Mechanized Verification of Fine-grained Concurrent Programs

Ilya Sergey
IMDEA Software Institute
ilya.sergey@imdea.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Anindya Banerjee
IMDEA Software Institute
anindya.banerjee@imdea.org

Abstract
Efficient concurrent programs and data structures rarely employ coarse-grained synchronization mechanisms (i.e., locks); instead, they implement custom synchronization patterns via fine-grained primitives, such as compare-and-swap. Due to sophisticated interference scenarios between threads, reasoning about such programs is challenging and error-prone, and can benefit from mechanization.

In this paper, we present the first completely formalized framework for mechanized verification of full functional correctness of fine-grained concurrent programs. Our tool is based on the recently proposed program logic FCaSL. It is implemented as an embedded domain-specific language in the dependently-typed language of the Coq proof assistant, and is powerful enough to reason about programming features such as higher-order functions and local thread spawning.

By incorporating a uniform concurrency model, based on state-transition systems and partial commutative monoids, FCaSL makes it possible to build proofs about concurrent libraries in a thread-local, compositional way, thus facilitating scalability and reuse: libraries are verified just once, and their specifications are used ubiquitously in client-side reasoning. We illustrate the proof layout in FCaSL by example, and report on our experience of using FCaSL to verify a number of concurrent algorithms and data structures.

1. Introduction
It has been long recognized that efficient concurrency is of crucial importance for high-performant software. Unfortunately, proving correctness of concurrent programs, in which several computations can be executed in parallel, is difficult due to the large number of possible interactions between concurrent processes/threads on shared data structures.

One way to deal with the complexity of verifying concurrent code is to employ the mechanisms of so-called coarse-grained synchronization, i.e., locks. By making use of locks in the code, the programmer ensures mutually-exclusive thread access to critical resources, therefore, reducing the proof of correctness of concurrent code to the proof of correctness of sequential code. While sound, this approach to concurrency prevents one from taking full advantage of parallel computations. An alternative is to implement shared data structures in a fine-grained (i.e., lock-free) manner, so the threads manipulating such structures would be reaching a consensus via the active use of non-blocking read-modify-write operations (e.g., compare-and-swap) instead of locks.

Despite the clear practical advantages of the fine-grained approach to the implementation of concurrent data structures, it requires significant expertise to devise such structures and establish correctness of their behavior.

In this paper, we focus on program logics as a generic approach to specify a program and formally prove its correctness wrt. the given specification. In such logics, program specifications (or specs) are represented by Hoare triples \( \{ P \} c \{ Q \} \), where \( c \) is a program being described, \( P \) is a precondition that constrains a state in which the program is safe to run, and \( Q \) is a postcondition, describing a state upon the program’s termination. Modern logics are sufficiently expressive: they can reason about programs operating with first-class executable code, locally-spawned threads and other features omnipresent in modern programming. Verifying a program in a Hoare-style program logic can be done structurally, i.e., by means of systematically applying syntax-directed inference rules, until the spec is proven.

Importantly, logic-based verification of fine-grained concurrency requires reasoning about a number of concepts that don’t have direct analogues in reasoning about sequential or coarse-grained concurrent programs:

1. **Custom resource protocols.** Each shared data structure (i.e., a resource) that can be used by several threads concurrently, requires a specific “evolution protocol”, in order to enforce preservation of the structure’s consistency. In contrast with coarse-grained case, where the protocol is fixed to be locking/unlocking, a fine-grained resource comes with its own notion of consistency and protocol.

2. **Interference and stability.** Absent locking, local reasoning about a shared resource from a single thread’s perspective should manifest the admissible changes that can be made by other threads that interfere with the current one. Every thread-local assertion about a fine-grained data structure’s state should be stable, i.e., invariant under possible concurrent modifications of the resource.

