a value, because any α mentioned therein will not be interpreted until after a step of computation.

Finally, e_I is *logically related* to e_S at some context Ω and type τ if $\Omega \vdash e_I \preceq e_S : \tau$, which is shorthand for a use of the $\mathcal{C} \vdash P$ judgment of CaReSL. The proof context closes all type variables α with arbitrary type interpretations and all term variables x with *pairs* of term variables related at the appropriate types. Thus, a term with a free variable of type **ref** τ will gain access in its proof context to a shared invariant protocol governing the corresponding location. The protocol in turn forces updates to the reference in the implementation to occur in lock-step with those in the spec. *Private* references, *i.e.,* those allocated *within* the implementation or spec, face no such requirement—unless or until they are exposed.

Treiber's stack, with an iterator We now sketch a simple refinement proof. Figure 9 gives two stack implementations. The first, stack_S, is a coarse-grained reference implementation (which we think of as a specification). Its representation includes a mutable reference hd_S to an immutable list, as well as a synchronization wrapper sync provided by mkSync. The exported functions to push and pop from the stack simply wrap the corresponding updates to hd_S with synchronization. But the iterator is *not* fully synchronized: it takes an atomic snapshot of the stack, then calls its argument f on each element of the snapshot without holding any locks. So by the time f is called, the contents of the stack may have changed.

The second implementation, stack_I, is Treiber's stack supplemented with an iterator. Treiber's stack [28] is one of the simplest lock-free data structures—a kind of "Hello World" for concurrent program logics. Like stack_S, Treiber's stack maintains a reference hd_I to a list that it uses to represent the stack. Instead of using a lock, however, it updates hd_I directly (and atomically) using **CAS**, which requires the contents of hd_I to have a *comparable* type, *i.e.,* to be testable for pointer equality. Thus hd_I is of type **ref** $\mathsf{clist}(\alpha)$, where $\mathsf{clist}(\alpha) \triangleq \mu\beta.\ \mathbf{ref}\ (\mathbf{1} + (\alpha \times \beta))$. The push and pop functions employ *optimistic concurrency*: instead of acquiring a lock to inform other threads that they are going to make a change, they just take a snapshot of the current hd_I, compute a new value, and use **CAS** to install the new value if the identity of the hd_I has not changed. Intuitively, this strategy works because if the hd_I's identity has not changed, then *nothing* about the stack has changed: *all* mutation is performed by identity-changing updates to hd_I. The iterator simply walks over the stack's contents starting from a (possibly stale!) snapshot of hd_I.

To prove that stack_I refines stack_S, we begin as follows:

$$\mathsf{stack}_I \triangleq \Lambda.\ \boxed{\textbf{Prop context: } \mathrm{Type}(\alpha),\ i \bowtie j \qquad \textbf{Variables: } \alpha, i, j, \kappa}$$

$$\{j \Mapsto_S \kappa[\mathsf{stack}_S\ \_]\}\ \ \mathbf{let}\ \mathsf{hd}_I = \mathbf{new}\ \mathsf{nil}$$
$$\{j \Mapsto_S \kappa[\mathsf{stack}_S\ \_] * \exists x.\ \mathsf{hd}_I \hookrightarrow_I x * x \hookrightarrow_I \mathsf{none}\} \qquad (\textsc{ExecSpec})$$
$$\left\{\begin{array}{l}\exists\ \mathsf{hd}_S, \mathsf{lock}.\ \mathsf{hd}_S \hookrightarrow_S \mathsf{none} * \mathsf{lock} \hookrightarrow_S \mathbf{false} * j \Mapsto_S \kappa[\mathsf{stack}'_S] * \\ \quad \exists x.\ \mathsf{hd}_I \hookrightarrow_I x * x \hookrightarrow_I \mathsf{none}\end{array}\right\}$$

We have executed the "preambles" of both the implementation and spec—the code that allocates their respective hidden state. (Let stack'_S denote the rest of the spec code after the **let**-binding of sync and hd_S.) At this point, we will use NEWISL to move *all* of this state into a hidden island, with a protocol that we can use when proving refinement for each exported function.

The states s of the protocol are maps $\mathrm{Loc} \overset{\text{fin}}{\rightharpoonup} \mathrm{Val}$ which simply record the identity and contents of every node added to stack_I. The transition relation $s \rightsquigarrow s' \triangleq s \subseteq s'$ captures the idea that, once a node has appeared in the stack, its contents never change. There are no tokens. We interpret a state s as follows:

$$\begin{array}{l}\varphi \triangleq (s).\ \mathsf{lock} \hookrightarrow_S \mathbf{false} * \mathop{\Huge *}_{\ell \in \mathrm{dom}(s)} \ell \hookrightarrow_I s(\ell) \\ \quad * \ \exists x_I, x_S.\ \mathsf{hd}_I \hookrightarrow_I x_I * \mathsf{hd}_S \hookrightarrow_S x_S * \mathrm{Link}(s, x_I, x_S)\end{array}$$

Because the lock is protected by the protocol, we can only execute the exported spec functions with AEXECSPEC, as part of an application of UPDISL; by stipulating $\mathsf{lock} \hookrightarrow_S \mathbf{false}$, the interpretation furthermore requires that we run the spec in "big steps", starting and ending in unlocked states. It also says that the implementation and spec stack contents are *linked*, in the following sense:

$$\begin{array}{l}\mathrm{Link} \triangleq \mu p.\ (s, x_I, x_S).\ \exists x'_I, s'.\ s = [x_I \mapsto x'_I] \uplus s' * \\ \quad (x'_I = \mathsf{none} * x_S = \mathsf{none})\ \vee \left(\begin{array}{l}\exists y_I, z_I.\ x'_I = \mathsf{some}(y_I, z_I) * \\ \exists y_S, z_S.\ x_S = \mathsf{some}(y_S, z_S) * \\ (y_I, y_S) \in \alpha * \triangleright p(s', z_I, z_S)\end{array}\right)\end{array}$$

Linking requires that the stack state s contain an entire linked list starting from hd_I, and that each element of the list be related (at type α) to the corresponding element of the list in hd_S. *But it does so without mentioning the heap at all!* This is the key: it means that $\mathrm{Link}(s, x_I, x_S) \Rightarrow \Box\mathrm{Link}(s, x_I, x_S)$, so the Link assertion can be freely copied in the proof for iter when the traversal begins. Since the abstract state s of the stack can only grow, all of the nodes mentioned in this "snapshot" of Link are guaranteed to still be available in the region of the heap governed by the protocol, with their original values, as traversal proceeds—even if, in the meantime, the nodes are no longer reachable from hd_I.

The full proof outlines are available in the appendix [30].

Combining refinement and Hoare-style reasoning Suppose a client of Treiber's stack wishes to use it only in a weak way, via a "per-item" spec similar to the one suggested in the introduction:

$$\begin{aligned}
&\mathrm{PerItem} \triangleq (e \in \mathrm{Expr}).\ \forall p, q \in \mathbb{P}(\mathrm{Val}).\\
&\quad (\forall x.\ \mathrm{TokPure}(p(x)) \ \wedge\ (p(x) \Rightarrow \Box q(x))) \Rightarrow\\
&\qquad \{\mathrm{True}\}\ e\ \{\mathrm{obj}.\ \mathsf{obj} = (\mathsf{add}, \mathsf{rem}, \mathsf{iter}) \wedge \Box P\}, \quad \textit{where}\\
&P \triangleq \forall x.\ \{p(x)\}\ \mathsf{add}(x)\ \{\mathrm{ret}.\ \mathsf{ret} = ()\}\\
&\quad \wedge\ \{\mathrm{True}\}\ \mathsf{rem}()\ \{\mathrm{ret}.\ \mathsf{ret} = \mathsf{none} \vee (\exists x.\ \mathsf{ret} = \mathsf{some}(x) * p(\mathsf{ret}))\}\\
&\quad \wedge\ \forall f, r.\ (\forall x.\ \{q(x) * r\}\ f(x)\ \{r\}) \Rightarrow \{r\}\ \mathsf{iter}(f)\ \{r\}
\end{aligned}$$

This "per-item" spec associates a resource p with each element of the stack, which is transferred to and from the data structure when adding or removing elements. If, in addition, each per-item resource entails some permanent knowledge $\Box q$, then that knowledge can be safely assumed by a function concurrently iterating over the stack, even if the resources originally supporting it have been consumed. As we will see in the case study (§4), this spec for iteration is useful for associating permanent, token-free knowledge about some other protocol with each item that appears in the stack.

While the per-item spec could in principle be applied directly to Treiber's stack (*i.e.,* we could prove PerItem($\mathsf{stack}_\mathrm{I}$ _)), doing so would repeat much of the verification we just performed. On the other hand, proving PerItem($\mathsf{stack}_\mathrm{S}$ _) is nearly trivial: we just instantiate the spinlock spec given in §3.2 with the representation invariant $\mathrm{Rep}_p \triangleq \exists l.\ \mathsf{hd}_\mathrm{S} \hookrightarrow_\mathrm{I} l * \mathrm{Map}_p(l)$. To verify iter, we need only observe that $\mathrm{Map}_p(l) \Rightarrow \Box\mathrm{Map}_q(l)$, *i.e.,* the snapshot of the stack provides a list of necessarily true associated facts. (The details are in the appendix [30].) By mixing refinement and Hoare logic, we can thus significantly modularize our verification effort.

3.4 Soundness

The model theory of CaReSL is based directly on the semantic model of Turon *et al.* [31], with minor adjustments to accommodate the assertions $\mathit{Tid}(i)$ and $i \bowtie j$ necessary for reasoning about thread-local storage. (Since this paper is focused on the complementary aim of giving a syntactic proof theory, we do not delve into the model here; it can be found in full in the appendix [30]).

Soundness for CaReSL encompasses two theorems. **First**, that $\mathcal{C} \vdash P$ implies $\mathcal{C} \models P$, where the latter is the semantic entailment relation of the model. Proving this theorem requires validating, in particular, the key Hoare logic rules, which we do in the appendix; these proofs resemble the proofs of key lemmas supporting Turon *et al.*'s model. **Second**, that $\cdot \vdash (\Omega \vdash e_\mathrm{I} \preceq e_\mathrm{S} : \tau)$ implies $\Omega \models e_\mathrm{I} \preceq e_\mathrm{S} : \tau$, *i.e.,* that the logical relation is sound for contextual refinement. Again, this proof follows the soundness proof of Turon *et al.*, except that we can carry it out at a much higher level of abstraction using CaReSL's proof rules.

4. Case study: Flat combining

Using CaReSL's combination of Hoare-style and refinement reasoning, we can verify higher-order concurrent algorithms in layers of abstraction—and this section shows how to do it.

We study the recent *flat combining* construction of Hendler *et al.* [11], which takes an arbitrary sequential data structure and transforms it into a concurrent one in which all operations appear to take effect atomically. One can do so by merely wrapping each operation with synchronization on a global lock—indeed, this is

```
flat_S ≜ mkSync()  :  [∀α. ∀β. (α → β) → (α → β)]

flat_I ≜ Λ[α]. Λ[β]. λf : [α → β].        (Annotated with types for clarity)
  let lock : [ref B] = new (false)
  let lclOps : [refLcl op] = newLcl                    (where op ≜ ref (α + β))
  let (add, _, iter) = stack_I [op]
  let install : [α → op] = λ req. case getLcl(lclOps)
    of some(o) ⇒ o := inj_1 req; o
     | none ⇒ let o  = new (inj_1 req) in setLcl(lclOps, o); add(o); o
  let doOp : [op → 1] = λo. case get(o)
    of inj_1 req ⇒ o := inj_2 f(req)
     |  inj_2 res ⇒ ()
  in λx : [α]. let o = install(x)
               let rec loop() = case get(o)
                 of inj_1 _   ⇒ if not(get lock) and tryAcq(lock) then
                                   iter doOp; rel(lock); loop()
                                 else loop()
                 |  inj_2 res ⇒ res
               in loop()
```

Figure 10. Flat combining: Spec and implementation

what our *spec* does—but flat combining takes a cache-friendly approach intended for hard-to-parallelize data structures.

The basic idea is to allow concurrent threads to advertise, through a lock-free side channel, their intent to perform an operation on the data structure. One of the threads then becomes the *combiner*: it locks the data structure and services the requests of all the other threads, one at a time (though new requests may be published concurrently with the combiner's execution). The algorithm exhibits relatively good cache behavior for two reasons: (1) most of the time, operations do not need to execute *any* **CAS**es to complete and (2) the combining thread enjoys good cache locality when executing the operations of the sequential data structure. In practice, flat combining yields remarkably strong performance, even when compared against completely lock-free data structures.

The algorithm and its spec The flat combining algorithm was originally given as informal prose, so our first task is to formalize its implementation and find an appropriate spec, while making its higher-order structure explicit. We do so in Figure 10.

We model an "arbitrary sequential data structure" as a closure f of some type $\alpha \to \beta$, where α represents the input to an operation to be performed, β represents the result, and calling the function performs the operation by imperatively updating some state hidden within the closure. In other words, $\alpha \to \beta$ represents the type of an *object*. The goal of flat combining, then, is to wrap this object with (cache-efficient) synchronization. Its spec, flat_S, simply uses mkSync to provide generic synchronization via global locking.

Our flat combining implementation flat_I uses three data structures to control synchronization.[10] First, it uses a global lock to ensure that only one thread at a time acts as the combiner. Second, it uses a Treiber's stack to enable threads to *enroll* in the data structure. To enroll, a thread inserts an *operation record* of type **ref** $(\alpha + \beta)$, by which it can both advertise requests (of type α) and then await the results (of type β). The combiner performs these requests by iterating over the stack: for each record containing an $\mathbf{inj}_1$ req, it applies f to req, obtaining result res and updating the record to $\mathbf{inj}_2$ res. Finally, the thread-local reference lclOps allows threads to re-use their operation record once they have enrolled. Note that operation records can be added to the stack or reset from β to α at *any* time—even when the combiner holds the lock.

[10] This implementation simplifies Hendler *et al.*'s description in three respects: it uses a stack rather than a queue, operation records are never de-enrolled, and the combiner does not coalesce multiple operations into a single step. The first simplification makes little difference, because ultimately we give a modular proof using our per-item spec, which could be applied to a queue as well. The other two need only minor changes to the protocol.

$$\begin{tikzcd}
\boxed{\text{Init}} \arrow[r, bend left, "{\ell, j, K, v}"] & \boxed{\text{Req}(\ell, j, K, v); \diamond_i} \arrow[r, bend left] & \boxed{\text{Exec}(\ell); \mathit{Lock}} \arrow[d, bend left, "v'"] \\
\boxed{\bot; \bullet_i, \diamond_i} \arrow[u, bend left] & \boxed{\text{Ack}(\ell); \bullet_i, \diamond_i} \arrow[u, bend left, "{j, K, v}"] & \boxed{\text{Resp}(\ell, j, K, v'); \diamond_i} \arrow[l, bend left]
\end{tikzcd}$$

$$\begin{aligned}
[\![\text{Init}]\!]_i^L &\triangleq \mathit{Tid}(i) \\
[\![\text{Req}(\ell, j, \kappa, x_{\text{I}})]\!]_i^L &\triangleq L(i) = \ell * \ell \hookrightarrow_{\text{I}} \mathbf{inj}_1\ x_{\text{I}} * \mathit{Tid}(i) * \exists x_{\text{S}}.\ (x_{\text{I}}, x_{\text{S}}) \in \alpha * j \Mapsto_{\text{S}} \kappa[f'_{\text{S}}\ x_{\text{S}}] \\
[\![\text{Exec}(\ell)]\!]_i^L &\triangleq L(i) = \ell * \ell \hookrightarrow_{\text{I}} \mathbf{inj}_1\ - * \mathit{Tid}(i) \\
[\![\text{Resp}(\ell, j, \kappa, y_{\text{I}})]\!]_i^L &\triangleq L(i) = \ell * \ell \hookrightarrow_{\text{I}} \mathbf{inj}_2\ y_{\text{I}} * \mathit{Tid}(i) * \exists y_{\text{S}}.\ (y_{\text{I}}, y_{\text{S}}) \in \beta * j \Mapsto_{\text{S}} \kappa[y_{\text{S}}] \\
[\![\text{Ack}(\ell)]\!]_i^L &\triangleq L(i) = \ell * \ell \hookrightarrow_{\text{I}} \mathbf{inj}_2\ -
\end{aligned}$$

where $f'_{\text{S}} \triangleq \lambda x.\ \mathsf{acq}(\mathsf{lock}_{\text{S}}); \mathbf{let}\ r = f_{\text{S}}(x)\ \mathbf{in}\ \mathsf{rel}(\mathsf{lock}_{\text{S}});\ r$

Figure 11. The protocol for the operation record of thread i

Unfortunately, if we set out to prove the refinement

$$\cdot \vdash \mathsf{flat}_{\text{I}} \preceq \mathsf{flat}_{\text{S}} : \forall\alpha.\ \forall\beta.\ (\alpha \to \beta) \to (\alpha \to \beta)$$

we will run headlong into a problem: it does not hold! To wit: let

$$\begin{aligned}
C \ \triangleq\ & \mathbf{let}\ r\ =\ \mathbf{newLcl},\ f\ =\ [\ ]\ \_\ \_\ (\lambda().\ \mathbf{getLcl}(r)) \\
& \mathbf{in\ fork}\ (\mathbf{setLcl}(r, \mathbf{true});\ f());\quad \mathbf{setLcl}(r, \mathbf{false});\ f()
\end{aligned}$$

When this context is applied to flat_{S}, it always returns $\mathsf{some}(\mathbf{false})$, because the final $f()$ is always executed on the thread whose local r is $\mathbf{false}$. But when applied to flat_{I}, it can also return $\mathsf{some}(\mathbf{true})$: the forked thread might act as the combiner, performing *both* applications of f and thus using its own thread-local value for r.

The crux of the problem is that thread-local storage allows functions to observe the identity of the thread executing them. We need to rule out this kind of side effect. We do so by unrolling the definition of refinement for functions, and simply removing access to ID-related knowledge, leading to a *qualified refinement*:

$$\begin{aligned}
&\forall\alpha, \beta, f_{\text{I}}, f_{\text{S}}. \\
&\quad \text{Type}(\alpha) \wedge \text{Type}(\beta) \wedge \Box(\forall(x_{\text{I}}, x_{\text{S}}) \in \alpha.\ (f_{\text{I}}\ x_{\text{I}}, f_{\text{S}}\ x_{\text{S}}) \downarrow^{\text{pure}} \beta) \\
&\qquad \Rightarrow\ \cdot \vdash \mathsf{flat}_{\text{I}}\ \_\ \_\ f_{\text{I}} \preceq \mathsf{flat}_{\text{S}}\ \_\ \_\ f_{\text{S}} : \alpha \to \beta \\
&\quad \text{where } (e_{\text{I}}, e_{\text{S}}) \downarrow^{\text{pure}} \varphi \ \triangleq\ \forall i, j, \kappa. \\
&\qquad\qquad \{j \Mapsto_{\text{S}} \kappa[e_{\text{S}}]\}\ i \Mapsto e_{\text{I}}\ \{x_{\text{I}}.\ \exists x_{\text{S}}.\ \varphi(x_{\text{I}}, x_{\text{S}}) * j \Mapsto_{\text{S}} \kappa[x_{\text{S}}]\}
\end{aligned}$$

This "pure" notion of related expressions embodies a semantic version of purity in a type-and-effect system where the only effect is "uses thread-local storage". It coincides *exactly* with the expression relation in Turon *et al.* [31], where thread-local storage was not present. It is easy to prove, using CaReSL, that for any well-typed $f : \tau \to \tau'$ that does not use thread-local storage (*i.e.,* an "effect-free" f) we have $\Box\forall(x_{\text{I}}, x_{\text{S}}) \in [\![\tau]\!].\ (f\ x_{\text{I}}, f\ x_{\text{S}}) \downarrow^{\text{pure}} [\![\tau']\!]$.

The protocol To prove the qualified refinement, we need to formulate a protocol characterizing the algorithm. We do so by giving a local protocol governing the operation record of each thread, and then tying these local protocols together into a single global one. Figure 11 gives the local protocol for a thread with ID i, where the states have the following informal meanings:

$\bot$	thread i has not yet created its operation record
Init	thread i is creating its operation record
Req	thread i has requested an operation be performed
Exec	the combiner is executing thread i's operation
Resp	thread i's operation has been completed
Ack	thread i has acknowledged the completion

The cycle in the protocol reflects that, once thread i has enrolled an operation record, it can reuse that record indefinitely. The labels on transitions signify *branching* on the values of the listed variables. Note that, while ℓ (the location of the operation record) is chosen arbitrarily during initialization, it must remain fixed thereafter.

Movement through the local protocol encompasses two roles: that of thread i, and that of the combiner. The protocol guarantees: (1) when thread i begins execution of the flat combining algorithm, the current state is either $\bot$ or Ack, meaning that either there is no operation record associated with the thread, or that the associated record is ready to be re-used; (2) only thread i can move to or from Init and Ack; and (3) only the combiner can move to or from Exec.

To fully understand how these constraints are enforced, we must take into account both the tokens and the state interpretations of the protocol. The tokens include $\bullet_i$ and $\diamond_i$ (one for each thread i, used only in i's local protocol) and a single global token *Lock* representing ownership of the combiner's lock.

The interpretations $[\![-]\!]_i^L$ of the non-$\bot$ states are shown in the figure; the parameter L represents the contents of lclOps. With the exception of $\bot$ and Ack, all states assert ownership of $\mathit{Tid}(i)$. On the other hand, when thread i begins executing the algorithm, *it* will have ownership of $\mathit{Tid}(i)$ (recall the definition of refinement, §3.3). Thus, thread i can assume that the protocol is in state $\bot$ or Ack—and only thread i can take a step away from these states. For thread i to take such a step, it must "prove" that it *is* thread i by giving up $\mathit{Tid}(i)$, but in return it gains the token $\bullet_i$ as a "receipt" that can later be traded back for $\mathit{Tid}(i)$ by moving (back) to Ack. All told, the ownership of $\mathit{Tid}(i)$ and the corresponding token $\bullet_i$ account for the first two of the guarantees listed above.

The third guarantee is achieved using a similar strategy: to move to the Exec state, the combiner must "prove" that it holds the lock by temporarily giving up the (global) *Lock* token, but it receives the (local) token $\diamond_i$ as a receipt. Subsequently, only the combiner can move to Resp, making the opposite trade.

After initialization, each state also asserts that the location ℓ of the operation record for thread i both corresponds to $L(i)$ and points to an appropriate sum injection (depending on the state).

A final aspect of the local protocol is capturing the *cooperation* inherent in flat combining: the combiner executes code on behalf of other threads. Cooperation is difficult for compositional refinement techniques to handle, because such techniques usually require proving that for *each piece* of implementation code the corresponding piece of spec code behaves the same way—and thus they typically provide no way to account for the *mismatch* between implementation and spec that cooperation entails. While our definition of refinement (§3.3) likewise imposes a one-to-one relationship between implementation and spec code in its pre- and post-conditions, our treatment of spec code as a resource allows ownership to be transferred to other threads *during* the execution of the implementation.

Thus, the Req and Resp states of our protocol assert *shared* ownership of i's spec code (running in spec thread j). In moving from Req to Exec the combiner thread must take ownership of the spec code $\kappa[f'_{\text{S}}\ x_{\text{S}}]$; and to subsequently move to Resp, the combiner must actually evaluate this code on thread i's behalf until $f'_{\text{S}}\ x_{\text{S}}$ reaches a value y_{S}. Subsequently, when moving to the Ack state, thread i regains ownership of both $\mathit{Tid}(i)$ *and* its spec code, by giving up its "receipt" token, $\bullet_i$.

The global transition system is then a *product construction*:

$$\begin{array}{ll}
\text{State space} & \left\{ (S, s) \;\middle|\; \begin{array}{l} S \in \mathbb{N} \overset{\text{fin}}{\rightharpoonup} \text{OpState}, \quad s \in \text{LockState}, \\ \text{at most one } S(i) \text{ or } s \text{ owns } \mathit{Lock} \end{array} \right\} \\
\text{Transitions} & (S, s) \rightsquigarrow (S', s') \ \triangleq\ s \rightsquigarrow^{?}_{\mathsf{lock}} s' \ \wedge\ \forall i.\ S(i) \rightsquigarrow^{?}_{i} S'(i) \\
\text{Owned tokens} & \mathcal{T}(S, s) \ \triangleq\ \mathcal{T}_{\mathsf{lock}}(s) \ \cup\ \bigcup_i \mathcal{T}_i(S(i))
\end{array}$$

A global state includes a collection S of local states from the operation record protocol (Figure 11), one for each thread ID. In giving the state space, we pun the $\bot$ state with an "undefined" input to a partial function—and thus, we require that only a finite number of threads i have a non-$\bot$ state in S. Global states also have a single state s drawn from the standard locking protocol (§3.2), reflecting the state of the global lock. The global *Lock* token is *shared* amongst all of the combined protocols: in a valid global state, at most one of the local states $S(i)$ or s may claim the *Lock* token. A transition in the global protocol allows each local protocol to make at most one transition. Finally, the tokens owned in a global state are just the union of all the tokens claimed by the local states.

Recall that in the refinement proof for Treiber's stack (§3.3), the spec lock is treated continuously as a protocol-owned resource with value **false**, capturing the atomicity of the implementation code: if an implementation step coincides with *any* spec steps, it must coincide with an entire critical section's worth, going from unlocked to unlocked spec states in big steps. The combiner implementation, by contrast, is executing a function f_I that is not necessarily atomic; all we know is that f_I refines f_S. We must therefore allow the spec lock to be held for multiple implementation steps, which we do by interpreting lock states as follows:

$$\begin{array}{rcl} [\![\mathsf{Unlocked}]\!] & \triangleq & \mathsf{lock} \hookrightarrow_\mathrm{I} \mathbf{false} * \mathsf{lock}_\mathrm{S} \hookrightarrow_\mathrm{S} \mathbf{false} \\ [\![\mathsf{Locked}]\!] & \triangleq & \mathsf{lock} \hookrightarrow_\mathrm{I} \mathbf{true} \end{array}$$

In short, the combiner gains private ownership of both the *Lock* token and the spec lock itself (*i.e.,* the internal lock used by mkSync; see f'_S in Figure 11) when it acquires the implementation lock.

Finally, we lift these local interpretations to interpret global states via the following predicate:

$$(S, s).\ [\![s]\!] * \exists L.\ \mathsf{lclOps} \hookrightarrow_\mathrm{I} L * \mathop{\Large\ast}_{i \in \mathrm{dom}(S)} [\![S(i)]\!]^L_i$$

The proof We close with a high-level view of the proof itself; details, as usual, are in the appendix. Unrolling the statement of qualified refinement, we need to prove

$$\{\mathit{Tid}(i) * j \mapsto\!\!\!\!\Rightarrow_\mathrm{S} \kappa[\mathsf{flat}_\mathrm{S}\ \_\ \_\ f_\mathrm{S}]\}\ \ i \mapsto\!\!\!\!\Rightarrow \mathsf{flat}_\mathrm{I}\ \_\ \_\ f_\mathrm{I}$$
$$\{x_\mathrm{I}.\ \exists x_\mathrm{S}.\ (x_\mathrm{I}, x_\mathrm{S}) \in [\![\alpha \to \beta]\!] * \mathit{Tid}(i) * j \mapsto\!\!\!\!\Rightarrow_\mathrm{S} \kappa[x_\mathrm{S}]\}$$

under the assumptions

$$i \bowtie j,\ \ \mathrm{Type}(\alpha),\ \ \mathrm{Type}(\beta),\ \ \Box(\forall (x_\mathrm{I}, x_\mathrm{S}) \in \alpha.(f_\mathrm{I}\ x_\mathrm{I}, f_\mathrm{S}\ x_\mathrm{S}) \downarrow^{\mathrm{pure}} \beta)$$

The proof begins in much the same way as the refinement proof for Treiber's stack (§3.3): we execute the **let**-bound expressions that allocate the hidden state in both the implementation and spec, *i.e.,*

$$\mathsf{lock} \hookrightarrow_\mathrm{I} \mathbf{false} * \mathsf{lclOps} \hookrightarrow_\mathrm{I} \emptyset * \exists\ \mathsf{lock}_\mathrm{S}.\ j \mapsto\!\!\!\!\Rightarrow_\mathrm{S} \kappa[f'_\mathrm{S}] * \mathsf{lock}_\mathrm{S} \hookrightarrow_\mathrm{S} \mathbf{false}$$

where lock_S is the lock allocated by mkSync and f'_S is as in Figure 11. These resources are precisely what we need to apply NewIsl for our global protocol, moving them from private to shared ownership. Letting π be the global protocol defined above, we can claim $\Box\ \boxed{(\emptyset, \mathsf{Unlocked}); \emptyset}^{n}_{\pi}$, *i.e.,* permanent knowledge that the global protocol exists. We must then, in the context of this island and our previous assumptions, show that the closure returned by flat_I refines the one returned by flat_S, *i.e.,* f'_S, at type $\alpha \to \beta$.

We first verify a version $\mathsf{flat}'_\mathrm{I}$ of the flat combining algorithm that is identical to the one in Figure 10, except that it uses the coarse-grained stack, allowing us to use the per-item spec of §3.3. We instantiate the per-item spec using the same predicate Op for per-item resources and iteration knowledge (p and q, respectively, in the per-item spec): $\mathrm{Op} \triangleq (\ell).\ \exists k.\ \boxed{[k \mapsto \mathrm{Req}(\ell, -, -, -)]}^{n}_{\pi}$.[11] This is a *local* assertion about the global protocol, in that the states of threads other than k can be in any rely-future state of $\bot$, *i.e.,* any state whatsoever. The Op predicate just claims that location ℓ is an initialized operation record for *some* thread. By the per-item spec, all locations inserted into the stack must satisfy this property—and since we have $\mathrm{Op}(\ell) \Rightarrow \Box\mathrm{Op}(\ell)$, the property can be assumed even when iterating over stale elements of the stack. Operation records are created via install, whose specification consumes $\mathit{Tid}(i)$ and associated spec resources in exchange for the "receipt" $\bullet_i$:

$$\left\{\mathit{Tid}(i) * j \mapsto\!\!\!\!\Rightarrow_\mathrm{S} \kappa[f'_\mathrm{S}\ x_\mathrm{S}]\right\} i \mapsto\!\!\!\!\Rightarrow \mathsf{install}\ x \left\{o.\ \boxed{[i \mapsto \mathrm{Req}(o, j, \kappa, x)]; \bullet_i}^{n}_{\pi}\right\}$$

[11] We leave off the locking state when it is Unlocked.

for any $(x, x_\mathrm{S}) \in \alpha$ (*i.e.,* for any related arguments). The combiner actually performs operations via doOp,

$$\{\mathrm{Op}(o) * P\}\ \mathsf{doOp}\ o\ \{P\} \quad \text{where} \quad P \triangleq \boxed{\emptyset; \mathit{Lock}}^{n}_{\pi} * \mathsf{lock}_\mathrm{S} \hookrightarrow_\mathrm{S} \mathbf{false}$$

which assumes that the given location o is a valid operation record, and that the invoking thread owns the *Lock* token as well as the spec lock. The shape of the spec for doOp exactly matches that required for iteration in the per-item spec (§3.3), with P serving as the loop invariant, thus allowing us to deduce $\{P\}$ iter doOp $\{P\}$. These auxiliary specs make it straightforward to prove refinement for the closures returned by $\mathsf{flat}'_\mathrm{I}$ (with coarse-grained stack) and flat_S.

Finally, suppose we plug the combiner into a client context C that provides a suitable argument f. The above proof (together with soundness, §3.4) allows us to deduce $\models C[\mathsf{flat}'_\mathrm{I}] \preceq C[\mathsf{flat}_\mathrm{S}] : \tau$. Likewise, refinement for Treiber's stack allows us to deduce $\models C[\mathsf{flat}_\mathrm{I}] \preceq C[\mathsf{flat}'_\mathrm{I}] : \tau$. Since refinement is *transitive*, we can compose these together to conclude $\models C[\mathsf{flat}_\mathrm{I}] \preceq C[\mathsf{flat}_\mathrm{S}] : \tau$.[12]

5. Related work

In many respects, CaReSL builds directly on the semantic foundation that we and colleagues laid in our prior work [31]. There, we developed a relational Kripke model of a language very similar to the one considered here, and showed how granularity abstraction for several sophisticated (but structurally simple) fine-grained data structures could be established by direct appeal to the model. The present work is a natural continuation of that work. First, we provide a *logic* that greatly simplifies reasoning compared to working with the model; such a proof theory is essential for scaling the verification method to large examples. Second, we use our logic not just to prove granularity abstraction results in isolation (as we did before), but also to facilitate the modular verification of a higher-order, structurally complex program. Along the way, we also extend our prior model to handle thread-local storage.

The logic itself draws inspiration from LADR [5], which in turn provided a proof theory for reasoning about a *sequential* relational Kripke model (ADR [1]). Aside from incorporating concurrency, CaReSL offers a deeper unification of refinement and Hoare logic by treating refinement as a mode of use of Hoare logic.

Higher-order functions and concurrency While there has been stunning progress in logics for concurrency over the last decade, very few of these logics meaningfully support *higher-order* programming, and among those that do, none supports reasoning about fine-grained synchronization or granularity abstraction.

The logic that comes closest is *higher-order concurrent abstract predicates* (HOCAP), first proposed for reasoning about first-order code [4] (the "higher-order" refers to the logic of predicates) and very recently applied to higher-order code as well [27]. HOCAP, like its predecessor CAP [3], accounts for concurrency by way of "shared region" assertions that constrain the possible updates to a shared resource. Our island assertions resemble shared region assertions—indeed, we have adopted notation suggesting as much—but work at a higher level of abstraction (*i.e.,* protocol states), separating *knowledge* bounding the state of the protocol (treated as a copyable assertion) from the *rights* to change the state (treated as a linear resource: tokens); see [31] for a more detailed comparison. Because CAP lacks granularity abstraction, it is difficult to give a single "principal" specification for a data structure. Instead, one gives specialized specs (like the per-item spec for stacks) reflecting particular usages envisioned for a client—which means new client scenarios necessitate new proofs. HOCAP attempts to

[12] We are appealing to *semantic* refinement here to take advantage of transitivity. This reasoning can be internalized in CaReSL by adding a proposition for semantic refinement and axioms for refinement soundness and transitivity, but there is little to be gained from doing so.

address this shortcoming by explicitly quantifying over the way a client uses a data structure, but (1) this introduces the potential for a problematic circularity, which clients must explicitly prove does not arise, and (2) it is not clear how to scale the approach to handle *cooperation* between threads (as in the flat combiner).

Other concurrency logics that support higher-order functions—such as Hobor *et al.*'s extension of concurrent separation logic [14], or the very recent *Subjective Concurrent Separation Logic* [17]—support only reasoning about simple lock-based synchronization, and do not enable the kinds of refinement proofs we have presented.

Granularity abstraction Herlihy and Wing's seminal notion of *linearizability* [13] has long been held as the gold standard of correctness for concurrent data structures, but as Filipović *et al.* argue [8], what clients of these data structures *really* want is a contextual refinement property. Filipović *et al.* go on to show that, under certain (strong) assumptions about a programming language, linearizability implies contextual refinement for that language. More recently, Gotsman and Yang generalized both linearizability and Filipović *et al.*'s result (the so-called *abstraction theorem*) to include potential ownership transfer of memory between concurrent data structures and their clients in a first-order setting [9]. While in principle proofs of linearizability can be composed with these results to support granularity abstraction, no existing logic has provided support for doing so. We found it simpler to work with refinement directly (in particular, to *encode* it directly into a Hoare logic), rather than reasoning about linearizability. CaReSL enables clients to layer ownership-transferring protocols *on top* of a coarse-grained reference implementation, *after* applying granularity abstraction, as we showed with the per-item spec (§3.3).

The only Hoare logic we are aware of that can (formally) prove refinement results is Liang and Feng's new logic [18] (extending LRG [7]), which is inspired by the use of ghost state in Vafeiadis's thesis [32]. The logic is powerful enough to deal with a range of sophisticated, fine-grained concurrent algorithms, but it is limited to *first-order* code. In addition, the specifications used in refinement are not reference implementations, but are instead essentially atomic Hoare triples. While such specifications are appealingly abstract, they present two limitations: (1) they do not model the more general notion of granularity abstraction (supporting only *atomicity* abstraction, as we explained in footnote 1) and (2) they do not support the kind of transitive composition of proofs that we used in our case study. As a result, it is unclear how to use the logic to build modular proofs layering Hoare logic and refinement.

A radically different approach to atomicity abstraction is Lipton's method of *reduction* [19], which is based on showing that an atomic step *commutes* with all concurrent activity, and can therefore be coalesced into a larger atomic block with *e.g.,* code that is sequenced after it. Elmas *et al.* developed a logic for proving linearizability via a combination of reduction and abstraction [6], which in some ways resembles our interleaved application of refinement and Hoare-style reasoning, but with a rather different way of *proving* refinement. It is limited, however, to first-order code and atomicity refinement, and like HOCAP it is not powerful enough to handle fine-grained algorithms that employ cooperation. Moreover, it does not allow linearizability proofs to be composed transitively.

Acknowledgments We would like to thank David Swasey for his careful reading of both the paper and its appendix. This work was partially funded by the EC FET project ADVENT.

References

[1] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, Aug. 1996.

[3] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[4] M. Dodds, S. Jagannathan, and M. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, 2011.

[5] D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. In *POPL*, 2010.

[6] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, 2010.

[7] X. Feng. Local rely-guarantee reasoning. In *POPL*, 2009.

[8] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411, 2010.

[9] A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.

[10] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, 2005.

[11] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[12] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[13] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[14] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[15] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS*, 5(4):596–619, 1983.

[16] D. Lea. The java.util.concurrent ConcurrentHashMap.

[17] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.

[18] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.

[19] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[20] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[21] M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.

[22] A. M. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS*, 1998.

[23] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *TLCA*, 1993.

[24] F. Pottier. Hiding local state in direct style: a higher-order anti-frame rule. In *LICS*, 2008.

[25] J. H. Reppy. *Higher-order concurrency*. PhD thesis, Cornell University, 1992.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[27] K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.

[28] R. Treiber. Systems programming: coping with parallelism. Technical report, Almaden Research Center, 1986.

[29] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *JFP*, 8(1):23–60, Jan. 1998.

[30] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency: Appendix. http://www.mpi-sws.org/~turon/caresl/appendix.pdf.

[31] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.

[32] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[33] H. Yang. Relational separation logic. *TCS*, 375(1-3):308–334, 2007.

Brief Announcement: Concurrency-Aware Linearizability

Nir Hemed
Tel Aviv University
nirh@mail.cs.tau.ac.il

Noam Rinetzky
Tel Aviv University
maon@cs.tau.ac.il

ABSTRACT

Linearizabilty allows to describe the behaviour of concurrent objects using sequential specifications. Unfortunately, as we show in this paper, sequential specifications cannot be used for concurrent objects whose *observable behaviour* in the presence of concurrent operations *should* be different than their behaviour in the sequential setting. As a result, such *concurrency-aware objects* do not have formal specifications, which, in turn, precludes formal verification.

In this paper we present *Concurrency Aware Linearizability* (CAL), a new correctness condition which allows to formally specify the behaviour of a certain class of concurrency-aware objects. Technically, CAL is formalized as a strict extension of linearizability, where *concurrency-aware specifications* are used instead of sequential ones. We believe that CAL can be used as a basis for modular formal verification techniques for concurrency-aware objects.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Linearizability; sequential specification; concurrent specification

1. INTRODUCTION

Linearizability [4] is a property of the externally-observable behaviour of concurrent objects [4]. Intuitively, a concurrent object is linearizable if in every execution each operation seems to take effect instantaneously between its invocation and response, and the resulting sequence of (seemingly instantaneous) operations respects a given sequential specification. Unfortunately, as we show below, for some concurrent objects it is *impossible* to provide a sequential specification: their behaviour in the presence of concurrent (overlapping) operations is, and should be, *observably different* from their behaviour in the sequential setting. For these objects, which we refer to as *Concurrency-Aware Concurrent Objects (CA-objects)*, the traditional notion of linearizability is simply not expressive enough to allow for describing all desired behaviours without introducing undesired ones. As a result, CA-objects are not given a formal specification. The lack of formal specifications is problematic as it precludes formal proofs.

Concurrency-Aware Linearizabilty (CAL) is a correctness condition which addresses the aforementioned problem. CAL enables programmers to provide natural and intuitive specifications for an important class of CA-objects. Technically, CAL is an extension of linearizabilty where *Concurrency-Aware specifications* are used to describe concurrency-dependent behaviours. Sequential specifications are a special case of concurrency-aware specifications in which concurrent behaviours can be explained by sequential ones.

Running Example. `Exchanger` objects (as found, e.g., in `java.util.concurrent.Exchanger`) serve as a synchronization point at which threads can pair up and *atomically swap* elements. `Exchangers` are useful in applications such as genetic algorithms and pipeline designs, and are embedded in practice in thread-pool implementations as well as other higher-level data structures [5, 6].

Figure 1(`E`) shows a simplified version of the wait-free exchanger of [5] in which retires are omitted. Intuitively, a client thread uses the `Exchanger` by invoking the `exchange()` method with a value that it offers to swap (in our case a positive `int`). `exchange()` attempts to find a partner thread and, if successful, instantaneously exchanges the offered value with the one offered by the partner. If a partner thread is not found, `exchange()` returns `-1`, indicating that the operation has failed. More technically, the exchange is performed by using `Offer` objects, consisting of the `data` offered for exchange and a `hole` pointer. A successful swap occurs when the `hole` pointer in the `Offer` of one thread points to the `Offer` of another thread. This can be achieved in two ways: A thread that finds that the value of `g` is null can set it to its `Offer` (line 10) and wait for a partner thread to match with (`sleep` in line 11). Upon awakening, it checks whether it was paired with another thread by executing a CAS on its own `hole` (line 12). If the CAS succeeds, then a match did not occur, and setting the `hole` pointer to point to the `fail` sentinel signals that the thread is no longer interested in the exchange. If the CAS fails then some other thread has already matched the `Offer` and the exchange can complete successfully. If `g` is not null, then the thread attempts to update the `hole` field of the `Offer` pointed to by `g` from its initial null value to its own `Offer` (line 16). An additional CAS (line 17) sets `g` back to null. By doing so, it helps to remove already-matched offers from the global pointer; hence, the CAS in line 17 is unconditional.