3. **Work stealing.** This common concurrent pattern appears in fine-grained programs due to relaxing the mutual exclusion policy; thus several threads can simultaneously operate with a single shared resource. The “stealing” happens when a thread is scheduled for a particular task involving the resource, but the task is then accomplished...
by another thread; however, the result of the work is nevertheless ascribed to the initially assigned thread. In addition, Hoare-style reasoning about coarse- or fine-grained concurrency requires a form of (4) **auxiliary state** to partially expose the internal threads’ behavior and relate local program assertions to global invariants, accounting for specific threads’ contributions into a resource [Jones 2010].

These aspects, critical for Hoare-style verification of fine-grained concurrent programs, have been recognized and formalized in one form or another in a series of recently published works by various authors [Feng et al. 2007; Vafeiadis and Parkinson 2007; Feng 2009; Dinsdale-Young et al. 2010; Jacobs and Piessens 2011; Turon et al. 2013; Svendsen and Birkedal 2014; da Rocha Pinto et al. 2014], providing logics of increasing expressivity and compositionality. In formal proofs of correctness of concurrent libraries, that are based upon these logical systems, the complexity is **not** due to the libraries’ sizes in terms of lines of code, but predominantly due to the intricacy of the corresponding data structure invariant, and the presence of thread interference and work stealing. This fact, in contrast to proofs about sequential and coarse-grained concurrent programs, requires one to establish stability of every intermediate verification assertion. Needless to say, manual verification of fine-grained concurrent programs therefore becomes a challenging and error-prone task, as it’s too easy for a human prover to forget about a piece of resource-specific invariant or to miss an assertion that is unstable under interference; thus the entire reasoning can be rendered unsound.

Since the process of structural program verification in a Hoare-style logic is largely mechanical, there have been a number of recent research projects that target mechanization and automation of the verification process by means of embedding it into a general-purpose proof assistant [Nanevski et al. 2008, 2010; Shao 2010; Chlipala 2011], or implementing a standalone verification tool [Leino and Müller 2009; Cohen et al. 2009; Jacobs et al. 2011]. However, to the best of our knowledge, none of the existing tools has yet adopted the logical foundations necessary for compositional reasoning about all of the aspects (1)–(4) of fine-grained concurrency. This is the gap which we intend to fill in this work.

In this paper, we present a framework for mechanized verification of fine-grained concurrent programs based on the recently proposed **Fine-grained Concurrent Separation Logic** (FCSL) by Nanevski et al. [2014]. ¹ FCSL is a library and an embedded domain-specific language (DSL) in the dependently-typed language of Coq proof assistant [2014]. Due to its logical foundations, FCSL, as a verification tool and methodology for fine-grained concurrency, is:

- **Uniform**: FCSL’s specification model is based on two basic constructions: *state-transition systems* (STSs) and *partial commutative monoids* (PCMs). The former describe concurrent protocols and thread interference, whereas the latter provide a generic treatment of shared resources and thread contributions, making it possible to encode, in particular, the work stealing pattern. Later in this paper, we will demonstrate how these two components are sufficient to specify a large spectrum of concurrent algorithms, data structures, and synchronization mechanisms, as well as to make the proofs of verification obligations to be uniform.

- **Expressive**: FCSL’s *specification* fragment is based on the propositional fragment of Calculus of Inductive Constructions (CIC) [Bertot and Castéran 2004]. Therefore, FCSL can accommodate and compose arbitrary mathematical theories, *e.g.*, PCMs, heaps, arrays, graphs, *etc*.

- **Realistic**: FCSL’s *programming* fragment features a complete toolset of modern programming abstractions, including user-defined algebraic datatypes, first-class functions and pattern matching. That is, any Coq program is also a valid FCSL program. The monadic nature of FCSL’s embedding into Coq [Nanevski et al. 2006] makes it possible to encode a number of computational effects, *e.g.*, thread spawning and general recursion. This makes programming in FCSL similar to programming in ML or Haskell.

- **Compositional**: Once a library is verified in FCSL against a suitable spec, its code is not required to be re-examined ever again: all reasoning about the client code of that library can be conducted out of the specification. The approach is thus scalable: even though the proofs for libraries might be large, they are done just once.

- **Interactive**: FCSL benefits from the infrastructure, provided by Coq’s fragment for mechanized reasoning, enhanced by Ssreflect extension [Gonthier et al. 2009]. While the verification process can’t be fully automated (as full functional correctness of concurrent programs often requires stating specs in terms of higher-order predicates), the human prover nevertheless can take advantage of all of Coq’s tools to discharge proof obligations.

- **Foundational**: The soundness of FCSL, as a logic has been proven in Coq with respect to a version of denotational semantics for concurrent programs in the spirit of Brookes [2007]. Moreover, since FCSL program specs are encoded as Coq types, the soundness result scales to the *entire language* of Coq, not just a toy core calculus. This ensures the absence of bugs in the whole verification tool and, as a consequence, in any program, which is verified in it.

In the remainder of the paper, we will introduce the FCSL framework by example, specifying and verifying full functional correctness of a characteristic fine-grained program—a concurrent spanning tree algorithm. Starting from the intuition behind the algorithm, we will demonstrate the common stages of program verification in FCSL. We next explain some design choices, made in the implementation of FCSL, and report on our experience of verifying a number of benchmark concurrent programs and data structures: locks, memory allocator, concurrent stack and its clients, an

---

¹ Hereinafter, we will be using the acronym FCSL to refer both to the Nanevski et al.’s logical framework and to our implementation of it.
The graph is implemented as a memory region where each node has three components—ptr to the node's successor, or null if no successor (line 6); bool indicating whether the node is marked (line 7); and int indicating the values of the node's left and right successors, or null if a successor doesn’t exist.

If x is null, span returns false (line 2). Otherwise, it tries to mark the node by invoking the compare-and-swap (CAS) operation (line 4). If CAS fails, then x was previously marked, i.e., included in the spanning tree by another call to span, so no need to continue (line 10). If CAS succeeds, two new parallel threads are spawned (line 6): span is called recursively on the left (x.l) and right (x.r) successors of x, returning respectively the booleans r_l and r_r upon termination. When r_l is false, x.l has already been marked, i.e., span has already found a spanning subtree that includes x.l but doesn’t traverse the edge from x to x.l. That edge is superfluous, and thus removed by nullifying x.l (line 7). The behavior is dual for x.r. Figure 3 illustrates a possible execution of span.

Why does span compute a tree? Assume that (1) the graph initially represented in memory is connected, and that (2) it is modified only by recursive calls to span, with no external interference. To see that span obtains a tree, consider four cases, according to the values of r_l and r_r. If r_l = r_r = true, then the calls to span have, by recursive assumption, computed trees from subgraphs rooted at x.l and x.r to trees. These trees have disjoint nodes, and there are no edges connecting them. As will be shown in Section 3, this will follow from a property that each tree is maximal wrt. the resulting

Figure 1: Concurrent spanning tree construction procedure.

Figure 2: FCSL implementation of the span procedure.
Figure 3: Stages of concurrent spanning tree construction. A node is painted grey right after a corresponding thread successfully marks it (line 4 of Figure 1). It is painted black right before the thread returns true (line 9). A black subtree is logically ascribed to a thread that marked its root. ✔ indicates a child thread exploring an edge and succeeding in marking its target node; ✗ indicates a thread that failed to do so. (1) the main thread marks a and forks two children; (2) the children succeed in marking b and c; (3) only one thread succeeds in marking c; (4) the processing of d and e is done; (5) the redundant edges b → e and c → e are removed by the corresponding threads; (6) the initial thread joins its children and terminates.