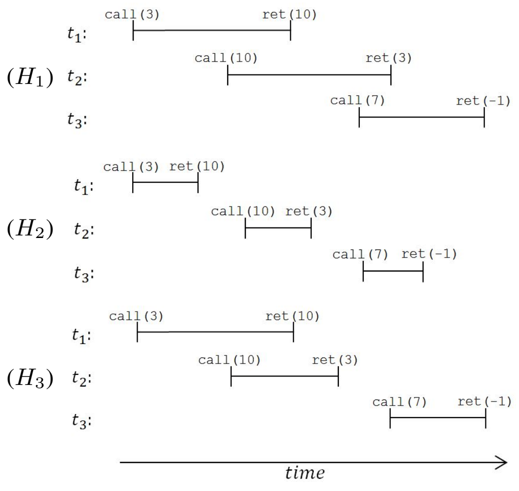
`Exchanger` objects do not have a formal specification. This is not surprising; describing the concurrent behaviour that requires that `exchange()` succeeds only if *two* threads invoke the method simultaneously is, as we show below, impossible using the form of sequential specifications suggested in [4]. As a result, correctness proofs of concurrent objects that utilize `Exchanger`-like ob-

PODC'14, July 15–18, 2014, Paris, France.
ACM 978-1-4503-2944-6/14/07.
http://dx.doi.org/10.1145/2611462.2611513.

(E)

```
class Offer {
  int data;
  Offer hole;
  Offer(int d){data = d; hole = null;}
}
Offer g = null;
Offer fail = new Offer(-1);
int exchange(int d){ // we assume 0 ≤ d
  Offer n = new Offer(d);
  if (CAS(g, null, n)){
    sleep(50);
    CAS(n.hole, null, fail)
    return n.hole.data;
  }
  Offer cur = g;
  bool success = CAS(cur.hole, null, n);
  CAS(g, cur, null);
  if (success)
    return cur.data;
  else return -1;
}
```

$$(P) \quad \underbrace{\texttt{exchange(3);}}_{t_1} \,||\, \underbrace{\texttt{exchange(10);}}_{t_2} \,||\, \underbrace{\texttt{exchange(7);}}_{t_3}$$



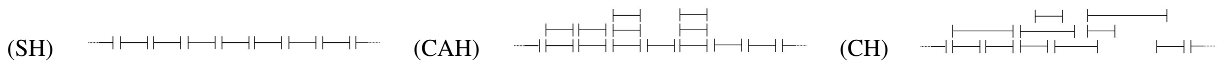


Figure 1: (E) a simplified Exchanger, (P) a client program, (H_1) a concurrent history, (H_2) an undesired sequential history, (H_3) a CA-history, a graphical depiction of a (SH) sequential history, a (CAH) CA-history, and a (CH) concurrent history.

jects are not modular. For example, the proof of the HSY-stack [3] mixes reasoning about the implementation of an (Exchanger-like) elimination array with its particular usage by the stack.

2. CONCURRENCY-AWARE LINEARIZABILITY (CAL)

Linearizability relates an implementation of a concurrent object with a sequential specification. Both the implementation and the specification are formalized as *prefix-closed sets of histories*. A history $H = \psi_1\psi_2\ldots$ is a sequence of methods invocations and responses. Specifications are given using *sequential histories* in which every response is immediately preceded by its matching invocation. Implementations, on the other hand, allow for arbitrary interleaving of actions by different threads, as long as the subsequence of actions of every thread is sequential. Informally, a concurrent object OS_C is linearizable with respect to a specification OS_A if every history H in OS_C can be *explained* by a history S in OS_A that "looks similar" to H. The similarity is formalized by a real-time relation $H \sqsubseteq_{RT} S$, which requires S to be a permutation of H preserving the per-thread order of actions and the order of non-overlapping operations.[1]

Why it is *impossible* to provide a sequential specification for Exchangers? Consider the client program P shown in Figure 1(P) which uses an Exchanger object. Figures 1(H_1-H_3) show three histories, where an exchange(n) operation returning value n' is depicted using an interval bounded by a "call(n)" and a "ret(n')" actions. Note that histories H_1 and H_3 might occur when P executes, but H_2 cannot.

History H_1 corresponds to the case where threads t_1 and t_2 exchange items 3 and 10 respectively and t_3 fails to pair-up. History H_2 is one possible sequential explanation of H_1. Using H_2 to explain H_1 raises the following problem: if H_2 is allowed by the specification then every prefix of H_2 must be allowed as-well. In particular, history H_2' in which only t_1 performs its operation should be allowed. Note that in H_2' a thread exchanges an item without finding a partner. Clearly, H_2' is an *undesired* behaviour. In fact, any sequential history that attempts to explain H_1 would allow for similar undesired behaviours. (In general, only executions in which all exchange() operations fail can be explained by sequential histories.) We conclude that any sequential specification of the Exchanger is either too restrictive or too loose.

We now turn to the definition of *concurrency-aware linearizability*. A key notion here is that of *concurrency-aware histories*. A history H is **concurrency-aware** (**CA-History**) if for any history H_1 such that $H = H_1\psi'\psi H_2$ if ψ' is a response and ψ is an invocation then the matching response of any invocation in $H_1\psi'$ is also in $H_1\psi'$. Note that a CA-history may contain concurrent operations. However, it ensures that such operations overlap pairwise. This provides the illusion that *all* concurrent operations are performed instantaneously at the same point in time. Figure 1 illustrates a sequential history (SH), a CA-history (CAH), and a concurrent history (CH). Note that every sequential history is a CA-history and every CA-history is a concurrent history, but not vice-versa. CAL extends linearizability by allowing specifications to be a (prefix-closed) set of CA-histories: A concurrent object OS_C is *CA-linearizable* with respect to a specification OS_A, if every history H in OS_C has a "similar-looking" *CA-history* S in OS_A. The "similar-looking" relation used in CAL is the same real-time order relation used to define linearizability [4]; the term *Concurrency-Aware Linearizability* emphasizes that the specification is comprised of **concurrency-aware** histories rather than a sequential ones.

Note that history H_3, depicted in Figure 1, is a *concurrency-aware history*. It describes the observable behavior as in H_1 while maintaining the same real-time order of operations and requiring

[1]For brevity, formal details, e.g., the treatment of *history completions*, are deferred to the Appendix.

that `exchange(3)` and `exchange(10)` execute concurrently and, seemingly, at the same point in time. Also note that every prefix of H_3 describes a behavior which is allowed by the implementation. Indeed, the behaviour of `Exchanger` objects can be specified precisely using CA-histories.

3. CONCLUSIONS AND FUTURE WORK

We present Concurrency-Aware Linearizabilty (CAL), a new correctness condition for an important class of CA-objects, concurrent objects whose behaviour does not have a sequential explanation. CA-objects exist in practice but currently do not have formal specifications. CAL allows providing accurate formal specifications for CA-objects using CA-histories, a restricted generalization of sequential histories. We believe that CAL can form the semantical basis for modular and reusable correctness proofs for CA-objects.

Acknowledgements. This research was supported by the EU project ADVENT.

References

[1] Abstraction for concurrent objects. *TCS*, 411(51-52), 2010.

[2] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Distributed Computing*. 2011.

[3] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.

[4] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[5] W. N. Scherer III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. *SCOOL*, 2005.

[6] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP*, 2006.

[7] W. N. Scherer III and M. L. Scott. Nonblocking concurrent data structures with condition synchronization. In *Distributed Computing*. 2004.

APPENDIX

In this section we formalize the notion of *contentaion-aware linearizability* (CAL). We assume infinite sets of object names $o \in \mathcal{O}$, method names $f \in \mathcal{F}$, and threads identifiers $t \in \mathcal{T}$.

DEFINITION 1. *An **object action** is either an **inocation** $\psi = (t, \text{inv}\, o.f(n))$ or a **response** $\psi' = (t', \text{res}(n')o'.f')$.*

Intuitively, an invocation $\psi = (t, \text{inv}\, o.f(n))$ means transfer of control from the client to the library, and response $\psi' = (t', \text{res}(n')o'.f')$ means the return of control to the invoking client. As in [4], the observable behaviour of a concurrent object is represented by a set of histories, which are sequences of *invocations* and *responses* of methods calls.

DEFINITION 2. *A **history** H is a finite sequence of invocations and responses. We use H_i to denote the ith action of H and $H|_t$ to denote the projection of H onto actions of thread t. We denote by _ an expression that is irrelevant and implicitly existentially quantified. A history is **sequential** if every response action is immediately preceded by a matching invocation. A history H is **well-formed** if invocations and responses are properly matched: for every thread t, $H|_t$ is sequential. A history is **complete** if it is well-formed and every invocation has a matching response (i.e, for every thread t, if $H|_t = \_\psi$ then ψ is a response). History H^c is a **completion** of a well-formed history H if it is complete and can be obtained from H by (possibly) extending H with some response actions and (possibly) removing some invocation actions. We denote by **complete(H)** the set of all completions of H.*

Linearizability is a relation between **object systems**, prefix-closed sets of well-formed histories. Following [4], we define it using the notion of real-time order.

DEFINITION 3. *The **real-time order** between actions of a well-formed history H is an irreflexive partial order $\prec_H$ on (indices of) object actions:*

$$\begin{aligned} &H_i \prec_H H_j \iff \\ &\quad \exists i \le i' < j' \le j.\, tid(H_i) = tid(H_{i'}) \wedge tid(H_j) = tid(H_{j'}) \wedge \\ &\qquad (tid(H_i) = tid(H_j) \vee H_{i'} = (\_, \text{res}\,\_) \wedge H_{j'} = (\_, \text{inv}\,\_)) \end{aligned}$$

*A history H **agrees** with the real-time order of a history S, denoted by $H \sqsubseteq_{RT} S$, if (i) for every thread t, $H|_t = S|_t$ and (ii) there is a bijection $\pi : \{1, \cdots, |H|\} \to \{1, \cdots, |S|\}$ such that*

$$\forall i.(H_i = S_{\pi(i)}) \wedge (\forall i, j. H_i \prec_H H_j \implies S_{\pi(i)} \prec_S S_{\pi(j)})\ .$$

Intuitively, history S "agrees" with H if in both histories every thread performs the same sequence of actions and the real-time order induced by H is a subset of that of S, i.e. $\prec_H \subseteq \prec_S$.

DEFINITION 4 (LINEARIZABILITY [4]). *Let OS_C and OS_A be object systems. We say that OS_C is **linearizable** with respect to OS_A if every history $H \in OS$ is* sequential *and*

$$\forall H \in OS_C.\, \exists H^c \in complete(H).\, \exists S \in OS_A.\, H^c \sqsubseteq_{RT} S\ .$$

We now turn on to formally define *CAL*. The key aspect is the notion of *CA-History*, which is the building block of new class of specifications which strictly extend sequential specifications. We use the notion of complete histories to provide an alternative definition for CA-histories.

DEFINITION 5 (CONCURRENCY-AWARE HISTORY). *A history H is **concurrency-aware (CA-History)** if for any history H_1 such that $H = H_1\psi'\psi H_2$ if ψ' is a response and ψ is an invocation then $H_1\psi'$ is a complete history. An object system is OS **concurrency-aware** if each $H \in OS$ is a concurrency-aware history.*

A concurrency-aware history allows for some operations to be executed concurrently by multiple threads. Moreover, it ensures that out of the set of threads that are operating concurrently, no thread will return before all other threads have invoked the operation (i.e. all operations must overlap pairwise). Figure 1 illustrates sequential history (SH), concurrency-aware history (CAH) and concurrent history (CH). Note that while every sequential history is CA, the opposite does not hold.

Extending Definition 4, Concurrency-aware linearizabilty of an object system is described using the $\sqsubseteq_{RT}$ relation to a concurrency-aware object system:

DEFINITION 6 (CONCURRENCY AWARE LINEARIZABILITY). *Let OS_C and OS_A be object systems. We say that OS_C is **concurrency-aware linearizable (CAL)** with respect to OS_A if*

$$\forall H \in OS_C.\, \exists H^c \in complete(H).\, \exists S \in OS_A.\, H^c \sqsubseteq_{RT} S$$

and every history $H \in OS_A$ is concurrency-aware.

Thus, CA-linearizable object is such that every interaction with it can be "explained" by a CA-history of some concurrency-aware object system OS_A.

Note that the same real-time order $\sqsubseteq_{RT}$ and notion of completions are used in the definitions of linearizability and concurrency-aware linearizability; the term concurrency-aware linearizability emphasizes that the specification is comprised of *concurrency-aware* histories, rather than a *sequential* ones.

Replicated Data Types: Specification, Verification, Optimality

Sebastian Burckhardt
Microsoft Research

Alexey Gotsman
IMDEA Software Institute

Hongseok Yang
University of Oxford

Marek Zawirski
INRIA & UPMC-LIP6

Abstract

Geographically distributed systems often rely on replicated eventually consistent data stores to achieve availability and performance. To resolve conflicting updates at different replicas, researchers and practitioners have proposed specialized consistency protocols, called replicated data types, that implement objects such as registers, counters, sets or lists. Reasoning about replicated data types has however not been on par with comparable work on abstract data types and concurrent data types, lacking specifications, correctness proofs, and optimality results.

To fill in this gap, we propose a framework for specifying replicated data types using relations over events and verifying their implementations using replication-aware simulations. We apply it to 7 existing implementations of 4 data types with nontrivial conflict-resolution strategies and optimizations (last-writer-wins register, counter, multi-value register and observed-remove set). We also present a novel technique for obtaining lower bounds on the worst-case space overhead of data type implementations and use it to prove optimality of 4 implementations. Finally, we show how to specify consistency of replicated stores with multiple objects axiomatically, in analogy to prior work on weak memory models. Overall, our work provides foundational reasoning tools to support research on replicated eventually consistent stores.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Replication; eventual consistency; weak memory

1. Introduction

To achieve availability and scalability, many networked computing systems rely on *replicated stores*, allowing multiple clients to issue operations on shared data on a number of *replicas*, which communicate changes to each other using message passing. For example, large-scale Internet services rely on *geo-replication*, which places data replicas in geographically distinct locations, and applications for mobile devices store replicas locally to support offline use. One benefit of such architectures is that the replicas remain locally available to clients even when network connections fail. Unfortunately, the famous CAP theorem [19] shows that such high **A**vailability and tolerance to network **P**artitions are incompatible with *strong* **C***onsistency*, i.e., the illusion of a single centralized replica handling all operations. For this reason, modern replicated stores often provide weaker forms of consistency, commonly dubbed *eventual consistency* [36]. 'Eventual' usually refers to the guarantee that

> if clients stop issuing update requests, then the replicas will eventually reach a consistent state. (1)

Eventual consistency is a hot research area, and new replicated stores implementing it appear every year [1, 13, 16, 18, 23, 27, 33, 34, 37]. Unfortunately, their semantics is poorly understood: the very term eventual consistency is a catch-all buzzword, and different stores claiming to be eventually consistent actually provide subtly different guarantees. The property (1), which is a form of *quiescent consistency*, is too weak to capture these. Although it requires the replicas to converge to the same state eventually, it doesn't say which one it will be. Furthermore, (1) does not provide any guarantees in realistic scenarios when updates never stop arriving. The difficulty of reasoning about the behavior of eventually consistent stores comes from a multitude of choices to be made in their design, some of which we now explain.

Allowing the replicas to be temporarily inconsistent enables eventually consistent stores to satisfy clients' requests from the local replica immediately, and broadcast the changes to the other replicas only after the fact, when the network connection permits this. However, this means that clients can concurrently issue conflicting operations on the same data item at different replicas; furthermore, if the replicas are out-of-sync, these operations will be applied to its copies in different states. For example, two users sharing an online store account can write two different zip codes into the delivery address; the same users connected to replicas with different views of the shopping cart can also add and concurrently remove the same product. In such situations the store needs to ensure that, after the replicas exchange updates, the changes by different clients will be merged and all conflicts will be resolved in a meaningful way. Furthermore, to ensure eventual consistency (1), the conflict resolution has to be uniform across replicas, so that, in the end, they converge to the same state.

The protocols achieving this are commonly encapsulated within *replicated data types* [1, 10, 16, 18, 31, 33, 34] that implement *objects*, such as registers, counters, sets or lists, with various conflict-resolution strategies. The strategies can be as simple as establishing a total order on all operations using timestamps and letting the last writer win, but can also be much more subtle. Thus, a data type can detect the presence of a conflict and let the client deal with it: e.g., the *multi-value register* used in Amazon's Dynamo key-value store [18] would return both conflicting zip codes in the above example. A data type can also resolve the conflict in an application-specific way. For example, the *observed-remove set* [7, 32] processes concurrent operations trying to add and remove the same element so that an add always wins, an outcome that may be appropriate for a shopping cart.

Replicated data type implementations are often nontrivial, since they have to maintain not only client-observable object state, but also *metadata* needed to detect and resolve conflicts and to handle network failures. This makes reasoning about their behavior challenging. The situation gets only worse if we consider multi-

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535848

ple replicated objects: in this case, asynchronous propagation of updates between replicas may lead to counterintuitive behaviors—*anomalies*, in database terminology. The following code illustrates an anomaly happening in real replicated stores [1, 18]:

$$\begin{array}{r||l} \text{Replica } r_1 \rightarrow x.\mathtt{wr}(\textit{post}) & i = y.\mathtt{rd} \text{ // } \textit{comment} \leftarrow \text{Replica } r_2 \\ y.\mathtt{wr}(\textit{comment}) & j = x.\mathtt{rd} \text{ // } \textit{empty} \end{array} \quad (2)$$

We have two clients reading from and writing to register objects x and y at two different replicas; i and j are client-local variables. The first client makes a post by writing to x at replica r_1 and then comments on the post by writing to y. After every write, replica r_1 might send a message with the update to replica r_2. If the messages carrying the writes of *post* to x and *comment* to y arrive to replica r_2 out of the order they were issued in, the second client can see the comment, but not the post. Different replicated stores may allow such an anomaly or not, and this has to be taken into account when reasoning about them.

In this paper, we propose techniques for reasoning about eventually consistent replicated stores in the following three areas.

1. Specification. We propose a comprehensive framework for specifying the semantics of replicated stores. Its key novel component is *replicated data type specifications* (§3), which provide the first way of specifying the semantics of replicated objects with advanced conflict resolution declaratively, like abstract data types [25]. We achieve this by defining the result of a data type operation not by a function of states, but of *operation contexts*—sets of events affecting the result of the operation, together with some relationships between them. We show that our specifications are sufficiently flexible to handle data types representing a variety of conflict-resolution strategies: last-write-wins register, counter, multi-value register and observed-remove set.

We then specify the semantics of a whole store with multiple objects, possibly of different types, by *consistency axioms* (§7), which constrain the way the store processes incoming requests in the style of weak shared-memory models [2] and thus define the anomalies allowed. As an illustration, we define consistency models used in existing replicated stores, including a weak form of eventual consistency [1, 18] and different kinds of causal consistency [23, 27, 33, 34]. We find that, when specialized to last-writer-wins registers, these specifications are very close to fragments of the C/C++ memory model [5]. Thus, our specification framework generalizes axiomatic shared-memory models to replicated stores with nontrivial conflict resolution.

2. Verification. We propose a method for proving the correctness of replicated data type implementations with respect to our specifications and apply it to seven existing implementations of the four data types mentioned above, including those with nontrivial optimizations. Reasoning about the implementations is difficult due to the highly concurrent nature of a replicated store, with multiple replicas simultaneously updating their object copies and exchanging messages. We address this challenge by proposing *replication-aware simulations* (§5). Like classical simulations from data refinement [21], these associate a concrete state of an implementation with its abstract description—structures on events, in our case. To combat the complexity of replication, they consider the state of an object at a single replica or a message in transit separately and associate it with abstract descriptions of only those events that led to it. Verifying an implementation then requires only reasoning about an instance of its code running at a single replica.

Here, however, we have to deal with another challenge: code at a single replica can access both the state of an object and a message at the same time, e.g., when updating the former upon receiving the latter. To reason about such code, we often need to rely on certain *agreement properties* correlating the abstract descriptions of the message and the object state. Establishing these properties requires global reasoning. Fortunately, we find that agreement properties needed to prove realistic implementations depend only on basic facts about their messaging behavior and can thus be established once for broad classes of data types. Then a particular implementation within such a class can be verified by reasoning purely locally.

By carefully structuring reasoning in this way, we achieve easy and intuitive proofs of single data type implementations. We then lift these results to stores with multiple objects of different types by showing how consistency axioms can be proved given properties of the transport layer and data type implementations (§7).

3. Optimality. Replicated data type designers strive to optimize their implementations; knowing that one is optimal can help guide such efforts in the most promising direction. However, proving optimality is challengingly broad as it requires quantifying over all possible implementations satisfying the same specification.

For most data types we studied, the primary optimization target is the size of the metadata needed to resolve conflicts or handle network failures. To establish optimality of metadata size, we present a novel method for proving lower bounds on the *worst-case metadata overhead* of replicated data types—the proportion of metadata relative to the client-observable content. The main idea is to find a large family of executions of an arbitrary correct implementation such that, given the results of data type operations from a certain fixed point in any of the executions, we can recover the previous execution history. This implies that, across executions, the states at this point are distinct and thus must have some minimal size.

Using our method, we prove that four of the implementations we verified have an optimal worst-case metadata overhead among all implementations satisfying the same specification. Two of these (counter, last-writer-wins register) are well-known; one (optimized observed-remove set [6]) is a recently proposed nontrivial optimization; and one (optimized multi-value register) is a small improvement of a known implementation [33] that we discovered during a failed attempt to prove optimality of the latter. We summarize all the bounds we proved in Fig. 10.

We hope that the theoretical foundations we develop will help in exploring the design space of replicated data types and replicated eventually consistent stores in a systematic way.

2. Replicated Data Types

We now describe our formal model for replicated stores and introduce *replicated data type implementations*, which implement operations on a single object at a replica and the protocol used by replicas to exchange updates to this object. Our formalism follows closely the models used by replicated data type designers [33].