The state of each concurroid is divided into three components \([self | joint | other]\). The joint component describes shared state that all threads can change. The self and other components are owned by the observing thread, and its environment, respectively, and may be changed only by its owner. If there are two threads \(t_1\) and \(t_2\) operating over a state, the proof of \(t_1\) will refer by self to the private state of \(t_1\), and by other to the private state of \(t_2\), and the roles are reversed for \(t_2\). This thread-specific, aka. subjective, split into self, joint and other is essential for making the proofs insensitive to the number of threads forked by the global program, and the order in which this is done [Ley-Wild and Nanavski 2013]. We also note that the self and other components have to be elements of a PCM, i.e., a set \(U\) with an associative and commutative join operation \(\bullet\), and a unit element 1.

All three components may contain real state, i.e., heap, or auxiliary state [Lucas 1968; Owicki and Gries 1976], which is kept for logical specification, but is erased before execution. In the case of the \(\text{span}\) procedure, the joint component is the heap encoding the graph to be spanned, as described above. The self and other components are auxiliary state, consisting of sets of nodes (i.e., pointers) marked by the observing thread and its environment, respectively. These components are elements of a PCM of sets with disjoint union \(\cup\) as \(\bullet\), and the empty set as the unit. Thus, self \(\bullet\) other is the set of marked nodes of the graph in joint.

Transitions of a concurroid are binary relations between states. They describe the state modifications that threads may do, if they are to respect the agreement represented by the concurroid. The concurroid for \(\text{span}\), named \(\text{SpanTree}\) in the sequel, has two non-trivial transitions, which we call \(\text{marknode}\_\text{trans}\) and \(\text{nullify}\_\text{trans}\). Additionally, every concurroid has the trivial identity transition \(\text{id}\_\text{e}\). A thread performs \(\text{marknode}\_\text{trans}\) when it successfully marks a node. Whenever the bit \(m\) of a node \(x\) is set, the pointer \(x\) is also added to the auxiliary self state of the thread that performed the operation. Thus, the self component correctly tracks the nodes marked by a thread. A thread performs \(\text{nullify}\_\text{trans}\) when it removes an edge out of a marked node. However, this transition can only be taken in states in which \(x\) is in the self component; thus, only a thread that marked \(x\) can take this transition.

Atomic actions. Concurroids logically specify the behavior of threads, and one needs a way to tie the logical specs to actual program operations, such as, e.g., C\&S. An atomic action is a program operation that can change the heap by one read-modify-write operation, and simultaneously change the auxiliary state. In Section 3, we expand on how actions are defined. For now, we just briefly describe the three actions required for implementation of \(\text{span}\) in FCSL.

The \(\text{trymark}\) action attempts to mark a node \(x\), and move \(x\) into the self auxiliary component simultaneously. Operationally, i.e., when the auxiliary state is erased, it corresponds to the C\&S on line 4 of Figure 1. Logically, if successful, it corresponds to a \(\text{marknode}\_\text{trans}\) transition in the concurroid. If unsuccessful, it corresponds to the concurroid’s idle transition. The \(\text{nullify}\_\text{action}\) invoked with an argument \(x\), and a two-valued indicator \(\text{side}\) (Left or Right), sets the \(x.l\) (or \(x.r\), depending on \(\text{side}\)) pointer to null, but emits a precondition that \(x\) is in self. Operationally, it corresponds to the assigning \(\text{null}\) on lines 7 and 8 of Figure 1. Logically, it corresponds to taking the \(\text{nullify}\_\text{trans}\) transition. Finally, \(\text{read}\_\text{child}\) atomic action, invoked with arguments \(x\) and \(\text{side}\), returns the pointer \(x.l\) (or \(x.r\), depending on \(\text{side}\)). It also emits a precondition that \(x\) is in self. Operationally, it corresponds to the pointer reads on line 6 in Figure 1. Logically, it corresponds to the concurroid’s idle transition.

Figure 2 shows how the actions are used to translate the \(\text{span}\) procedure in Figure 1 into FCSL.

Hoare specifications as types and stability. In Figure 2, \(\text{span}\) is ascribed the type \(\text{span}\_\text{tp}\). While Section 3 defines it formally, here we provide some basic intuition for it.