A replicated store is organized as a collection of named ***objects*** $\mathsf{Obj} = \{x, y, z, \dots\}$. Each object is hosted at all ***replicas*** $r, s \in \mathsf{ReplicaID}$. The sets of objects and replicas may be infinite, to model their dynamic creation. Clients interact with the store by performing ***operations*** on objects at a specified replica. Each object $x \in \mathsf{Obj}$ has a ***type*** $\tau = \mathsf{type}(x) \in \mathsf{Type}$, whose ***type signature*** $(\mathsf{Op}_\tau, \mathsf{Val}_\tau)$ determines the set of supported operations Op_τ (ranged over by o) and the set of their return values Val_τ (ranged over by a, b, c, d). We assume that a special value $\bot \in \mathsf{Val}_\tau$ belongs to all sets Val_τ and is used for operations that return no value. For example, we can define a counter data type `ctr` and an integer register type `intreg` with operations for reading, incrementing or writing an integer a: $\mathsf{Val}_{\mathtt{ctr}} = \mathsf{Val}_{\mathtt{intreg}} = \mathbb{Z} \cup \{\bot\}$, $\mathsf{Op}_{\mathtt{ctr}} = \{\mathtt{rd}, \mathtt{inc}\}$ and $\mathsf{Op}_{\mathtt{intreg}} = \{\mathtt{rd}\} \cup \{\mathtt{wr}(a) \mid a \in \mathbb{Z}\}$.

We also assume sets $\mathsf{Message}$ of messages (ranged over by m) and timestamps $\mathsf{Timestamp}$ (ranged over by t). For simplicity, we let timestamps be positive integers: $\mathsf{Timestamp} = \mathbb{N}_1$.

DEFINITION 1. *A* ***replicated data type implementation*** *for a data type τ is a tuple $\mathcal{D}_\tau = (\Sigma, \vec{\sigma}_0, M, \mathsf{do}, \mathsf{send}, \mathsf{receive})$, where $\vec{\sigma}_0$:*

Figure 1. Illustrations of a concrete (a) and two abstract executions (b, c)

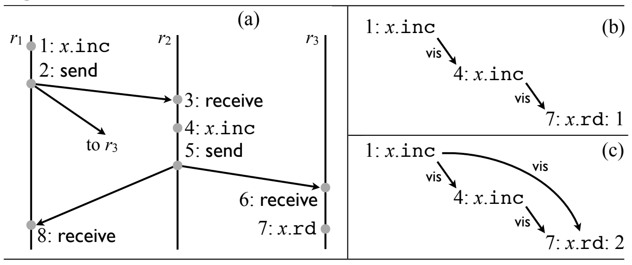


Figure 2. Three replicated data type implementations

(a) Op-based counter (`ctr`)

$$\begin{aligned}
\Sigma &= \mathbb{N}_0 \times \mathbb{N}_0 & \mathsf{do}(\mathtt{rd}, \langle a, d\rangle, t) &= (\langle a, d\rangle, a)\\
M &= \mathbb{N}_0 & \mathsf{do}(\mathtt{inc}, \langle a, d\rangle, t) &= (\langle a+1, d+1\rangle, \bot)\\
\vec{\sigma}_0 &= \lambda r.\, \langle 0, 0\rangle & \mathsf{send}(\langle a, d\rangle) &= (\langle a, 0\rangle, d)\\
& & \mathsf{receive}(\langle a, d\rangle, d') &= \langle a + d', d\rangle
\end{aligned}$$

(b) State-based counter (`ctr`)

$$\begin{aligned}
\Sigma &= \mathsf{ReplicaID} \times (\mathsf{ReplicaID} \to \mathbb{N}_0)\\
\sigma_0 &= \lambda r.\, \langle r, \lambda s.\, 0\rangle\\
M &= \mathsf{ReplicaID} \to \mathbb{N}_0\\
\mathsf{do}(\mathtt{rd}, \langle r, v\rangle, t) &= (\langle r, v\rangle, \textstyle\sum\{v(s) \mid s \in \mathsf{ReplicaID}\})\\
\mathsf{do}(\mathtt{inc}, \langle r, v\rangle, t) &= (\langle r, v[r \mapsto v(r)+1]\rangle, \bot)\\
\mathsf{send}(\langle r, v\rangle) &= (\langle r, v\rangle, v)\\
\mathsf{receive}(\langle r, v\rangle, v') &= \langle r, (\lambda s.\, \max\{v(s), v'(s)\})\rangle
\end{aligned}$$

(c) State-based last-writer-wins register (`intreg`)

$$\begin{aligned}
\Sigma &= \mathbb{Z} \times (\mathsf{Timestamp} \uplus \{0\})\\
\vec{\sigma}_0 &= \lambda r.\, \langle 0, 0\rangle\\
M &= \Sigma\\
\mathsf{do}(\mathtt{rd}, \langle a, t\rangle, t') &= (\langle a, t\rangle, a)\\
\mathsf{do}(\mathtt{wr}(a'), \langle a, t\rangle, t') &= \text{if } t < t' \text{ then } (\langle a', t'\rangle, \bot) \text{ else } (\langle a, t\rangle, \bot)\\
\mathsf{send}(\langle a, t\rangle) &= (\langle a, t\rangle, \langle a, t\rangle)\\
\mathsf{receive}(\langle a, t\rangle, \langle a', t'\rangle) &= \text{if } t < t' \text{ then } \langle a', t'\rangle \text{ else } \langle a, t\rangle
\end{aligned}$$

$\mathsf{ReplicaID} \to \Sigma$, $M \subseteq \mathsf{Message}$ *and*

$$\mathsf{do} : \mathsf{Op}_\tau \times \Sigma \times \mathsf{Timestamp} \to \Sigma \times \mathsf{Val}_\tau;$$
$$\mathsf{send} : \Sigma \to \Sigma \times M; \qquad \mathsf{receive} : \Sigma \times M \to \Sigma.$$

We denote a component of $\mathcal{D}_\tau$, such as do, by $\mathcal{D}_\tau$.do. A tuple $\mathcal{D}_\tau$ defines the class of implementations of objects with type τ, meant to be instantiated for every such object in the store. Σ is the set of states (ranged over by σ) used to represent the current state of the object, including metadata, at a single replica. The initial state at every replica is given by $\vec{\sigma}_0$.

$\mathcal{D}_\tau$ provides three methods that the rest of the store implementation can call at a given replica; we assume that these methods execute atomically. We visualize store executions resulting from repeated calls to the methods as in Fig. 1(a), by arranging the calls on several vertical timelines corresponding to replicas at which they occur and denoting the delivery of messages by diagonal arrows. In §4, we formalize them as sequences of transitions called *concrete executions* and define the store semantics by their sets; the intuition given by Fig. 1(a) should suffice for the following discussion.

A client request to perform an operation $o \in \mathsf{Op}_\tau$ triggers the call $\mathsf{do}(o, \sigma, t)$ (e.g., event 1 in Fig. 1(a)). This takes the current state $\sigma \in \Sigma$ of the object at the replica where the request is issued and a timestamp $t \in \mathsf{Timestamp}$ provided by the rest of the store implementation and produces the updated object state and the return value of the operation. The data type implementation can use the timestamp provided, e.g., to implement the last-writer-wins conflict-resolution strategy mentioned in §1, but is free to ignore it.

Nondeterministically, in moments when the network is able to accept messages, a replica calls send. Given the current state of the object at the replica, send produces a message in M to broadcast to all other replicas (event 2 in Fig. 1(a)); sometimes send also alters the state of the object. Using broadcast rather than point-to-point communication does not limit generality, since we can always tag messages with the intended receiver. Another replica that receives the message generated by send calls receive to merge the enclosed update into its copy of the object state (event 3 in Fig. 1(a)).

We now reproduce three replicated data type implementations due to Shapiro et al. [33].They fall into two categories: in ***op-based*** implementations, each message carries a description of the latest operations that the sender has performed, and in ***state-based*** implementations, a description of *all* operations it knows about.

Op-based counter (`ctr`). Fig. 2(a) shows an implementation of the `ctr` data type. A replica stores a pair $\langle a, d\rangle$, where a is the current value of the counter, and d is the number of increments performed since the last broadcast (we use angle brackets for tuples representing states and messages). The send method returns d and resets it; the receive method adds the content of the message to a. This implementation is correct, as long as each message is delivered exactly once (we show how to prove this in §5). Since `inc` operations commute, they never conflict: applying them in different orders at different replicas yields the same final state.

State-based counter (`ctr`). The implementation in Fig. 2(b) summarizes the currently known history by recording the contribution of every replica to the counter value separately (reminiscent of vector clocks [29]). A replica stores its identifier r and a vector v such that for each replica s the entry $v(s)$ gives the number of increments made by clients at s that have been received by r. A `rd` operation returns the sum of all entries in the vector. An `inc` operation increments the entry for the current replica. We denote by $v[i \mapsto j]$ the function that has the same value as v everywhere, except for i, where it has the value j. The send method returns the vector, and the receive method takes the maximum of each entry in the vectors v and v' given to it. This is correct because an entry for s in either vector reflects a prefix of the sequence of increments done at replica s. Hence, we know that $\min\{v(s), v'(s)\}$ increments by s are taken into account both in $v(s)$ and in $v'(s)$.

State-based last-writer-wins (LWW) register (`intreg`). Unlike counters, registers have update operations that are not commutative. To resolve conflicts, the implementation in Fig. 2 uses the last-writer-wins strategy, creating a total order on writes by associating a unique timestamp with each of them. A state contains the current value, returned by `rd`, and the timestamp at which it was written (initially, we have 0 instead of a timestamp). A $\mathtt{wr}(a')$ compares its timestamp t' with the timestamp t of the current value a and sets the value to the one with the highest timestamp. Note that here we have to allow for $t' < t$, since we do not make any assumptions about timestamps apart from uniqueness: e.g., the rest of the store implementation can compute them using physical or Lamport clocks [22]. We show how to state assumptions about timestamps in §4. The send method just returns the state, and the receive method chooses the winning value by comparing the timestamps in the current state and the message, like `wr`.

State-based vs. op-based. State-based implementations converge to a consistent state faster than op-based implementations because they are ***transitively delivering***, meaning that they can propagate updates indirectly. For example, when using the counter in Fig. 2(b), in the execution in Fig. 1(a) the read at r_3 (event 7) returns 2, even though the message from r_1 has not arrived yet, because r_3 learns about r_1's update via r_2. State-based implementations are also resilient against transport failures like message *loss, reordering, or duplication*. Op-based implementations require the replicated store using them to mask such failures (e.g., using message sequence numbers, retransmission buffers, or reorder buffers).

The potential weakness of state-based implementations is the size of states and messages, which motivates our examination of space optimality in §6. For example, we show that the counter in Fig. 2(b) is optimal, meaning that no counter implementation satisfying the same requirements (transitive delivery and resilience against message loss, reordering, and duplication) can do better.

3. Specifying Replicated Data Types and Stores

Consider the concrete execution in Fig. 1(a). What are valid return values for the read in event 7? Intuitively, 1 or 2 can be justifiable, but not 100. We now present a framework for specifying the expected outcome declaratively, without referring to implementation details. For example, we give a specification of a replicated counter that is satisfied by both implementations in Fig. 2(a, b).

In presenting the framework, we rely on the intuitive understanding of the way a replicated store executes given in §2. Later we define the store semantics formally (§4), which lets us state what it means for a store to satisfy our specifications (§4 and §7).

3.1 Abstract Executions and Specification Structure

We define our specifications on *abstract executions*, which include only user-visible events (corresponding to do calls) and describe the other information about the store processing in an implementation-independent form. Informally, we consider a concrete execution correct if it can be justified by an abstract execution satisfying the specifications that is "similar" to it and, in particular, has the same operations and return values.

Abstract executions are inspired by axiomatic definitions of weak shared-memory models [2]. In particular, we use their previously proposed reformulation with visibility and arbitration relations [13], which are similar to the reads-from and coherence relations from weak shared-memory models. We provide a comparison with shared-memory models in §7 and with [13] in §8.

DEFINITION 2. *An **abstract execution** is a tuple* $A = (E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$*, where*

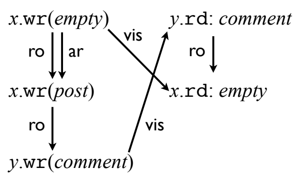
- $E \subseteq \mathsf{Event}$ *is a set of events from a countable universe* Event*;*
- *each event* $e \in E$ *describes a replica* $\mathsf{repl}(e) \in \mathsf{ReplicaID}$ *performing an operation* $\mathsf{oper}(e) \in \mathsf{Op}_{\mathsf{type}(\mathsf{obj}(e))}$ *on an object* $\mathsf{obj}(e) \in \mathsf{Obj}$*, which returns the value* $\mathsf{rval}(e) \in \mathsf{Val}_{\mathsf{type}(\mathsf{obj}(e))}$*;*
- $\mathsf{ro} \subseteq E \times E$ *is a **replica order**, which is a union of transitive, irreflexive and total orders on events at each replica;*
- $\mathsf{vis} \subseteq E \times E$ *is an acyclic **visibility relation** such that* $\forall e, f \in E.\, e \xrightarrow{\mathsf{vis}} f \implies \mathsf{obj}(e) = \mathsf{obj}(f)$*;*
- $\mathsf{ar} \subseteq E \times E$ *is an **arbitration relation**, which is a union of transitive, irreflexive and total orders on events on each object.*

We also require that ro*,* vis *and* ar *be well-founded.*

In the following, we denote components of A and similar structures as in $A.\mathsf{repl}$. We also use $(e, f) \in \mathsf{r}$ and $e \xrightarrow{\mathsf{r}} f$ interchangeably.

Informally, $e \xrightarrow{\mathsf{vis}} f$ means that f is aware of e and thus e's effect can influence f's return value. In implementation terms, this may be the case if the update performed by e has been delivered to the replica performing f before f is issued. The exact meaning of "delivered", however, depends on how much information messages carry in the implementation. For example, as we explain in §3.2, the return value of a read from a counter is equal to the number of inc operations visible to it. Then, as we formalize in §4, the abstract execution illustrated in Fig. 1(b) justifies the op-based implementation in Fig. 2(a) reading 1 in the concrete execution in Fig. 1(a). The abstract execution in Fig. 1(c) justifies the state-based implementation in Fig. 2(b) reading 2 due to transitive delivery (§2). There is no abstract execution that would justify reading 100.

The ar relation represents the ordering information provided by the store, e.g., via timestamps. On the right we show an abstract execution corresponding to a variant of the anomaly (2). The ar edge means that any replica that sees both writes to x should assume that *post* overwrites *empty*.



We give a store specification by two components, constraining abstract executions:

1. ***Replicated data type specifications*** determine return values of operations in an abstract execution in terms of its vis and ar relations, and thus define conflict-resolution policies for individual objects in the store. The specifications are the key novel component of our framework, and we discuss them next.
2. ***Consistency axioms*** constrain vis and ar and thereby disallow anomalies and extend the semantics of individual objects to that of the entire store. We defer their discussion to §7. See Fig. 13 for their flavor; in particular, COCV prohibits the anomaly above.

Each of these components can be varied separately, and our specifications will define the semantics of any possible combination. Given a specification of a store, we can determine whether a set of events can be observed by its users by checking if there is an abstract execution with this set of events satisfying the data type specifications and consistency axioms.

3.2 Replicated Data Type Specifications

In a sequential setting, the semantics of a data type τ can be specified by a function $\mathcal{S}_\tau : \mathsf{Op}_\tau^+ \to \mathsf{Val}_\tau$, which, given a non-empty sequence of operations performed on an object, specifies the return value of the last operation. For a register, read operations return the value of the last preceding write, or zero if there is no prior write. For a counter, read operations return the number of preceding increments. Thus, for any sequence of operations ξ:

$$\begin{aligned}
\mathcal{S}_{\texttt{intreg}}(\xi\,\texttt{rd}) &= a, \text{ if } \texttt{wr}(0)\,\xi = \xi_1\,\texttt{wr}(a)\,\xi_2 \text{ and} \\
&\qquad \xi_2 \text{ does not contain } \texttt{wr} \text{ operations;} \\
\mathcal{S}_{\texttt{ctr}}(\xi\,\texttt{rd}) &= (\text{the number of } \texttt{inc} \text{ operations in } \xi); \\
\mathcal{S}_{\texttt{intreg}}(\xi\,\texttt{inc}) &= \mathcal{S}_{\texttt{ctr}}(\xi\,\texttt{wr}(a)) \;=\; \bot.
\end{aligned}$$

In a replicated store, the story is more interesting. We specify a data type τ by a function $\mathcal{F}_\tau$, generalizing $\mathcal{S}_\tau$. Just like $\mathcal{S}_\tau$, this determines the return value of an operation based on prior operations performed on the object. However, $\mathcal{F}_\tau$ takes as a parameter not a sequence, but an *operation context*, which includes all we need to know about a store execution to determine the return value of a given operation o—the set E of all events that are visible to o, together with the operations performed by the events and visibility and arbitration relations on them.

DEFINITION 3. *An **operation context** for a data type* τ *is a tuple* $L = (o, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar})$*, where* $o \in \mathsf{Op}_\tau$*,* E *is a finite subset of* Event*,* $\mathsf{oper} : E \to \mathsf{Op}_\tau$*,* $\mathsf{vis} \subseteq E \times E$ *is acyclic and* $\mathsf{ar} \subseteq E \times E$ *is transitive, irreflexive and total.*

We can extract the context of an event $e \in A.E$ in an abstract execution A by selecting all events visible to it according to $A.\mathsf{vis}$:

$$\mathsf{ctxt}(A, e) = (A.\mathsf{oper}(e), G, (A.\mathsf{oper})|_G, (A.\mathsf{vis})|_G, (A.\mathsf{ar})|_G),$$

where $G = (A.\mathsf{vis})^{-1}(e)$ and $\cdot|_G$ is the restriction to events in G. Thus, in the abstract execution in Fig. 1(b), the operation context of the read from x includes only one increment event; in the execution in Fig. 1(c) it includes two.

DEFINITION 4. *A **replicated data type specification** for a type* τ *is a function* $\mathcal{F}_\tau$ *that, given an operation context* L *for* τ*, specifies a return value* $\mathcal{F}_\tau(L) \in \mathsf{Val}_\tau$*.*

Note that $\mathcal{F}_\tau(o, \emptyset, \ldots)$ returns the value resulting from performing o on the initial state for the data type (e.g., 0 for the LWW-register).

We specify multiple data types used in a replicated store by a partial function $\mathbb{F}$ mapping them to data type specifications.

DEFINITION 5. *An abstract execution A **satisfies** $\mathbb{F}$, written $A \models \mathbb{F}$, if the return value of every event in A is computed on its context by the specification for the type of the object the event accesses:*

$$\forall e \in A.E.\ (A.\mathsf{rval}(e) = \mathbb{F}(\mathsf{type}(A.\mathsf{obj}(e)))(\mathsf{ctxt}(A, e))).$$

We specify a whole store by $\mathbb{F}$ and a set of consistency axioms (§7). This lets us determine if its users can observe a given set of events by checking if there is an abstract execution with these events that satisfies $\mathbb{F}$ according to the above definition, as well as the axioms.

Note that $\mathcal{F}_\tau$ is deterministic. This does not mean that so is an outcome of an operation on a store; rather, that all the non-determinism arising due to its distributed nature is resolved by vis and ar in the context passed to $\mathcal{F}_\tau$. These relations are chosen arbitrarily subject to consistency axioms. Due to the determinacy property, two events that perform the same operation and see the same set of events produce the same return values. As we show in §7, this property ensures that our specifications can formalize eventual consistency in the sense of (1).

We now give four examples of data type specifications, corresponding to the four conflict-resolution strategies mentioned in §1 and §2: (1) operations commute, so no conflicts arise; (2) last writer wins; (3) all conflicting values are returned; and (4) conflicts are resolved in an application-specific way. We start by specifying the data types whose implementations we presented in §2.

1. Counter (`ctr`) is defined by

$$\begin{aligned} \mathcal{F}_{\texttt{ctr}}(\texttt{inc}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) &= \bot; \\ \mathcal{F}_{\texttt{ctr}}(\texttt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) &= \big|\{e \in E \mid \mathsf{oper}(e) = \texttt{inc}\}\big|. \end{aligned} \tag{3}$$

Thus, according to Def. 5 the executions in Fig. 1(b) and 1(c) satisfy the counter specification: both 1 and 2 are valid return values for the read from x when there are two concurrent increments.

2. LWW-register (`intreg`) is defined by

$$\mathcal{F}_{\texttt{intreg}}(o, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \mathcal{S}_{\texttt{intreg}}(E^{\mathsf{ar}} o), \tag{4}$$

where E^{ar} denotes the sequence obtained by ordering the operations performed by the events in E according to ar. Thus, the return value is determined by establishing a total order of the visible operations and applying the regular sequential semantics. For example, by Def. 5 in the example execution from §3.1 the read from x has to return *empty*; if we had a vis edge from the write of *post* to the read from x, then the read would have to return *post*. As we show in §7, weak shared-memory models are obtained by specializing our framework to stores with only LWW-registers.

We can obtain a concurrent semantics $\mathcal{F}_\tau$ of any data type τ based on its sequential semantics $\mathcal{S}_\tau$ similarly to (4). For example, $\mathcal{F}_{\texttt{ctr}}$ defined above is equivalent to what we obtain using this generic construction. The next two examples go beyond this.

3. Multi-value register (`mvr`). This register [1, 18] has the same operations as the LWW-register, but its reads return a set of values:

$$\begin{aligned} \mathcal{F}_{\texttt{mvr}}(\texttt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E.\, \mathsf{oper}(e) = \texttt{wr}(a) \\ \wedge \neg\exists f \in E.\, \mathsf{oper}(f) = \texttt{wr}(\_) \wedge e \xrightarrow{\mathsf{vis}} f\}. \end{aligned}$$

(We write _ for an expression whose value is irrelevant.) A read returns the values written by currently conflicting writes, defined as those that are not superseded in vis by later writes; ar is not used. For example, a `rd` would return $\{2, 3\}$ in the context on the right.

$$\begin{tikzcd} & \texttt{wr}(0) \arrow[d, "\mathsf{vis}"] & \\ & \texttt{wr}(1) \arrow[dl, "\mathsf{vis}"'] \arrow[dr, "\mathsf{vis}"] & \\ \texttt{wr}(2) & & \texttt{wr}(3) \end{tikzcd}$$

Figure 3. The set of configurations Config and the transition relation $\longrightarrow_{\mathbb{D}}$: Config $\times$ Event $\times$ Config for a data type library $\mathbb{D}$. We use $e : \{h_1 = u_1, h_2 = u_2\}$ to abbreviate $h_1(e) = u_1$ and $h_2(e) = u_2$. We uncurry $R \in$ RState where convenient.

$$\begin{aligned} \mathsf{Obj}_\tau &= \{x \in \mathsf{Obj} \mid \mathsf{type}(x) = \tau\} \\ \mathsf{RState} &= \textstyle\bigcup_{X \subseteq \mathsf{Obj}} \prod_{x \in X} (\mathsf{ReplicaID} \to \mathbb{D}(\mathsf{type}(x)).\Sigma) \\ \mathsf{TState} &= \mathsf{MessageID} \rightharpoonup \textstyle\bigcup_{\tau \in \mathsf{Type}} (\mathsf{ReplicaID} \times \mathsf{Obj}_\tau \times \mathbb{D}(\tau).M) \\ \mathsf{Config} &= \mathsf{RState} \times \mathsf{TState} \end{aligned}$$

$$\frac{\begin{array}{c}\mathbb{D}(\mathsf{type}(x)).\mathsf{do}(o, \sigma, t) = (\sigma', a) \\ e : \{\mathsf{act} = \mathsf{do},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{oper} = o,\ \mathsf{time} = t,\ \mathsf{rval} = a\}\end{array}}{(R[(x, r) \mapsto \sigma],\ T) \xrightarrow{e}_{\mathbb{D}} (R[(x, r) \mapsto \sigma'],\ T)}$$

$$\frac{\begin{array}{c}\mathbb{D}(\mathsf{type}(x)).\mathsf{send}(\sigma) = (\sigma', m) \qquad \mathit{mid} \notin \mathsf{dom}(T) \\ e : \{\mathsf{act} = \mathsf{send},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{msg} = \mathit{mid}\}\end{array}}{(R[(x, r) \mapsto \sigma],\ T) \xrightarrow{e}_{\mathbb{D}} (R[(x, r) \mapsto \sigma'],\ T[\mathit{mid} \mapsto (r, x, m)])}$$

$$\frac{\begin{array}{c}\mathbb{D}(\mathsf{type}(x)).\mathsf{receive}(\sigma, m) = \sigma' \qquad r \neq r' \\ e : \{\mathsf{act} = \mathsf{receive},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{srepl} = r',\ \mathsf{msg} = \mathit{mid}\}\end{array}}{\begin{array}{l}(R[(x, r) \mapsto \sigma],\ T[\mathit{mid} \mapsto (r', x, m)]) \xrightarrow{e}_{\mathbb{D}} \\ \qquad\qquad (R[(x, r) \mapsto \sigma'],\ T[\mathit{mid} \mapsto (r', x, m)])\end{array}}$$

4. Observed-remove set (`orset`). How do we specify a replicated set of integers? The operations of adding and removing different elements commute and thus do not conflict. Conflicts arise from concurrently adding and removing the same element. For example, we need to decide what `rd` will return as the contents of the set in the context $(\texttt{rd}, \{e, f\}, \mathsf{oper}, \mathsf{vis}, \mathsf{ar})$, where $\mathsf{oper}(e) = \texttt{add}(42)$ and $\mathsf{oper}(f) = \texttt{remove}(42)$. If we use the generic construction from the LWW-register, the result will depend on the arbitration relation: $\emptyset$ if $e \xrightarrow{\mathsf{ar}} f$, and $\{42\}$ otherwise. An application may require a more consistent behavior, e.g., that an add operation always win against concurrent remove operations. Observed-remove (OR) set [7, 32] achieves this by mandating that `remove` operations cancel only the `add` operations that are visible to them:

$$\begin{aligned} \mathcal{F}_{\texttt{orset}}(\texttt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E.\, \mathsf{oper}(e) = \texttt{add}(a) \\ \wedge \neg\exists f \in E.\, \mathsf{oper}(f) = \texttt{remove}(a) \wedge e \xrightarrow{\mathsf{vis}} f\}, \end{aligned} \tag{5}$$

In the above operation context `rd` will return $\emptyset$ if $e \xrightarrow{\mathsf{vis}} f$, and $\{42\}$ otherwise. The rationale is that, in the former case, `add`(42) and `remove`(42) are not concurrent: the user who issued the `remove` knew that 42 was in the set and thus meant to remove it. In the latter case, the two operations are concurrent and thus `add` wins.

As the above examples illustrate, our specifications can describe the semantics of data types and their conflict-resolution policies declaratively, without referring to the internals of their implementations. In this sense the specifications generalize the concept of an *abstract data type* [25] to the replicated setting.

4. Store Semantics and Data Type Correctness

A ***data type library*** $\mathbb{D}$ is a partial mapping from types τ to data type implementations $\mathbb{D}(\tau)$ from Def. 1. We now define the semantics of a replicated store with a data type library $\mathbb{D}$ as a set of its *concrete executions*, previously introduced informally by Fig. 1(a). We then state what it means for data type implementations of §2 to satisfy their specifications of §3.2 by requiring their concrete executions to be justified by abstract ones. In §7 we generalize this to the correctness of the whole store with multiple object with respect to both data type specifications and consistency axioms.

Semantics. We define the semantics using the relation $\longrightarrow_{\mathbb{D}}$: Config $\times$ Event $\times$ Config in Fig. 3, which describes a single step of the store execution. The relation transforms ***configurations*** $(R, T) \in \mathsf{Config}$ describing the store state: R gives the object state at each replica, and T the set of messages in transit between them, each identified by a message identifier $mid \in \mathsf{MessageID}$. A message is annotated by the origin replica and the object to which it pertains. We allow the store to contain only some objects from Obj and thus allow R to be partial on them. We use a number of functions on events, such as act, obj, etc., to record the information about the corresponding transitions, so that $\longrightarrow_{\mathbb{D}}$ is implicitly parameterized by them; we give their full list in Def. 6 below.

The first rule in Fig. 3 describes a replica r performing an operation o on an object x using the do method of the corresponding data type implementation. We record the return value using the function rval. To communicate the change to other replicas, we can at any time perform a transition defined by the second rule, which puts a new message m created by a call to send into the set of messages in transit. The third rule describes the delivery of such a message to a replica r other than the origin replica r', which triggers a call to $\mathsf{receive}$. Note that the relation $\longrightarrow_{\mathbb{D}}$ does not make any assumptions about message delivery: messages can be delivered in any order, multiple times, or not at all. These assumptions can be introduced separately, as we show later in this section. A concrete execution can be thought of as a finite or infinite sequence of transitions:

$$(R_0, T_0) \xrightarrow{e_1} (R_1, T_1) \xrightarrow{e_2} \dots \xrightarrow{e_n} (R_n, T_n) \dots,$$

where all events e_i are distinct. To ease mapping between concrete and abstract executions in the future, we formalize it as a structure on events, similarly to Def. 2.

DEFINITION 6. *A **concrete execution** of a store with a data type library $\mathbb{D}$ is a tuple*

$$C = (E, \mathsf{eo}, \mathsf{pre}, \mathsf{post}, \mathsf{act}, \mathsf{obj}, \mathsf{repl}, \mathsf{oper}, \mathsf{time}, \mathsf{rval}, \mathsf{msg}, \mathsf{srepl}).$$

*Here $E \subseteq \mathsf{Event}$, the **execution order** eo is a well-founded, transitive, irreflexive and total order on E, relating the events according to the order of the transitions they describe,* time *is injective and* $\mathsf{pre}, \mathsf{post} : E \to \mathsf{Config}$ *form a valid sequence of transitions:*

$$(\forall e \in E.\, \mathsf{pre}(e) \xrightarrow{e}_{\mathbb{D}} \mathsf{post}(e)) \wedge$$
$$(\forall e, f \in E.\, e \xrightarrow{\mathsf{eo}} f \wedge \neg\exists g.\, e \xrightarrow{\mathsf{eo}} g \xrightarrow{\mathsf{eo}} f \implies \mathsf{post}(e) = \mathsf{pre}(f)).$$

We have omitted the types of functions on events, which are easily inferred from Fig. 3: e.g., $\mathsf{act} : E \to \{\mathsf{do}, \mathsf{send}, \mathsf{receive}\}$ and $\mathsf{time} : E \rightharpoonup \mathsf{Timestamp}$, defined only on e with $\mathsf{act}(e) = \mathsf{do}$.

We denote the initial configuration of C by $\mathsf{init}(C) = C.\mathsf{pre}(e_0)$, where e_0 is the minimal event in $C.\mathsf{eo}$. If $C.E$ is finite, we denote the final configuration of C by $\mathsf{final}(C) = C.\mathsf{post}(e_{\mathbf{f}})$, where $e_{\mathbf{f}}$ is the maximal event in $C.\mathsf{eo}$. The ***semantics*** $[\![\mathbb{D}]\!]$ of $\mathbb{D}$ is the set of all its concrete executions C that start in a configuration with an empty set of messages and all objects in initial states, i.e.,

$$\exists X \subseteq \mathsf{Obj}.\, \mathsf{init}(C) = ((\lambda x \in X.\, \mathbb{D}(\mathsf{type}(x)).\vec{\sigma}_0), [\,]),$$

where $[\,]$ is the everywhere-undefined function.

Transport layer specifications. Data type implementations such as the op-based counter in Fig. 2(a) can rely on some guarantees concerning the delivery of messages ensured by the rest of the store implementation. They may similarly assume certain properties of timestamps other than uniqueness (guaranteed by the injectivity of time in Def. 6). We take such assumptions into account by admitting only a subset of executions from $[\![\mathbb{D}]\!]$ that satisfy a ***transport layer specification*** $\mathcal{T}$, which is a predicate on concrete executions. Thus, we consider a ***replicated store*** to be defined by a pair $(\mathbb{D}, \mathcal{T})$ and the set of its executions be $[\![\mathbb{D}]\!] \cap \mathcal{T}$.

Even though our definition of $\mathcal{T}$ lets it potentially restrict data type implementation internals, the particular instantiations we use only restrict message delivery and timestamps. For technical reasons, we assume that $\mathcal{T}$ always satisfies certain closure properties: for every $C \in \mathcal{T}$, the projection of C onto events on a given object or a subset of events forming a prefix in the eo order is also in $\mathcal{T}$.

As an example, we define a transport layer specification ensuring that a message is delivered to any single replica at most once, as required by the implementation in Fig. 2(a). Let the ***delivery relation*** $\mathsf{del}(C) \in C.E \times C.E$ pair events sending and receiving the same message:

$$e \xrightarrow{\mathsf{del}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{act}(e) = \mathsf{send} \wedge$$
$$C.\mathsf{act}(f) = \mathsf{receive} \wedge C.\mathsf{msg}(e) = C.\mathsf{msg}(f).$$

Then the desired condition on concrete executions C is

$$\forall e, f, g \in C.E.\, e \xrightarrow{\mathsf{del}(C)} f \wedge e \xrightarrow{\mathsf{del}(C)} g \wedge$$
$$C.\mathsf{repl}(f) = C.\mathsf{repl}(g) \implies f = g. \quad (\mathsf{T\text{-}Unique})$$

Data type implementation correctness. We now state what it means for an implementation $\mathcal{D}_\tau$ of type τ from Def. 1 to satisfy a specification $\mathcal{F}_\tau$ from Def. 4. To this end, we consider the behavior of $\mathcal{D}_\tau$ under the "most general" client and transport layer, performing all possible operations and message deliveries. Formally, let $[\![\mathcal{D}_\tau]\!]$ be the set of executions $C \in [\![[\tau \mapsto \mathcal{D}_\tau]]\!]$ of a store containing a single object x of a type τ with the implementation $\mathcal{D}_\tau$, i.e., $\mathsf{init}(C) = (R, [\,])$ for some R such that $\mathsf{dom}(R) = \{x\}$.

Then $\mathcal{D}_\tau$ should satisfy $\mathcal{F}_\tau$ under a transport specification $\mathcal{T}$ if for every concrete execution $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ we can find a "similar" abstract execution satisfying $\mathcal{F}_\tau$ and, in particular, having the same operations and return values. As it happens, all components of the abstract execution except visibility are straightforwardly determined by C; as explained in §3.1, we have some freedom in choosing visibility. We define the choice using a ***visibility witness*** $\mathcal{V}$, which maps a concrete execution $C \in [\![\mathcal{D}_\tau]\!]$ to an acyclic relation on $(C.E)|_{\mathsf{do}}$ defining visibility (here $\cdot|_{\mathsf{do}}$ is the restriction to events e with $C.\mathsf{act}(e) = \mathsf{do}$). Let

$$e \xrightarrow{\mathsf{ro}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{repl}(e) = C.\mathsf{repl}(f);$$
$$e \xrightarrow{\mathsf{ar}(C)} f \iff e, f \in (C.E)|_{\mathsf{do}} \wedge$$
$$C.\mathsf{obj}(e) = C.\mathsf{obj}(f) \wedge C.\mathsf{time}(e) < C.\mathsf{time}(f).$$

Then the abstract execution justifying $C \in [\![\mathcal{D}_\tau]\!]$ is defined by

$$\mathsf{abs}(C, \mathcal{V}) = (C.E|_{\mathsf{do}}, E.\mathsf{repl}|_{\mathsf{do}}, E.\mathsf{obj}|_{\mathsf{do}}, E.\mathsf{oper}|_{\mathsf{do}}, E.\mathsf{rval}|_{\mathsf{do}},$$
$$\mathsf{ro}(C)|_{\mathsf{do}}, \mathcal{V}(C), \mathsf{ar}(C)).$$

DEFINITION 7. *A data type implementation $\mathcal{D}_\tau$ **satisfies** a specification $\mathcal{F}_\tau$ with respect to $\mathcal{V}$ and $\mathcal{T}$, written $\mathcal{D}_\tau$ $\mathsf{sat}[\mathcal{V}, \mathcal{T}]$ $\mathcal{F}_\tau$, if $\forall C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}.\, (\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau])$, where $\models$ is defined in Def. 5.*

As we explained informally in §3.1, the visibility witness depends on how much information the implementation puts into messages. Since state-based implementations, such as the ones in Fig. 2(b, c), are transitively delivering (§2), for them we use the witness $\mathcal{V}^{\mathsf{state}}(C) = (\mathsf{ro}(C) \cup \mathsf{del}(C))^+|_{\mathsf{do}}$. By the definition of $\mathsf{ro}(C)$ and $\mathsf{del}(C)$, $(\mathsf{ro}(C) \cup \mathsf{del}(C))$ is acyclic, so $\mathcal{V}^{\mathsf{state}}$ is well-defined. State-based implementations do not make any assumptions about the transport layer: in this case we write $\mathcal{T} = \mathsf{T\text{-}Any}$. In contrast, op-based implementations, such as the one in Fig. 2(a), require $\mathcal{T} = \mathsf{T\text{-}Unique}$. Since such implementations are not transitively delivering, the witness $\mathcal{V}^{\mathsf{state}}$ is not appropriate for them. We could attempt to define a witness for them by straightforwardly lifting the delivery relation:

$$\mathcal{V}^{\mathsf{op}}(C) = \mathsf{ro}(C)|_{\mathsf{do}} \cup \{(e, f) \mid e, f \in (C.E)|_{\mathsf{do}} \wedge$$
$$\exists e', f'.\, e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f\}.$$

However, we need to be more careful, since for op-based implementations $e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f$ does not ensure that the update of e is taken into account by f: if there is another send event e'' in between e and e', then e'' will capture the update of e and e' will not. Hence, we define the witness as:

$$\begin{aligned}\mathcal{V}^{\mathsf{op}}(C) = {} & \mathsf{ro}(C)|_{\mathsf{do}} \cup \{(e, f) \mid e, f \in (C.E)|_{\mathsf{do}} \\ & \wedge \exists e', f'.\, e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f \\ & \wedge \neg\exists e''.\, e \xrightarrow{\mathsf{ro}(C)} e'' \xrightarrow{\mathsf{ro}(C)} e' \wedge C.\mathsf{act}(e'') = \mathsf{send}\}.\end{aligned}$$

We next present a method for proving data type implementation correctness in the sense of Def. 7. In §7 we lift this to stores with multiple objects and take into account consistency axioms.

5. Proving Data Type Implementations Correct

The straightforward approach to proving correctness in the sense of Def. 7 would require us to consider global store configurations in executions C, including object states at all replicas and all messages in transit, making the reasoning non-modular and unintuitive. To deal with this challenge, we focus on a single component of a store configuration using *replication-aware simulation relations* $\mathcal{R}_r$ and $\mathcal{M}$, analogous to simulation (aka coupling) relations used in data refinement [21]. The $\mathcal{R}_r$ relation associates the object state at a replica r with an abstract execution that describes *only those events that led to this state*; $\mathcal{M}$ does the same for a message. For example, when proving $\mathcal{D}_{\mathtt{ctr}}$ in Fig. 2(b) with respect to $\mathcal{F}_{\mathtt{ctr}}$ in (3), $\mathcal{M}$ associates a message carrying a vector v with executions in which each replica s makes $v(s)$ increments. As part of a proof of $\mathcal{D}_\tau$, we require checking that the effect of its methods, such as $\mathcal{D}_\tau$.do, can be simulated by appropriately transforming related abstract executions while preserving the relations. We define these transformations using *abstract methods* $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$ and $\mathsf{receive}^\sharp$ as illustrated in Fig. 4(a, b). For example, if a replica r executes $\mathcal{D}_\tau$.do from a state σ related by $\mathcal{R}_r$ to an abstract execution I (we explain the use of I instead of A later), we need to find an I' related by $\mathcal{R}_r$ to the resulting state σ'. We also need to check that the value returned by $\mathcal{D}_\tau$.do on σ is equal to that returned by $\mathcal{F}_\tau$ on I.

These conditions consider the behavior of an implementation method on a single state and/or message and its effect on only the relevant part of the abstract execution. However, by localizing the reasoning in this way, we lose some global information that is actually required to verify realistic implementations. In particular, this occurs when discharging the obligation for receive in Fig. 4(b). Taking a global view, σ and m there are meant to come from the same configuration in a concrete execution C; correspondingly, I and J are meant to be fragments of the same abstract execution $\mathsf{abs}(C, \mathcal{V})$. In this context we may know certain ***agreement properties*** correlating I and J, e.g., that the union of their visibility relations is itself a well-formed visibility relation and is thus acyclic. Establishing them requires global reasoning about whole executions C and $\mathsf{abs}(C, \mathcal{V})$. Fortunately, we find that this can be done knowing only the abstract methods, not the implementation $\mathcal{D}_\tau$. Furthermore, these methods state basic facts about the messaging behavior of implementations and are thus common to broad classes of them, such as state-based or op-based. This allows us to establish agreement properties using global reasoning once for a given class of implementations; at this stage we can also benefit from the transport layer specification $\mathcal{T}$ and check that the abstract methods construct visibility according to the given witness $\mathcal{V}$. Then a particular implementation within the class can be verified by discharging local obligations, such as those in Fig. 4(a, b), while assuming agreement properties. This yields easy and intuitive proofs.

To summarize, we deal with the challenge posed by a distributed data type implementation by decomposing reasoning about it into global reasoning done once for a broad class of implementations and local implementation-specific reasoning. We start by presenting the general form of obligations to be discharged for a single implementation within a certain class (§5.1) and the particular form they take for the class of state-based implementations (§5.2), together with some examples (§5.3). We then formulate the obligations to be discharged for a class of implementations (§5.4), which in particular, establish the agreement properties assumed in the per-implementation obligations. In [12, §B], we give the obligations for op-based implementations, together with a proof of the counter in Fig. 2(a). An impatient reader can move on to §6 after finishing §5.3, and come back to §5.4 later.

Figure 4. Diagrams illustrating replication-aware simulations

(a)

$$\begin{tikzcd} I \arrow[r, dotted, "\mathsf{do}^\sharp"] \arrow[d, leftrightarrow, "\mathcal{R}_r"'] & I' \arrow[d, leftrightarrow, dotted, "\mathcal{R}_r"] \\ \sigma \arrow[r, "\mathcal{D}_\tau.\mathsf{do}"'] & \sigma' \end{tikzcd}$$

(b)

$$\begin{tikzcd} (I, J) \arrow[r, dotted, "\mathsf{receive}^\sharp"] \arrow[d, leftrightarrow, "\mathcal{R}_r \times \mathcal{M}"] & I' \arrow[d, leftrightarrow, dotted, "\mathcal{R}_r"] \\ (\sigma, m) \arrow[r, "\mathcal{D}_\tau.\mathsf{receive}"'] & \sigma' \end{tikzcd}$$

(c)

$$\begin{tikzcd} D \arrow[r, dotted, "\mathsf{step}(C', e, D)"] \arrow[d, leftrightarrow, "\mathcal{G}"'] & D' \arrow[d, leftrightarrow, dotted, "\mathcal{G}"] \\ C \arrow[r, "e"'] & C' \end{tikzcd}$$

Since Def. 7 considers only single-object executions, we fix an object x of type τ and consider only concrete and abstract executions over x, whose sets we denote by $\mathsf{CEx}[x]$ and $\mathsf{AEx}[x]$.