Among other components, the type \(\text{span}\_\text{tp}\) contains the formal pre- and postconditions, ascribed to \(\text{span}\). Hence, it is a user-defined type, rather than inferred by the system. Also, \(\text{span}\_\text{tp}\) is declared as the type of the fixpoint combinator \(\text{fix}\)’s argument \(\text{loop}\), and thus serves as the “loop invariant” as well. The components of \(\text{span}\_\text{tp}\) provide the following
information: (a) The precondition in \texttt{span\_tp} ensures that the input node \(x\) is either \texttt{null} or points to a node in the heap. (b) If \texttt{span} returns \texttt{false}, the postcondition ensures that \(x\) is either \texttt{null} or is marked in the graph, and the thread hasn’t marked any other nodes during the call. (c) If \texttt{span} returns \texttt{true}, the postcondition states that \(x \neq \texttt{null}\), and the thread being specified has marked a set of nodes \(t\), which form a maximal tree in the final graph with root \(x\); moreover, \(t\)’s front \(wrt\) initial graph is marked, possibly by other threads.

We further note that the assertions (a)–(c) will be \textit{stable} \(wrt\) interference, \textit{i.e.}, they remain valid no matter which transitions of the \texttt{span} concurroid the interfering threads take. Proving stability is an important component of FCSL.

Typically, every spec used in FCSL will be \texttt{stable}, or else it won’t be possible to ascribe it to a program. In the next section, we will exhibit several stable example specifications \(wrt\) the concurroid for \texttt{span}, including \texttt{span\_tp}.

\textbf{Hiding from external interference.} The type \texttt{span\_tp} specifies the calls to \texttt{span} in the loop, but the top-most call to \texttt{span} requires a somewhat stronger context, as it should know that no other threads, aside from its children, can interfere on the shared graph. Without this knowledge, explicitly stated by the assumption (2), it is impossible to show that \texttt{span} actually constructs a \textit{spanning} tree, so we need to enforce it.

The encapsulation of interference is achieved in FCSL by the program constructor \texttt{hide}. For instance, writing \texttt{hide} \texttt{g1} \texttt{\{ span\(x\) \}} makes it apparent to the type system of FCSL that the top-most call to \texttt{span} runs without interference on the shared graph. More precisely, the call, \texttt{span(x)}, within hide will execute relative to the protocol implemented by the \texttt{SpanTree} concurroid. Any threads spawned internally will also follow this protocol. Outside of hide, the active protocol allows manipulation of the caller’s private state only, but is oblivious to the \texttt{span} protocol. The surrounding threads thus cannot interfere with the inside call to \texttt{span}. In this sense, hide installs a concurroid in a scoped manner, and then executes the supplied program relative to that concurroid. The role of hide is thus purely logical, and operationally it behaves as a no-op.

The annotation \(\Phi\) is a predicate over heaps that indicates the portion of the private heap of \texttt{span}’s caller onto which the \texttt{span} concurroid should be installed. In the case of \texttt{span}, \(\Phi\) merely describes the nodes of the graph we want to span. \(\emptyset\) indicates that \texttt{span} is initially invoked with the empty auxiliary state, \textit{i.e.}, no nodes are initially marked.

3. **Outline of the mechanized development**

We next discuss how the above informal overview is mechanized in Coq. We start with the definition of \texttt{span\_tp} and proceed to explain all of its components. The specifications and code shown will be very similar to what’s in our Coq files, though, to improve presentation, we occasionally take liberties with the order of definitions and notational abbreviations. We do not assume any familiarity with Coq, and explain the code displays as they appear. We also omit the proofs and occasional auxiliary definitions, which can be found in the FCSL code, accompanying the paper.