5.1 Replication-Aware Simulations

As is typical for simulation-based proofs, we need to use auxiliary state to record information about the computation history. For this reason, actually our simulation relations associate a state or a message with an ***instrumented execution***—a pair $(A, \mathsf{info}) \in \mathsf{IEx}$ of an abstract execution $A \in \mathsf{AEx}[x]$ and a function $\mathsf{info} : A.E \to \mathsf{AInfo}$, tagging events with auxiliary information from a set AInfo. As we show below, AInfo can be chosen once for a class of data type implementations: e.g., $\mathsf{AInfo} = \mathsf{Timestamp}$ for state-based ones (§5.2). We use I and J to range over instrumented executions and shorten, e.g., $I.A.E$ to $I.E$. For a partial function h we write $h(x)\!\downarrow$ for $x \in \mathsf{dom}(h)$, and adopt the convention that $h(x) = y$ implies $h(x)\!\downarrow$.

DEFINITION 8. *A **replication-aware simulation** between $\mathcal{D}_\tau$ and $\mathcal{F}_\tau$ with respect to* info *and **abstract methods*** $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$ *and* $\mathsf{receive}^\sharp$ *is a collection of relations* $\{\mathcal{R}_r, \mathcal{M} \mid r \in \mathsf{ReplicaID}\}$ *satisfying the conditions in Fig. 5.*

Here info and abstract methods are meant to be fixed for a given class of implementations, such as state or op-based. To prove a particular implementation within this class, one needs to find simulation relations satisfying the conditions in Fig. 5. For example, as we show in §5.3, the following relation lets us prove the correctness of the counter in Fig. 2(b) with respect to info and abstract methods appropriate for state-based implementations:

$$\begin{aligned}&\langle s, v\rangle\ [\mathcal{R}_r]\ I \iff (r = s)\ \wedge\ (v\ [\mathcal{M}]\ I); \\ &v\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff \\ &\qquad \forall s.\, v(s) = \big|\{e \in E \mid \mathsf{oper}(e) = \mathtt{inc} \wedge \mathsf{repl}(e) = s\}\big|.\end{aligned} \tag{6}$$

INIT in Fig. 5 associates the initial state at a replica r with the execution having an empty set of events. DO, SEND and RECEIVE formalize the obligations illustrated in Fig. 4(a, b). Note that $\mathsf{do}^\sharp$ is parameterized by an event e (required to be fresh in instantiations) and the information about the operation performed.

The abstract methods are partial and the obligations in Fig. 5 *assume* that their applications are defined. When instantiating $\mathsf{receive}^\sharp$ for a given class of implementations, we let it be defined only when its arguments satisfy the agreement property for this class, which we establish separately (§5.4). While doing this, we can also establish some ***execution invariants***, holding of single ex-

Figure 5. Definition of a replication-aware simulation $\{\mathcal{R}_r, \mathcal{M}\}$ between $\mathcal{D}_\tau$ and $\mathcal{F}_\tau$. All free variables in each condition are implicitly universally quantified and have the following types: $\sigma, \sigma' \in \Sigma$, $m \in M$, $I, I', J \in \mathsf{IEx}$, $e \in \mathsf{Event}$, $r \in \mathsf{ReplicaID}$, $o \in \mathsf{Op}_\tau$, $a \in \mathsf{Val}_\tau$, $t \in \mathsf{Timestamp}$.

$$\mathcal{R}_r \subseteq \Sigma \times \mathsf{IEx},\ r \in \mathsf{ReplicaID}; \qquad \mathcal{M} \subseteq M \times \mathsf{IEx}$$

$$\mathsf{do}^\sharp : \mathsf{IEx} \times \mathsf{Event} \times \mathsf{ReplicaID} \times \mathsf{Op}_\tau \times \mathsf{Val}_\tau \times \mathsf{Timestamp} \rightharpoonup \mathsf{IEx}$$
$$\mathsf{send}^\sharp : \mathsf{IEx} \rightharpoonup \mathsf{IEx} \times \mathsf{IEx}; \qquad \mathsf{receive}^\sharp : \mathsf{IEx} \times \mathsf{IEx} \rightharpoonup \mathsf{IEx}$$

INIT: $\mathcal{D}_\tau.\vec{\sigma}_0(r)\ [\mathcal{R}_r]\ I_\emptyset$, where $I_\emptyset.E = \emptyset$

DO: $(\mathsf{do}^\sharp(I, e, r, o, a, t) = I' \ \wedge\ \mathcal{D}_\tau.\mathsf{do}(o, \sigma, t) = (\sigma', a) \ \wedge\ \sigma\ [\mathcal{R}_r]\ I)$
$\qquad \implies (\sigma'\ [\mathcal{R}_r]\ I' \ \wedge\ a = \mathcal{F}_\tau(\mathsf{ctxt}(I'.A, e)))$

SEND: $(\mathsf{send}^\sharp(I) = (I', J) \ \wedge\ \mathcal{D}_\tau.\mathsf{send}(\sigma) = (\sigma', m) \ \wedge\ \sigma\ [\mathcal{R}_r]\ I)$
$\qquad \implies (\sigma'\ [\mathcal{R}_r]\ I' \ \wedge\ m\ [\mathcal{M}]\ J)$

RECEIVE: $(\mathsf{receive}^\sharp(I, J)\!\downarrow \ \wedge\ \sigma\ [\mathcal{R}_r]\ I \ \wedge\ m\ [\mathcal{M}]\ J)$
$\qquad \implies (\mathcal{D}_\tau.\mathsf{receive}(\sigma, m)\ [\mathcal{R}_r]\ \mathsf{receive}^\sharp(I, J))$

ecutions supplied as parameters to $\mathsf{do}^\sharp$ and $\mathsf{send}^\sharp$. We similarly assume them in Fig. 5 via the definedness of these abstract methods.

5.2 Instantiation for State-Based Implementations

Fig. 6 defines the domain AInfo and abstract methods appropriate for state-based implementations. In §5.4 we show that the existence of a simulation of Def. 8 with respect to these parameters implies $\mathcal{D}_\tau$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_\tau$ (Theorems 9 and 10). The $\mathsf{do}^\sharp$ method adds a fresh event e with the given attributes to I; its timestamp t is recorded in info. In the resulting execution I', the event e is the last one by its replica, observes all events in I and occupies the place in arbitration consistent with t. The $\mathsf{send}^\sharp$ method just returns I, which formalizes the intuition that, in state-based implementations, send returns a message capturing all the information about the object available at the replica. The $\mathsf{receive}^\sharp$ method takes the component-wise union $I \sqcup J$ of executions I related to the current state and J related to the message, applied recursively to the components of $I.A$ and $J.A$. We also assume that $I \sqcup J$ recomputes the arbitration relation in the resulting execution from the timestamps. This is the reason for recording them in info: we would not be able to construct $\mathsf{receive}^\sharp(I, J).\mathsf{ar}$ solely from $I.\mathsf{ar}$ and $J.\mathsf{ar}$.

The agreement property $\mathsf{agree}(I, J)$ guarantees that $I \sqcup J$ is well-formed (e.g., its visibility relation is acyclic) and that, for each replica, I describes a computation extending J or vice versa. The latter follows from the observation we made when explaining the state-based counter in §2: a message or a state in a state-based implementation reflects a prefix of the sequence of events performed at a given replica. The first conjunct of the execution invariant inv requires arbitration to be consistent with event timestamps; the second conjunct follows from the definition of $\mathcal{V}^{\mathsf{state}}$ (§4). When discharging the obligations in Fig. 5 with respect to the parameters in Fig. 6 for a particular implementation, we can rely on the agreement property and the execution invariant.

5.3 Examples

We illustrate the use of the instantiation from §5.2 on the state-based counter, LWW-register and OR-set. In [12, §A] we also give proofs of two multi-value register implementations.

Counter: $\mathcal{D}_{\mathtt{ctr}}$ in Fig. 2(b) and $\mathcal{F}_{\mathtt{ctr}}$ in (3). Discharging the obligations in Fig. 5 for the simulation (6) is easy. The key case is RECEIVE, where the first conjunct of agree in Fig. 6 ensures that $\min\{v(s), v'(s)\}$ increments by a replica s are taken into account both in $v(s)$ and in $v'(s)$:

$$(\langle r, v\rangle\ [\mathcal{R}_r]\ I) \wedge (v'\ [\mathcal{M}]\ J) \implies (\forall s.\ \min\{v(s), v'(s)\} = |\{e \in I.E \cap J.E \mid I.\mathsf{oper}(e) = \mathtt{inc} \wedge I.\mathsf{repl}(e) = s\}|).$$

Figure 6. Instantiation for state-based data type implementations. In $\mathsf{do}^\sharp$ we omit the straightforward definition of ar' in terms of $I.\mathsf{info}$ and t.

$$\mathsf{AInfo} \ = \ \mathsf{Timestamp}$$

$\mathsf{agree}(I, J) \iff \forall r.\ ((\{e \in I.E \mid I.\mathsf{repl}(e) = r\}, I.\mathsf{ro})$ is a prefix of
$\qquad (\{e \in J.E \mid J.\mathsf{repl}(e) = r\}, J.\mathsf{ro})$ or vice versa$) \wedge (I \sqcup J \in \mathsf{IEx})$

$\mathsf{inv}(I) \iff (\forall e, f \in I.E.\ (e, f) \in I.\mathsf{ar} \iff I.\mathsf{info}(e) < I.\mathsf{info}(f))$
$\qquad \wedge ((I.\mathsf{vis} \cup I.\mathsf{ro})^+ \subseteq I.\mathsf{vis})$

$\mathsf{do}^\sharp(I, e, r, o, a, t) \ = \ I', \qquad$ if $\mathsf{inv}(I) \wedge e \notin I.E \wedge I' \in \mathsf{IEx}$
where $I' = ((I.E \cup \{e\},\ I.\mathsf{repl}[e \mapsto r],\ I.\mathsf{obj}[e \mapsto x],\ I.\mathsf{oper}[e \mapsto o],$
$\qquad I.\mathsf{rval}[e \mapsto a],\ I.\mathsf{ro} \cup \{(f, e) \mid f \in I.E \wedge I.\mathsf{repl}(f) = r\},$
$\qquad I.\mathsf{vis} \cup \{(f, e) \mid f \in I.E\}, \mathsf{ar}'),\ I.\mathsf{info}[e \mapsto t])$

$\mathsf{send}^\sharp(I) \qquad = (I, I), \qquad$ if $\mathsf{inv}(I)$

$\mathsf{receive}^\sharp(I, J) \quad = I \sqcup J, \qquad$ if $\mathsf{inv}(I) \wedge \mathsf{inv}(J) \wedge \mathsf{agree}(I, J)$

This allows establishing $\mathsf{receive}(\langle r, v\rangle, v')\ \ [\mathcal{R}_r]\ \ (I \sqcup J)$, thus formalizing the informal justification of correctness we gave in §2.

LWW-register: $\mathcal{D}_{\mathtt{intreg}}$ in Fig. 2(c) and $\mathcal{F}_{\mathtt{intreg}}$ in (4). We associate a state or a message $\langle a, t\rangle$ with any execution that contains a $\mathtt{wr}(a)$ event with the timestamp t maximal among all other $\mathtt{wr}$ events (as per info). By inv in Fig. 6, this event is maximal in arbitration, which implies that $\mathtt{rd}$ returns the correct value; the other obligations are also discharged easily. Formally, $\forall r.\ \mathcal{R}_r = \mathcal{M}$ and

$$\begin{aligned}&\langle a, t\rangle\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff \\ &\quad (t = 0 \wedge a = 0 \wedge (\neg \exists e \in E.\ \mathsf{oper}(e) = \mathtt{wr}(\_))) \vee \\ &\quad (t > 0 \wedge (\exists e \in E.\ \mathsf{oper}(e) = \mathtt{wr}(a) \wedge \mathsf{info}(e) = t) \\ &\qquad \wedge (\forall f \in E.\ \mathsf{oper}(f) = \mathtt{wr}(\_) \implies \mathsf{info}(f) \le t)).\end{aligned}$$

Optimized OR-set: $\mathcal{D}_{\mathtt{orset}}$ in Fig. 7 and $\mathcal{F}_{\mathtt{orset}}$ in (5). A problem with implementing a replicated set is that we often cannot discard the information about an element from a replica state after it has been removed: if another replica unaware of the removal sends us a snapshot of its state containing this element, the semantics of the set may require our receive to keep the element out of the set. As we prove in §6, for the OR-set keeping track of information about removed elements cannot be fully avoided, which makes its space-efficient implementation very challenging. Here we consider a recently-proposed OR-set implementation [6] that, as we show in §6, has an optimal space complexity. It improves on the original implementation [32], whose complexity was suboptimal (we have proved the correctness of the latter as well; see [12, §A]).

An additional challenge posed by the OR-set is that, according to $\mathcal{F}_{\mathtt{orset}}$, a $\mathtt{remove}$ operation may behave differently with respect to different events adding the same element to the set, depending on whether it sees them or not. This causes the implementation to treat internally each add operation as generating a unique *instance* of the *element* being added, further increasing the space required. To combat this, the implementation concisely summarizes information about instances. An instance is represented by a unique *instance identifier* that is generated when a replica performs an add and consists of the replica identifier and the number of adds (of any elements) performed at the replica until then. In a state $\langle r, V, w\rangle$, the vector w determines the identifiers of all instances that the current replica r has ever observed: for any replica s, the replica r has seen $w(s)$ successive identifiers $(s, 1), (s, 2), \ldots, (s, w(s))$ generated at s. To generate a new identifier in $\mathsf{do}(\mathsf{add}(a'))$, the replica r increments $w(r)$. The connection between the vector w in a state or a message and add events $e_{s,k}$ in corresponding executions is formalized in lines 1-3 of the simulation relation, also shown in Fig. 7. In receive we take the pointwise maximum of the two vectors w and w'. Like for the counter, the first conjunct of agree implies that this preserves the clauses in lines 1-3.

Figure 7. Optimized OR-set implementation [6] and its simulation

$$
\begin{aligned}
&\Sigma = \mathsf{ReplicaID} \times ((\mathbb{Z} \times \mathsf{ReplicaID}) \to \mathbb{N}_0) \times (\mathsf{ReplicaID} \to \mathbb{N}_0)\\
&\vec{\sigma}_0 = \lambda r.\, \langle r, (\lambda a, s.\, 0), (\lambda s.\, 0)\rangle\\
&M = ((\mathbb{Z} \times \mathsf{ReplicaID}) \to \mathbb{N}_0) \times (\mathsf{ReplicaID} \to \mathbb{N}_0)\\
&\mathsf{do}(\mathtt{add}(a'), \langle r, V, w\rangle, t) \;=\; (\langle r, (\lambda a, s.\, \text{if } a = a' \wedge s = r\\
&\qquad\qquad \text{then } w(r) + 1 \text{ else } V(a, s)), w[r \mapsto w(r) + 1]\rangle, \bot)\\
&\mathsf{do}(\mathtt{remove}(a'), \langle r, V, w\rangle, t) \;=\\
&\qquad\qquad (\langle r, (\lambda a, s.\, \text{if } a = a' \text{ then } 0 \text{ else } V(a, s)), w\rangle, \bot)\\
&\mathsf{do}(\mathtt{rd}, \langle r, V, w\rangle, t) \;=\; (\langle r, V, w\rangle, \{a \mid \exists s.\, V(a, s) > 0\})\\
&\mathsf{send}(\langle r, V, w\rangle) \;=\; (\langle r, V, w\rangle, \langle V, w\rangle)\\
&\mathsf{receive}(\langle r, V, w\rangle, \langle V', w'\rangle) \;=\\
&\qquad\qquad \langle r, (\lambda a, s.\, \text{if } (V(a, s) = 0 \wedge w(s) \geq V'(a, s)) \vee\\
&\qquad\qquad\qquad (V'(a, s) = 0 \wedge w'(s) \geq V(a, s))\\
&\qquad\qquad\qquad \text{then } 0 \text{ else } \max\{V(a, s), V'(a, s)\}),\\
&\qquad\qquad (\lambda s.\, \max\{w(s), w'(s)\})\rangle
\end{aligned}
$$

$$\langle s, V, w\rangle\ [\mathcal{R}_r]\ I \iff (r = s)\ \wedge\ (\langle V, w\rangle\ [\mathcal{M}]\ I)$$

$$
\begin{aligned}
&\langle V, w\rangle\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff\\
1:\;\; &\exists \text{ distinct } e_{s,k}.\, (\{e_{s,k} \mid s \in \mathsf{ReplicaID} \wedge 1 \leq k \leq w(s)\} =\\
2:\;\; &\qquad \{e \in E \mid \mathsf{oper}(e) = \mathtt{add}(\_)\}) \wedge\\
3:\;\; &(\forall s, k, j.\, (\mathsf{repl}(e_{s,k}) = s)\ \wedge\ (e_{s,j} \xrightarrow{\mathsf{ro}} e_{s,k} \iff j < k))\ \wedge\\
4:\;\; &(\forall a, s.\, (V(a, s) \leq w(s))\ \wedge\ (V(a, s) \neq 0 \implies\\
5:\;\; &\qquad (\mathsf{oper}(e_{s,V(a,s)}) = \mathtt{add}(a)) \wedge\\
6:\;\; &\qquad (\neg\exists k.\, V(a, s) < k \leq w(s) \wedge \mathsf{oper}(e_{s,k}) = \mathtt{add}(a)) \wedge\\
7:\;\; &\qquad (\neg\exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e_{s,V(a,s)} \xrightarrow{\mathsf{vis}} f))) \wedge\\
8:\;\; &(\forall a, s, k.\, e_{s,k} \in E \wedge \mathsf{oper}(e_{s,k}) = \mathtt{add}(a) \implies\\
9:\;\; &\qquad (k \leq V(a, s) \vee \exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e_{s,k} \xrightarrow{\mathsf{vis}} f))
\end{aligned}
$$

The component w in $\langle r, V, w\rangle$ records identifiers of both of those instances that have been removed and those that are still in the set (are *active*). The component V serves to distinguish the latter. As it happens, we do not need to store all active instances of an element a: for every replica s, it is enough to keep the last active instance identifier generated by an $\mathtt{add}(a)$ at this replica. If $V(a, s) \neq 0$, this identifier is $(s, V(a, s))$; if $V(a, s) = 0$, all instances of a generated at s that the current replica knows about are inactive. The meaning of V is formalized in the simulation: each instance identifier given by V is covered by w (line 4) and, if $V(a, s) \neq 0$, then the event $e_{s,V(a,s)}$ performs $\mathtt{add}(a)$ (line 5), is the last $\mathtt{add}(a)$ by replica s (line 6) and has not been observed by a $\mathtt{remove}(a)$ (line 7). Finally, the $\mathtt{add}(a)$ events that are not seen by a $\mathtt{remove}(a)$ in the execution are either the events $e_{s,V(a,s)}$ or those superseded by them (lines 8-9). This ensures that returning all elements with an active instance in $\mathtt{rd}$ matches $\mathcal{F}_{\mathtt{orset}}$.

When a replica r performs $\mathsf{do}(\mathtt{add}(a'))$, we update $V(a', r)$ to correspond to the new instance identifier. Conversely, in $\mathsf{do}(\mathtt{remove}(a'))$, we clear all entries in $V(a')$, thereby deactivating all instances of a'. However, after this their identifiers are still recorded in w, and so we know that they have been previously removed. This allows us to address the problem with implementing receive we mentioned above: if we receive a message with an active instance $(s, V'(a, s))$ of an element a that is not in the set at our replica ($V(a, s) = 0$), but previously existed ($w(s) \geq V'(a, s)$), this means that the instance has been removed and should not be active in the resulting state (the entry for (a, s) should be 0). We also do the same check with the state and the message swapped.

As the above explanation shows, our simulation relations are useful not only for proving correctness of data type implementations, but also for explaining their designs. Discharging obligations in Fig. 5 requires some work for the OR-set; due to space constraints, we defer this to [12, §A].

Figure 8. Function step that mirrors the effect of an event $e \in C'.E$ from $C' \in \mathsf{CEx}[x]$ in $D \in \mathsf{DEx}$, defined when so is the abstract method used

$$
\begin{aligned}
&\mathsf{step}(C', e, D) =\\
&\qquad D[r \mapsto \mathsf{do}^\sharp(D(r), e, r, C'.\mathsf{oper}(e), C'.\mathsf{rval}(e), C'.\mathsf{time}(e))],\\
&\quad \text{if } C'.\mathsf{act}(e) = \mathsf{do} \wedge C'.\mathsf{repl}(e) = r\\
&\mathsf{step}(C', e, D) = D[r \mapsto I, C'.\mathsf{msg}(e) \mapsto J],\\
&\quad \text{if } C'.\mathsf{act}(e) = \mathsf{send} \wedge C'.\mathsf{repl}(e) = r \wedge C'.\mathsf{msg}(e) \notin \mathsf{dom}(D) \wedge\\
&\qquad \mathsf{send}^\sharp(D(r)) = (I, J)\\
&\mathsf{step}(C', e, D) = D[r \mapsto \mathsf{receive}^\sharp(D(r), D(C'.\mathsf{msg}(e)))],\\
&\quad \text{if } C'.\mathsf{act}(e) = \mathsf{receive} \wedge C'.\mathsf{repl}(e) = r
\end{aligned}
$$

5.4 Soundness and Establishing Agreement Properties

We present conditions on AInfo and abstract methods ensuring the soundness of replication-aware simulations over them and, in particular, establishing the agreement property and execution invariants assumed via the definedness of abstract operations in Fig. 5.

THEOREM 9 (Soundness). *Assume* AInfo*,* $\mathsf{do}^\sharp$*,* $\mathsf{send}^\sharp$*,* $\mathsf{receive}^\sharp$*,* $\mathcal{V}$ *and* $\mathcal{T}$ *that satisfy the conditions in Fig. 9 for some* $\mathcal{G}$*. If there exists a replication-aware simulation between* $\mathcal{D}_\tau$ *and* $\mathcal{F}_\tau$ *with respect to these parameters, then* $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$*.*

Conditions in Fig. 9 require global reasoning, but can be discharged once for a class of data types. For example, they hold of the instantiation for state-based implementations from §5.2, as well as one for op-based implementations presented in [12, §B].

THEOREM 10. *There exists* $\mathcal{G}$ *such that, for all* $\mathcal{D}_\tau$*, the parameters in Fig. 6 satisfy the conditions in Fig. 9 with respect to this* $\mathcal{G}$*,* $\mathcal{V} = \mathcal{V}^{\mathsf{state}}$*,* $\mathcal{T} = \mathsf{T\text{-}Any}$*.*

The proofs of Theorems 9 and 10 are given in [12, §B]. To explain the conditions in Fig. 9, here we consider the proof strategy for Theorem 9. To establish $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$, for any $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ we need to show $\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau]$. We prove this by induction on the length of C. To use the localized conditions in Fig. 5, we require a relation $\mathcal{G}$ associating C with a ***decomposed execution***—a partial function $D : (\mathsf{ReplicaID} \cup \mathsf{MessageID}) \rightharpoonup \mathsf{IEx}$ that gives fragments of $\mathsf{abs}(C, \mathcal{V})$ corresponding to replica states and messages in the final configuration of C. We write DEx for the set of all decomposed executions, so that $\mathcal{G} \subseteq \mathsf{CEx}[x] \times \mathsf{DEx}$. The existence of a decomposed execution D such that $C\ [\mathcal{G}]\ D$ forms the core of our induction hypothesis. G-CTXT in Fig. 9 checks that the abstract methods construct visibility according to $\mathcal{V}$: it requires the context of any event e by a replica r to be the same in $D(r)$ and $\mathsf{abs}(C, \mathcal{V})$. Together with DO in Fig. 5, this ensures $\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau]$.

We write $C' \sim (C \xrightarrow{e} (R, T))$ when C' is an extension of C in the following sense: $C'.E = C.E \uplus \{e\}$, the other components of C are those of C' restricted to $C.E$, e is last in $C'.\mathsf{eo}$ and $C'.\mathsf{post}(e) = (R, T)$. For the induction step, assume $C\ [\mathcal{G}]\ D$ and $C' \sim (C \xrightarrow{e} (R, T))$; see Fig. 4(c). Then the decomposed execution D' corresponding to C' is given by $\mathsf{step}(C', e, D)$, where the function step in Fig. 8 mirrors the effect of the event e from C' in D using the abstract methods. G-STEP ensures that it preserves the relation $\mathcal{G}$. Crucially, G-STEP also requires us to establish the definedness of step and thus the corresponding abstract method. This justifies the agreement property and execution invariants encoded by the definedness and allows us to use the conditions in Fig. 5 to complete the induction. We also require G-INIT, which establishes the base case, and G-VIS, which formulates a technical restriction on $\mathcal{V}$. Finally, the conditions in Fig. 9 allow us to use the transport specification $\mathcal{T}$ by considering only executions C satisfying it.

6. Space Bounds and Implementation Optimality

Object states in replicated data type implementations include not only the current client-observable content, but also metadata

Figure 9. Proof obligations for abstract methods. Free variables are implicitly universally quantified and have the following types: $C, C' \in \mathsf{CEx}[x] \cap \mathcal{T}$, $D \in \mathsf{DEx}$, $r \in \mathsf{ReplicaID}$, $e \in \mathsf{Event}$, $(R, T) \in \mathsf{Config}$.

G-CTXT: $(C\ [\mathcal{G}]\ D\ \wedge\ e \in \mathsf{abs}(C, \mathcal{V}).E\ \wedge\ \mathsf{abs}(C, \mathcal{V}).\mathsf{repl}(e) = r)$
$\quad\implies\ \mathsf{ctxt}(D(r).A, e) = \mathsf{ctxt}(\mathsf{abs}(C, \mathcal{V}), e)$

G-STEP: $(C' \sim (C \xrightarrow{e} (R, T))\ \wedge\ (C\ [\mathcal{G}]\ D))$
$\quad\implies\ (\mathsf{step}(C', e, D)\!\downarrow\ \wedge\ C'\ [\mathcal{G}]\ \mathsf{step}(C', e, D))$

G-INIT: $(C.E = \{e\}\ \wedge\ C.\mathsf{pre}(e) = (\_, [\,]))$
$\quad\implies\ (\mathsf{step}(C, e, D_\emptyset)\!\downarrow\ \wedge\ C\ [\mathcal{G}]\ \mathsf{step}(C, e, D_\emptyset))$,
where $D_\emptyset$ is such that $\mathsf{dom}(D_\emptyset) = \mathsf{ReplicaID}\ \wedge$
$\forall r \in \mathsf{ReplicaID}.\ D_\emptyset(r).E = \emptyset$

G-VIS: $(e \in \mathsf{abs}(C, \mathcal{V}).E\ \wedge\ (C$ is a prefix of C' under $C'.\mathsf{eo}))$
$\quad\implies\ \mathsf{ctxt}(\mathsf{abs}(C, \mathcal{V}), e) = \mathsf{ctxt}(\mathsf{abs}(C', \mathcal{V}), e)$

needed for conflict resolution or masking network failures. Space taken by this metadata is a major factor determining their efficiency and feasibility. As illustrated by the OR-set in §5.3, this is especially so for state-based implementations, i.e., those that satisfy their data type specifications with respect to the visibility witness $\mathcal{V}^{\mathsf{state}}$ and the transport layer specification $\mathsf{T\text{-}Any}$. We now present a general technique for proving lower bounds on this space overhead, which we use to prove optimality of four state-based implementations (we leave other implementation classes for future work; see §9). As in §5, we only consider executions over a fixed object x of type τ.

6.1 Metadata Overhead

To measure space, we need to consider how data are represented. An ***encoding*** of a set S is an injective function $\mathsf{enc} : S \to \Lambda^+$, where Λ is some suitably chosen fixed finite set of characters (left unspecified). Sometimes, we clarify the domain being encoded using a subscript: e.g., $\mathsf{enc}_{\mathbb{N}_0}(1)$. For $s \in S$, we let $\mathsf{len}_S(s)$ be the length of $\mathsf{enc}_S(s)$. The length can vary: e.g., for an integer k, $\mathsf{len}_{\mathbb{N}_0}(k) \in \Theta(\lg k)$. We use standard encodings (listed in [12, §C]) for return values $\mathsf{enc}_{\mathsf{Val}_\tau}$ of the data types τ we consider and assume an arbitrary but fixed encoding of object states $\mathsf{enc}_{\mathcal{D}_\tau.\Sigma}$.

To distinguish metadata from the client-observable content of the object, we assume that each data type has a special `rd` operation that returns the latter, as is the case in the examples considered so far. For a concrete execution $C \in [\![\mathcal{D}_\tau]\!]$ over the object x and a read event $e \in (C.E)|_{\mathtt{rd}}$, we define $\mathsf{state}(e)$ to be the state of the object accessed at e: $\mathsf{state}(e) = R(x, C.\mathsf{repl}(e))$ for $(R, \_) = C.\mathsf{pre}(e)$.

We now define the metadata overhead as a ratio, by dividing the size of the object state by the size of the observable state. We then quantify the worst-case overhead by taking the maximum of this ratio over all read operations in all executions with given numbers of replicas n and update operations m. To define the latter, we assume that each data type τ specifies a set $\mathsf{Upd}_\tau \subset \mathsf{Op}_\tau$ of update operations; for all examples in this paper $\mathsf{Upd}_\tau = \mathsf{Op}_\tau \setminus \{\mathtt{rd}\}$.

DEFINITION 11. *The* ***maximum metadata overhead*** *of an execution $C \in [\![\mathcal{D}_\tau]\!]$ of an implementation $\mathcal{D}_\tau$ is*

$$\mathsf{mmo}(\mathcal{D}_\tau, C) = \max\left\{ \frac{\mathsf{len}_{\mathcal{D}_\tau.\Sigma}(\mathsf{state}(e))}{\mathsf{len}_{\mathsf{Val}_\tau}(C.\mathsf{rval}(e))} \mid e \in (C.E)|_{\mathtt{rd}} \right\}.$$

The ***worst-case metadata overhead*** *of an implementation $\mathcal{D}_\tau$ over all executions with n replicas and m updates ($2 \le n \le m$) is*

$$\begin{aligned}\mathsf{wcmo}(\mathcal{D}_\tau, n, m) = \max\{&\mathsf{mmo}(\mathcal{D}_\tau, C) \mid C \in [\![\mathcal{D}_\tau]\!] \wedge \\ &n = |\{C.\mathsf{repl}(e) \mid e \in C.E\}| \wedge \\ &m = |\{e \in C.E \mid C.\mathsf{oper}(e) \in \mathsf{Upd}_\tau\}|\}.\end{aligned}$$

We consider only executions with $m \ge n$, since we are interested in the asymptotic overhead of executions where all replicas are mutated (i.e., perform at least one update operation).

Figure 10. Summary of bounds on metadata overhead for stated-based implementations, as functions of the number of replicas n and updates m

Type	Existing implementation			Any implementation
	algorithm	ref.	overhead	overhead
`ctr`	Fig. 2(b)	[32]	$\widehat{\Theta}(n)$	$\widehat{\Omega}(n)$
`orset`	Fig. 7	[6]	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
	Fig. 15, [12, §A]	[32]	$\widehat{\Theta}(m \lg m)$	
`intreg`	Fig. 2(c)	[32]	$\widehat{\Theta}(\lg m)^\dagger$	$\widehat{\Omega}(\lg m)$
`mvr`	Fig. 17, [12, §A]	new‡	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
	Fig. 16, [12, §A]	[32]	$\widehat{\Theta}(n^2 \lg m)$	

† Assuming timestamp encoding is $O(\lg m)$, satisfied by Lamport clocks.
‡ An optimization of [32] discovered during the optimality proof.

DEFINITION 12. *Assume $\mathcal{D}_\tau$ and a positive function $f(n, m)$.*

- *f is an* ***asymptotic upper bound*** *($\mathcal{D}_\tau \in \widehat{O}(f(n, m))$) if $\sup_{n,m\to\infty}(\mathsf{wcmo}(\mathcal{D}_\tau, n, m)/f(n, m)) < \infty$, i.e.,*

$$\exists K > 0.\, \forall m \ge n \ge 2.\, \mathsf{wcmo}(\mathcal{D}_\tau, n, m) < K f(n, m);$$

- *f is an* ***asymptotic lower bound*** *($\mathcal{D}_\tau \in \widehat{\Omega}(f(n, m))$) if $\lim_{n,m\to\infty}(\mathsf{wcmo}(\mathcal{D}_\tau, n, m)/f(n, m)) \ne 0$, i.e.,*

$$\exists K > 0.\, \forall m_0 \ge n_0 \ge 2.\, \exists n \ge n_0, m \ge n_0.\ \mathsf{wcmo}(\mathcal{D}_\tau, n, m) > K f(n, m);$$

- *f is an* ***asymptotically tight bound*** *($\mathcal{D}_\tau \in \widehat{\Theta}(f(n, m))$) if it is both an upper and a lower asymptotic bound.*

Fig. 10 summarizes our results; as described in §5, we have proved all the implementations correct. Matching lower and upper bounds indicate worst-case optimality of an implementation (note that this is different from optimality in all cases). The derivation of upper bounds relies on standard techniques and is deferred to [12, §C]. We now proceed to the main challenge: how to derive lower bounds that apply to *any* implementation of τ. We present proofs for `ctr` and `orset`; `intreg` and `mvr` are covered in [12, §C].

6.2 Experiment Families

The goal is to show that for any correct implementation $\mathcal{D}_\tau$ (i.e., such that $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_\tau$), the object state must store some minimum amount of information. We achieve this by constructing an *experiment family*, which is a collection of executions $\mathbb{C}_\alpha$, where $\alpha \in Q$ for some index set Q. Each experiment contains a distinguished read event $\mathbf{e}_\alpha$. The experiments are designed in such a way that the object states $\mathsf{state}(\mathbf{e}_\alpha)$ must be distinct, which then implies a lower bound $\lg_{|\Lambda|} |Q|$ on the size of their encoding. To prove that they are distinct, we construct black-box tests that execute the methods of $\mathcal{D}_\tau$ on the states and show that the tests must produce different results for each $\mathsf{state}(\mathbf{e}_\alpha)$ provided $\mathcal{D}_\tau$ is correct. Formally, the tests induce a read-back function rb that satisfies $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. We encapsulate the core argument in the following lemma.

DEFINITION 13. *An* ***experiment family*** *for an implementation $\mathcal{D}_\tau$ is a tuple $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ where Q is a finite set, $2 \le n \le m$, and for each $\alpha \in Q$, $\mathbb{C}_\alpha \in [\![\mathcal{D}_\tau]\!]$ is an execution with n replicas and m updates, $\mathbf{e}_\alpha \in (\mathbb{C}_\alpha.E)|_{\mathtt{rd}}$ and $\mathsf{rb} : \mathcal{D}_\tau.\Sigma \to Q$ is a function satisfying $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$.*

LEMMA 14. *If $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ is an experiment family, then*

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \ge \lfloor \lg_{|\Lambda|} |Q| \rfloor \,/\, (\max_{\alpha\in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))).$$

PROOF. Since $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$, the states $\mathsf{state}(\mathbf{e}_\alpha)$ are pairwise distinct and so are their encodings $\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))$. Since there are fewer than $|Q|$ strings of length strictly less than $\lfloor \lg_{|\Lambda|} |Q| \rfloor$, for some $\alpha \in Q$ we have $\mathsf{len}(\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor$. Then

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \geq \mathsf{mmo}(\mathcal{D}_\tau, \mathbb{C}_\alpha) \geq \frac{\mathsf{len}(\mathsf{state}(\mathbf{e}_\alpha))}{\mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))} \geq \frac{\lfloor \lg_{|\Lambda|} |Q| \rfloor}{\max_{\alpha' \in Q} \mathsf{len}(\mathbb{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))}. \quad \square$$

To apply this lemma to the best effect, we need to find experiment families with $|Q|$ as large as possible and $\mathsf{len}(\mathbb{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))$ as small as possible. Finding such families is challenging, as there is no systematic way to derive them. We relied on intuitions about "which situations force replicas to store a lot of information" when searching for experiment families.

Driver programs. We define experiment families using ***driver programs*** (e.g., see Fig. 11). These are written in imperative pseudocode and use traditional constructs like loops and conditionals. As they execute, they construct concrete executions of the data type library $[\tau \mapsto \mathcal{D}_\tau]$ by means of the following instructions, each of which triggers a uniquely-determined transition from Fig. 3:

$\mathsf{do}_r\ o\ ^t$	do operation o on x at replica r with timestamp t
$u \leftarrow \mathsf{do}_r\ o\ ^t$	same, but assign the return value to u
$\mathsf{send}_r(mid)$	send a message for x with identifier mid at r
$\mathsf{receive}_r(mid)$	receive the message mid at replica r

Programs explicitly supply timestamps for do and message identifiers for send and receive. We require that they do this correctly, e.g., respect uniqueness of timestamps. When a driver program terminates, it may produce a return value. For a program P, an implementation $\mathcal{D}_\tau$, and a configuration (R, T), we let $\mathsf{exec}(\mathcal{D}_\tau, (R, T), P)$ be the concrete execution of the data type library $[\tau \mapsto \mathcal{D}_\tau]$ starting in (R, T) that results from running P; we define $\mathsf{result}(\mathcal{D}_\tau, (R, T), P)$ as the return value of P in this run.

6.3 Lower Bound for State-Based Counter (ctr)

THEOREM 15. *If $\mathcal{D}_{\mathtt{ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{ctr}}$, then $\mathcal{D}_{\mathtt{ctr}}$ is $\widehat{\Omega}(n)$.*

We start by formulating a suitable experiment family.

LEMMA 16. *If $\mathcal{D}_{\mathtt{ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{ctr}}$, $n \geq 2$ and $m \geq n$ is a multiple of $(n-1)$, then tuple $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ as defined in the left column of Fig. 11 is an experiment family.*

The idea of the experiments is to force replica 1 to remember one number for each of the other replicas in the system, which then introduces an overhead proportional to n; cf. the implementation in Fig. 2(b). We show one experiment in Fig. 12. All experiments start with a common initialization phase, defined by $init$, where each of the replicas $2..n$ performs $m/(n-1)$ increments and sends a message after each increment. All messages remain undelivered until the second phase, defined by $exp(\alpha)$. There replica 1 receives exactly one message from each replica $r = 2..n$, selected using $\alpha(r)$. An experiment concludes with the read $\mathbf{e}_\alpha$ on the first replica.

The read-back works by performing separate tests for each of the replicas $r = 2..n$, defined by $test(r)$. For example, to determine which message was sent by replica 2 during the experiment in Fig. 12, the program $test(2)$: reads the counter value at replica 1, getting 12; delivers the final message by replica 2 to it; and reads the counter value at replica 1 again, getting 14. By observing the difference, the program can determine the message sent during the experiment: $\alpha(2) = 5 - (14 - 12) = 3$.

PROOF OF LEMMA 16. The only nontrivial obligation is to prove $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, T_\alpha) = \mathsf{final}(\mathbb{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{\text{(i)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_0, T_0), (init; exp(\alpha); test(r))) \\
&= \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_\alpha, T_\alpha), test(r)) \\
&\overset{\text{(ii)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_{init}[(x,1) \mapsto R_\alpha(x,1)], T_{init}), test(r)) \\
&= \mathsf{rb}(R_\alpha(x,1))(r) \ = \ \mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

Figure 11. Experiment families $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ used in the proofs of Theorem 15 (ctr) and Theorem 17 (orset)

ctr	orset
Conditions on n, m (number of replicas/updates)	
$m \geq n \geq 2$	$m \geq n \geq 2$
$m \mod (n-1) = 0$	$(m-1) \mod (n-1) = 0$
Index set Q	
$Q = ([2..n] \rightarrow [1..\frac{m}{n-1}])$	$Q = ([2..n] \rightarrow [1..\frac{m-1}{n-1}])$
Family size $\lvert Q\rvert$	
$\lvert Q\rvert = (\frac{m}{n-1})^{n-1}$	$\lvert Q\rvert = (\frac{m-1}{n-1})^{n-1}$
Driver programs	
see below	see below
Definition of executions $\mathbb{C}_\alpha$	
$\mathbb{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, T_0), init; exp(\alpha))$ where $(R_0, T_0) = ([x \mapsto \mathcal{D}_\tau.\vec{\sigma}_0], \emptyset)$	
Definition of read-back function $\mathsf{rb} : \mathcal{D}_\tau.\Sigma \rightarrow Q$	
$\mathsf{rb}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[(x,1) \mapsto \sigma], T_{init}), test(r))$ where $(R_{init}, T_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, T_0), init))$	

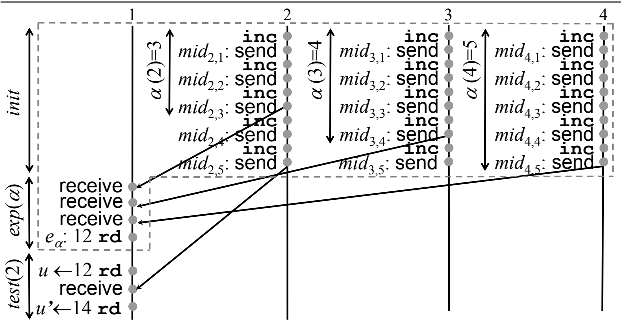
Driver programs (ctr):

```
procedure $init$
  for all $r \in [2..n]$
    for all $i \in [1..\frac{m}{n-1}]$
      $\mathsf{do}_r$ inc $^{rm+i}$
      $\mathsf{send}_r(mid_{r,i})$
procedure $exp(\alpha)$
  for all $r \in [2..n]$
    $\mathsf{receive}_1(mid_{r,\alpha(r)})$
  $\mathsf{do}_1$ rd $^{(n+2)m}$    // read $\mathbf{e}_\alpha$
procedure $test(r)$
  $u \leftarrow \mathsf{do}_1$ rd $^{(n+3)m}$
  $\mathsf{receive}_1(mid_{r,\frac{m}{n-1}})$
  $u' \leftarrow \mathsf{do}_1$ rd $^{(n+4)m}$
  return $\frac{m}{n-1} - (u' - u)$
```

Driver programs (orset):

```
procedure $init$
  for all $r \in [2..n]$
    for all $i \in [1..\frac{m-1}{n-1}]$
      $\mathsf{do}_r$ add(0) $^{rm+i}$
      $\mathsf{send}_r(mid_{r,i})$
procedure $exp(\alpha)$
  for all $r \in [2..n]$
    $\mathsf{receive}_1(mid_{r,\alpha(r)})$
  $\mathsf{do}_1$ remove(0) $^{(n+2)m}$
  $\mathsf{do}_1$ rd $^{(n+3)m}$    // read $\mathbf{e}_\alpha$
procedure $test(r)$
  for all $i \in [1..(\frac{m-1}{n-1})]$
    $\mathsf{receive}_1(mid_{r,i})$
    $u \leftarrow \mathsf{do}_1$ rd $^{(n+4)m+i}$
    if $0 \in u$
      return $i - 1$
  return $\frac{m-1}{n-1}$
```

Figure 12. Example experiment ($n = 4$ and $m = 15$) and test for ctr. Gray dashed lines represent the configuration $(R_{init}[(x,1) \mapsto R_\alpha(x,1)], T_{init})$ where the $test$ driver program is applied.



where:

(i) This is due to $\mathcal{D}_{\mathtt{ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{ctr}}$, as we explained informally above. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\mathtt{ctr}}, (R_0, T_0), (init; exp(\alpha); test(r))).$$

Then the operation context in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ of the first read in $test(r)$ contains $\sum_{r=2}^{n} \alpha(r)$ increments, while that of the second read contains $(m/(n-1)) - \alpha(r)$ more increments.