\textbf{The definition of the type} \texttt{span\_tp} is given in Figure 4. It is an example of a \textit{dependent type}, as it takes formal arguments in the form of variables \(x, i\) and \(g_1\), that the body of the type can use, \textit{i.e., depend on}. The roles of the variables differ depending on the keyword that binds them. For example, the Coq keyword \texttt{forall} binds the variable \(x\) of type \texttt{ptr}, and indicates that \texttt{span\_tp} is a specification for a procedure that has \(x\) as input. Indeed, \texttt{span} is exactly such a procedure, as apparent from Section 2. Using \texttt{forall} to bind \(x\) allows \(x\) to be used in the body of \texttt{span\_tp}, \textit{but also in the body of} \texttt{span} (Figure 2). On the other hand, \(i\) and \(g_1\) are bound by FCSL binder \(\{ \ldots \}\). This binding is different; it allows \(i\) and \(g_1\) to be used in the body of \texttt{span\_tp}, \textit{but not in the procedure} \texttt{span}. In terminology of Hoare-style logic, \(i\) and \(g_1\) are \textit{logical variables} (aka. \textit{ghosts}), which are used in specs, but not in the code. \texttt{STsep} is a Coq macro, defined by FCSL announcing that what follows is a Hoare-style partial correctness specification for a concurrent program. The component \texttt{SpanTree} \texttt{sp} in the brackets is the concurroid whose protocol \texttt{span\_tp} respects. We will define \texttt{SpanTree} shortly. Finally, the parentheses include the precondition and the postcondition (defined as Coq’s \textit{functions}) that we want to ascribe to \texttt{span}. The precondition is a predicate over the pre-state \(s_1\). The postcondition is a predicate over the boolean result \(r\) and post-state \(s_2\). As customary in many programming languages, Coq included, we omit the types of various variables when the system can infer them (\textit{e.g.}, the variables \(i, s_1\), and \(s_2\) are all of type \texttt{state}).

The precondition says that the input \(x\) is either \texttt{null} (since \texttt{span} can be called on a leaf node), or belongs to the domain of the input heap, and hence is a valid node in the heap-represented graph. The heap is computed as the projection \texttt{joint} out of the input state \(s_1\), which \(i\) snapshots. The projections \texttt{self} and \texttt{other} are sets of marked nodes, belonging to the caller of \texttt{span} and to its environment, respectively.

The postcondition says that in the case the return result is \(r = \texttt{false}\), the pointer \(x\) was either \texttt{null} or already marked. Otherwise, there is a set of nodes \(t\) which is freshly marked by the call to \texttt{span}; that is, \texttt{self} \(s_2\) is a disjoint union \((\vee+)\) of \(t\) with the set of nodes marked in the pre-state \texttt{self} \(i\). The set \(t\) satisfies several important properties. First, \(t\) is a subtree in the graph, \(g_2\), of the post-state \(s_2\), with root \(x\).
Second, the tree \( \tau \) is maximal, i.e., it cannot be extended into a larger tree by adding more nodes from \( g_2 \), as all the edges between \( \tau \) and the rest of the graph have been severed by \( \text{span} \). Third, all the nodes immediately reachable from \( \tau \) in the initial state \( \delta \) (i.e., \( \tau \)'s front) are marked in \( g_2 \) either by this or some other thread (\( \text{self} \ a \not\!\not\!\not\!\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\not\nota
itself, \textit{i.e.}, no edges connect $ty_1$ and $ty_2$ (the notation $#x$ is concrete syntax for the singleton finite map containing node $x$). This lemma is essential in proving that \texttt{span} produces a tree, as mentioned in Section 2 for the case $r_l = r_r = \text{true}$.  

\textbf{Lemma} maxTree2 $x y_1 y_2 ty_1 ty_2$ :  
edge $x =_1 [y_1 ; y_2] \rightarrow$ tree $y_1 ty_1 \rightarrow$ maximal $ty_1 \rightarrow$  
tree $y_2 ty_2 \rightarrow$ maximal $ty_2 \rightarrow$ valid $(ty_1 \downarrow ty_2) \rightarrow$  
tree $x (\#x \downarrow ty_1 \downarrow ty_2)$.

The second lemma shows that subgraph is monotone \textit{wrt.} the stepping of environment threads in the \texttt{SpanTree} concurroid.