(ii) We have $T_\alpha = T_{init}$ because $exp(\alpha)$ does not send any messages. Also, R_α and $R_{init}[(x,1) \mapsto R_\alpha(x,1)]$ can differ only in the states of the replicas $2..n$. These cannot influence the run of $test(r)$, since it performs events on replica 1 only. $\square$

PROOF OF THEOREM 15. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that m is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ given by Lemma 16. Then for any α, $\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)$ is at most the total number of increments m in $\mathbb{C}_\alpha$. Using Lemma 14 and $m \geq n^2$, for some constants K_1, K_2, K_3, K independent from n_0, m_0 we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{ctr}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq$$

$$K_1 \frac{\lg_{|\Lambda|}(\frac{m}{n-1})^{n-1}}{\mathsf{len}_{\mathbb{N}_0}(m)} \geq K_2 \frac{n \lg(m/n)}{\lg m} \geq K_3 \frac{n \lg \sqrt{m}}{\lg m} \geq Kn. \quad \square$$

6.4 Lower Bound for State-Based OR-Set (`orset`)

THEOREM 17. *If $\mathcal{D}_{\mathtt{orset}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{orset}}$, then $\mathcal{D}_{\mathtt{orset}}$ is $\widehat{\Omega}(n \lg m)$.*

LEMMA 18. *If $\mathcal{D}_{\mathtt{orset}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{orset}}$, $n \geq 2$ and $m \geq n$ is such that $(m-1)$ is a multiple of $(n-1)$, then the tuple $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ on the right in Fig. 11 is an experiment family.*

The proof is the same as that of Lemma 16, except for obligation (i). We therefore give only informal explanations.

The main idea of the experiments defined in the lemma is to force replica 1 to remember element instances even after they have been removed at that replica; cf. our explanation of the challenges of implementing the OR-set from §5.3. The experiments follow a similar pattern to those for `ctr`, but use different operations. In the common *init* phase, each replica $2..n$ performs $\frac{m-1}{n-1}$ operations adding a designated element 0, which are interleaved with sending messages. In the experiment phase $exp(\alpha)$, one message from each replica $r = 2..n$, selected by $\alpha(r)$, is delivered to replica 1. At the end of execution, replica 1 removes 0 from the set and performs the read $\mathbf{e}_\alpha$. The return value of this read is always the empty set.

To perform the read-back of $\alpha(r)$ for $r = 2..n$, $test(r)$ delivers all messages by replica r to replica 1 in the order they were sent and, after each such delivery, checks if replica 1 now reports the element 0 as part of the set. From $\mathcal{D}_{\mathtt{orset}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{orset}}$ and the definition (5) of $\mathcal{F}_{\mathtt{orset}}$, we get that exactly the first $\alpha(r)$ such deliveries will have no effect on the contents of the set: the respective add operations have already been observed by the remove operation that replica 1 performed in the experiment phase. Thus, if 0 appears in the set right after delivering the i-th message of replica r, then $\alpha(r) = i - 1$, and if 0 does not appear by the time the loop is finished, then $\alpha(r) = (m-1)/(n-1)$.

PROOF OF THEOREM 17. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that $(m-1)$ is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ given by Lemma 18. For any $\alpha \in Q$, $\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \emptyset$, which can be encoded with a constant length. Using Lemma 14 and $m \geq n^2$, for some constants K_1, K_2, K we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{orset}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor \, / \, (\max_{\alpha \in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)))$$
$$\geq K_1 n \lg(m/n) \geq K_2 n \lg \sqrt{m} \geq Kn \lg m. \quad \square$$

7. Store Correctness and Consistency Axioms

Recall that we define a replicated store by a data type library $\mathbb{D}$ and a transport layer specification $\mathcal{T}$ (§4), and we specify its behavior by a function $\mathbb{F}$ from types $\tau \in \mathsf{dom}(\mathbb{D})$ to data type specifications and a set of consistency axioms (§3). The axioms are just constraints over abstract executions, such as those shown in Fig. 13; from now on we denote their sets by X. So far we have concentrated on single data type specifications $\mathbb{F}(\tau)$ and their correspondence to implementations $\mathbb{D}(\tau)$, as stated by Def. 7. In this section we consider consistency axioms and formulate the notion of correctness of the whole store $(\mathbb{D}, \mathcal{T})$ with respect to its specification $(\mathbb{F}, \mathsf{X})$.

Our first goal is to lift the statement of correctness given by Def. 7 to a store $(\mathbb{D}, \mathcal{T})$ with multiple objects of different data types. To this end, we assume a function $\mathbb{V}$ mapping each type $\tau \in \mathsf{dom}(\mathbb{D})$ to its visibility witness $\mathbb{V}$. This allows us to construct the visibility relation for a concrete execution $C \in [\![\mathbb{D}]\!] \cap \mathcal{T}$ by applying $\mathbb{V}(\tau)$ to its projection onto the events on every object of type τ:

$$\mathsf{witness}(\mathbb{V}) = \lambda C. \bigcup \{\mathbb{V}(\mathsf{type}(x))(C|_x) \mid x \in \mathsf{Obj}\},$$

where $\cdot|_x$ projects to events over x. Then the correctness of every separate data type τ in the store with respect to $\mathbb{F}(\tau)$ according to Def. 7 automatically ensures that the behavior of the whole store is consistent with $\mathbb{F}$ in the sense of Def. 5.

PROPOSITION 19. $(\forall \tau \in \mathsf{dom}(\mathbb{D}).\ \mathbb{D}(\tau)\ \mathsf{sat}[\mathbb{V}(\tau), \mathcal{T}]\ \mathbb{F}(\tau)) \implies$
$(\forall C \in [\![\mathbb{D}]\!] \cap \mathcal{T}.\ \mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathbb{F})$.

This motivates the following definition of store correctness. Let us write $A \models \mathsf{X}$ when the abstract execution A satisfies the axioms X.

DEFINITION 20. *A store $(\mathbb{D}, \mathcal{T})$ is **correct** with respect to a specification $(\mathbb{F}, \mathsf{X})$, if for some $\mathbb{V}$:*
(i) $\forall \tau \in \mathsf{dom}(\mathbb{D})$. $(\mathbb{D}(\tau)\ \mathsf{sat}[\mathbb{V}(\tau), \mathcal{T}]\ \mathbb{F}(\tau))$; and
(ii) $\forall C \in [\![\mathbb{D}]\!] \cap \mathcal{T}$. $(\mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathsf{X})$.

We showed how to discharge (i) in §5. The validity of axioms X required by (ii) most often depends on the transport layer specification $\mathcal{T}$: e.g., to disallow the anomaly (2) from §1, $\mathcal{T}$ needs to provide guarantees on how messages pertaining to different objects are delivered. However, data type implementations can also enforce axioms by putting enough information into messages: e.g., implementations correct with respect to $\mathcal{V}^{\mathsf{state}}$ from §4 ensure that vis is transitive regardless of the behavior of the transport layer. Fortunately, to establish (ii) in practice, we do not need to consider the internals of data type implementations in $\mathbb{D}$—just knowing the visibility witnesses used in the statements of their correctness is enough, as formulated in the following definition.

DEFINITION 21. *A set W of visibility witnesses and a transport layer specification $\mathcal{T}$ **validate** axioms X, if*

$$\forall C, \mathbb{V}.\ (C \in \mathcal{T}) \wedge (\{\mathbb{V}(\tau) \mid \tau \in \mathsf{dom}(\mathbb{V})\} \subseteq W) \implies (\mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathsf{X}).$$

Since visibility witnesses are common to wide classes of data types (e.g., state- or op-based), our proofs of the validity of axioms will not have to be redone if we add new data type implementations to the store from a class already considered.

We next present axioms formalizing several variants of eventual consistency used in replicated stores (Fig. 13 and 14) and W and $\mathcal{T}$ that validate them. We then use this as a basis for discussing connections with weak shared-memory models. Due to space constraints, we defer technical details and proofs to [12, §D].

Basic eventual consistency. EVENTUAL and THINAIR define a weak form of eventual consistency. EVENTUAL ensures that an event cannot be invisible to infinitely many other events on the same object and thus implies (1) from §1: informally, if updates stop, then reads at all replicas will eventually see all updates and will return the same values (§3.2). However, EVENTUAL is stronger than quiescent consistency: the latter does not provide any guarantees at all for executions with infinitely many updates to the store, whereas our specification implies that the return values are computed according to $\mathbb{F}(\tau)$ using increasingly up-to-date view of the store state. We formalize these relationships in [12, §D].

THINAIR prohibits values from appearing "out-of-thin-air" [28], like 42 in Fig. 14(a) (recall that registers are initialized to 0). Cycles in $\mathsf{ro} \cup \mathsf{vis}$ that lead to out-of-thin-airs usually arise

Figure 13. A selection of consistency axioms over an execution $(E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$

Auxiliary relations

$\mathsf{sameobj}(e, f) \iff \mathsf{obj}(e) = \mathsf{obj}(f)$

Per-object causality (aka happens-before) order:
$\mathsf{hbo} = ((\mathsf{ro} \cap \mathsf{sameobj}) \cup \mathsf{vis})^+$

Causality (aka happens-before) order: $\mathsf{hb} = (\mathsf{ro} \cup \mathsf{vis})^+$

Axioms

EVENTUAL:
$\forall e \in E.\, \neg(\exists \text{ infinitely many } f \in E.\, \mathsf{sameobj}(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f))$

THINAIR: $\mathsf{ro} \cup \mathsf{vis}$ is acyclic

POCV (Per-Object Causal Visibility): $\mathsf{hbo} \subseteq \mathsf{vis}$

POCA (Per-Object Causal Arbitration): $\mathsf{hbo} \subseteq \mathsf{ar}$

COCV (Cross-Object Causal Visibility): $(\mathsf{hb} \cap \mathsf{sameobj}) \subseteq \mathsf{vis}$

COCA (Cross-Object Causal Arbitration): $\mathsf{hb} \cup \mathsf{ar}$ is acyclic

Figure 14. Anomalies allowed or disallowed by different axioms

(a) Disallowed by THINAIR:

$x, y : \mathtt{intreg}$

$i = x.\mathtt{rd}$ ‖ $j = y.\mathtt{rd}$
$y.\mathtt{wr}(i)$ ‖ $x.\mathtt{wr}(j)$

$$\begin{tikzcd}
x.\mathtt{rd}\colon 42 \arrow[d, "\mathsf{ro}"'] & y.\mathtt{rd}\colon 42 \arrow[d, "\mathsf{ro}"] \\
y.\mathtt{wr}(42) \arrow[ur, "\mathsf{vis}"] & x.\mathtt{wr}(42) \arrow[ul, "\mathsf{vis}"']
\end{tikzcd}$$

(b) Disallowed by POCV:

$x : \mathtt{orset}$

$x.\mathtt{add}(1)$ ‖ $i = x.\mathtt{rd}$ ‖ $j = x.\mathtt{rd}$
$x.\mathtt{add}(2)$ ‖ $x.\mathtt{add}(3)$ ‖

$$\begin{tikzcd}
x.\mathtt{add}(1) \arrow[d, "\mathsf{ro}"'] & x.\mathtt{rd}\colon \{2\} \arrow[d, "\mathsf{ro}"] & \\
x.\mathtt{add}(2) \arrow[ur, "\mathsf{vis}"] & x.\mathtt{add}(3) \arrow[r, "\mathsf{vis}"'] & x.\mathtt{rd}\colon \{3\}
\end{tikzcd}$$

(c) Allowed by COCV and COCA:

$x, y : \mathtt{intreg}$

$x.\mathtt{wr}(1)$ ‖ $y.\mathtt{wr}(1)$
$i = y.\mathtt{rd}$ ‖ $j = x.\mathtt{rd}$

$$\begin{tikzcd}
x.\mathtt{wr}(1) \arrow[d, "\mathsf{ro}"'] & y.\mathtt{wr}(1) \arrow[d, "\mathsf{ro}"] \\
y.\mathtt{rd}\colon 0 & x.\mathtt{rd}\colon 0
\end{tikzcd}$$

from effects of speculative computations, which are done by some older replicated stores [36].

THINAIR is validated by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and $\mathsf{T\text{-}Any}$, and EVENTUAL by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and the following condition on C ensuring that every message is eventually delivered to all other replicas and every operation is followed by a message generation:

$$(\forall e \in C.E.\, \forall r, r'.\, C.\mathsf{act}(e) = \mathsf{send} \wedge C.\mathsf{repl}(e) = r \wedge r \neq r' \implies \exists f.\, C.\mathsf{repl}(f) = r' \wedge e \xrightarrow{\mathsf{del}(C)} f) \wedge$$
$$(\forall e \in C.E.\, C.\mathsf{act}(e) = \mathsf{do} \implies \exists f.\, \mathsf{act}(f) = \mathsf{send} \wedge e \xrightarrow{\mathsf{roo}(C)} f),$$

where $\mathsf{roo}(C)$ is $\mathsf{ro}(C)$ projected to events on the same object.

Causality guarantees. Many replicated stores achieve availability and partition tolerance while providing stronger guarantees, which we formalize by the other axioms in Fig. 13. We call an execution ***per-object***, respectively, ***cross-object causally consistent***, if it is eventually consistent (as per above) and satisfies the axioms POCV and POCA, respectively, COCV and COCA. POCV guarantees that an operation sees all operations connected to it by a causal chain of events on the same object; COCV also considers causal chains via different objects. Thus, POCV disallows the execution in Fig. 14(b), and COCV the one in §3.1, corresponding to (2) from §1. POCA and COCA similarly require arbitration to be consistent with causality. The axioms highlight the principle of formalizing stronger consistency models: including more edges into vis and ar, so that clients have more up-to-date information.

Cross-object causal consistency is implemented by, e.g., COPS [27] and Gemini [23]. It is weaker than strong consistency, as it allows reading stale data. For example, it allows the execution in Fig. 14(c), where both reads fetch the initial value of the register, despite writes to it by the other replica. It is easy to check that this outcome cannot be produced by any interleaving of the events at the two replicas, and is thus not strongly consistent.

An interesting feature of per-object causal consistency is that state-based data types ensure most of it just by the definition of $\mathcal{V}^{\mathsf{state}}$: POCV is validated by $\{\mathcal{V}^{\mathsf{state}}\}$ and $\mathsf{T\text{-}Any}$. If the witness set is $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$, then we need $\mathcal{T}$ to guarantee the following: informally, if a send event e and another event f are connected by a causal chain of events on the same object, then the message created by e is delivered to $C.\mathsf{repl}(f)$ by the time f is done. POCA is validated by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and the transport layer specification $(\mathsf{roo}(C) \cup \mathsf{del}(C))^+|_{\mathsf{do}} \subseteq \mathsf{ar}(C)$. This states that timestamps of events on every object behave like a Lamport clock [22]. Conditions for COCV and COCA are similar.

There also exist consistency levels in between basic eventual consistency and per-object causal consistency, defined using so-called ***session guarantees*** [35]. We cover them in [12, §D].

Comparison with shared-memory consistency models. Interestingly, the specializations of the consistency levels defined by the axioms in Fig. 13 to the type `intreg` of LWW-registers are very close to those adopted by the memory model in the 2011 C and C++ standards [5]. Thus, POCA and POCV define the semantics of the fragment of C/C++ restricted to so-called *relaxed* operations; there this semantics is defined using *coherence* axioms, which are analogous to session guarantees [35]. COCV and COCA are close to the semantics of C/C++ restricted to *release-acquire* operations. However, C/C++ does not have an analog of EVENTUAL and does not validate THINAIR, since it makes the effects of speculations visible to the programmer [4]. We formalize the correspondence to C/C++ in [12, §D]. In the future, this correspondence may open the door to applying technology developed for shared-memory models to eventually consistent systems; promising directions include model checking [3, 9], automatic inference of required consistency levels [26] and compositional reasoning [4].

8. Related Work

For a comprehensive overview of replicated data type research we refer the reader to Shapiro et al. [32]. Most papers proposing new data type implementations [6, 31–33] do not provide their formal declarative specifications, save for the expected property (1) of quiescent consistency or first specification attempts for sets [6, 7]. Formalizations of eventual consistency have either expressed quiescent consistency [8] or gave low-level operational specifications [17].

An exception is the work of Burckhardt et al. [10, 13], who proposed an axiomatic model of causal eventual consistency based on visibility and arbitration relations and an operational model based on revision diagrams. Their store specification does not provide customizable consistency guarantees, and their data type specifications are limited to the sequential $\mathcal{S}$ construction from §3.2, which cannot express advanced conflict resolution used by the multi-value register, the OR-set and many other data types [32]. More significantly, their operational model does not support general op- or state-based implementations, and is thus not suited for studying the correctness or optimality of these commonly used patterns.

Simulation relations have been applied to verify the correctness of sequential [25] and shared-memory concurrent data type implementations [24]. We take this approach to the more complex setting of a replicated store, where the simulation needs to take into account multiple object copies and messages and associate them with structures on events, rather than single abstract states. This poses technical challenges not considered by prior work, which we address by our novel notion of replication-aware simulations.

The distributed computing community has established a number of asymptotic lower bounds on the complexity of implementing certain distributed or concurrent abstractions, including one-shot timestamp objects [20] and counting protocols [15, 30]. These works have considered either programming models or metrics significantly different from ours. An exception is the work of Charron-Bost [14], who proved that the size of vector clocks [29] is optimal to represent the happens-before relation of a computation (similar to the visibility relation in our model). Specifications of `mvr` and `orset` rely on visibility; however, Charron-Bost's result does not directly translate into a lower bound on their implementation complexity, since a specification may not require complete knowledge about the relation and an implementation may represent it in an arbitrary manner, not necessarily using a vector.

9. Conclusion and Future Work

We have presented a comprehensive theoretical toolkit to advance the study of replicated eventually consistent stores, by proposing methods for (1) specifying the semantics of replicated data types and stores abstractly, (2) verifying implementations of replicated data types, and (3) proving that such implementations have optimal metadata overhead. By proving both correctness and optimality of four nontrivial data type implementations, we have demonstrated that our methods can indeed be productively applied to the kinds of patterns used by practitioners and researchers in this area.

Although our work marks a big step forward, it is only a beginning, and creates plenty of opportunities for future research. We have already made the first steps in extending our specification framework with more features, such as mixtures of consistency levels [23] and transactions [34, 37]; see [11]. In the future we would also like to study more data types, such as lists used for collaborative editing [32], and to investigate metadata bounds for data type implementations other than state-based ones, including more detailed overhead metrics capturing optimizations invisible to the worst-case overhead analysis. Even though our execution model for replicated stores follows the one used by replicated data type designers [33], there are opportunities for bringing it closer to actual implementations. Thus, we would like to verify the algorithms used by store implementations [27, 34, 37] that our semantics abstracts from. This includes fail-over and session migration protocols, which permit clients to interact with multiple physical replicas, while being provided the illusion of a single virtual replica.

Finally, by bringing together prior work on shared-memory models and data replication, we wish to promote an exchange of ideas and results between the research communities of programming languages and verification on one side and distributed systems on the other.

Acknowledgements. We thank Hagit Attiya, Anindya Banerjee, Carlos Baquero, Lindsey Kuper and Marc Shapiro for comments that helped improve the paper. Gotsman was supported by the EU FET project ADVENT, and Yang by EPSRC.

References

[1] Riak key-value store. http://basho.com/products/riak-overview/.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), 1996.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.

[4] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[6] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. Technical Report 8083, INRIA, 2012.

[7] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *DISC*, 2012.

[8] A.-M. Bosneag and M. Brockmeyer. A formal model for eventual consistency semantics. In *IASTED PDCS*, 2002.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[10] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP*, 2012.

[11] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft Research, 2013.

[12] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality (extended version), 2013. http://research.microsoft.com/apps/pubs/?id=201602.

[13] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[14] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1), 1991.

[15] J.-Y. Chen and G. Pandurangan. Optimal gossip-based aggregate computation. In *SPAA*, 2010.

[16] N. Conway, R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[17] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC*, 1996.

[18] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[19] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[20] M. Helmi, L. Higham, E. Pacheco, and P. Woelfel. The space complexity of long-lived and one-shot timestamp implementations. In *PODC*, 2011.

[21] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1, 1972.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[23] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.

[24] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[25] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.

[26] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[28] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[29] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1989.

[30] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *PODC*, 1992.

[31] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[32] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[33] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[35] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[36] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[37] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report 8347, INRIA, 2013.