\textbf{Lemma} subgraph_steps $s1 s2$  
$\text{(g1 : graph (joint s1)) (g2 : graph (joint s2))}$  
env_steps (SpanTree $sp$) $s1 s2$ $\rightarrow$ subgraph $g1 g2$.

We used this lemma as the main tool in establishing a number of stability properties in Coq, related to the conjunctions from the definition of subgraph $g1 g2$. For example, the lemma implies that if $x$ is a node of joint $s1$, then it is so in a stable manner; that is, $x$ is a node in joint $s2$ for any $s2$ obtained from $s1$, by environment interference.

\textbf{SpanTree} \texttt{concurroid}. Next we define the \texttt{SpanTree} concurroid. Being an STS, the definition includes the specification of the state space, and transitions between states. In the case of concurroids, we have an additional component: \textit{labels} (semantically, natural numbers) that differentiate instances of the concurroid. Thus the definition of \texttt{SpanTree} is parametrized by the variable $sp$, which makes it possible to use several instances of \texttt{SpanTree} with different labels in a specification of a single program. For example, say we want to run two \texttt{span} procedures in parallel on disjoint heaps. Such a program could be specified by a Cartesian product of \texttt{SpanTree} $sp1$ and \texttt{SpanTree} $sp2$, where the different labels $sp1$ and $sp2$ instantiate the variable $sp$.

The state space of \texttt{SpanTree} is defined by the following state predicate $coh$, which we call coherence predicate.

\textbf{Variable} $sp : \text{nat}$.

\textbf{Definition} $coh s$ := exists $g : \text{graph (joint s)}$,  
s $= sp$ $\rightarrow$ \{self $s$, joint $s$, other $s$\} $\wedge$  
valid (self $s \wedge$ other $s$) $\wedge$  
forall $x$, $\notin$ dom (self $s \wedge$ other $s$) = mark $g x$.

The coherence predicate codifies that the state $s$ is a triple, \{self $s$, joint $s$, other $s$\}, and that it is labelled by $sp$. The proof $g$ is a witness that the joint component is a graph-shaped heap. The conjunct valid (self $s \wedge$ other $s$) says that the \texttt{self} and \texttt{other} components of the auxiliary state are disjoint; their union is a finite map which is valid, \textit{i.e.}, doesn’t contain duplicate keys. Finally, the most important invariant is that a node $x$ is contained in either self or other subjective view \texttt{iff} it’s marked in the joint graph.

The metatheory of FCSL [Nanevski et al. 2014, §4] requires the coherence predicates to satisfy several properties that we omit here, but prove in our implementation. The most important property is the \textit{fork-join closure}, stating that the state space is closed under realignment of self and other components. In other words, one may subtract a value from self and add it to other (and vice versa), without changing the coherence of the underlying state.

\texttt{SpanTree} $sp$ contains two non-idle transitions. Transition \texttt{marknode_trans}, parametrized by the node $x$, describes how an unmarked $x$ is physically marked in the joint graph, and simultaneously added to the self component. The transition \texttt{nullify_trans} is parametrized by node $x$ and the direction $c$, indicating the successor of $x$ (left or right) that must be cut off from the graph. We omit the definitions of the functions $\text{mark_node}$ and $\text{null_edge}$ that describe the physical changes performed by the two transitions to the underlying shared graph. These can be found in the Coq code.

\textbf{Definition} $\text{marknode_trans}$ $x s s' :=$ exists $g : \text{graph (joint s)}$,  
$\{\text{mark g x} \wedge\}$ joint $s' = \text{mark_node g x} \wedge$  
self $s' = \#x \wedge$ self $s \wedge$ other $s \wedge coh s \wedge coh s'$.

\textbf{Definition} $\text{nullify_trans}$ $x (c : \text{side}) s s' :=$ exists $g : \text{graph (joint s)}$,  
$\text{x \notin dom (self s \wedge joint s' = \text{null_edge g c x} \wedge}$  
self $s' = \#x \wedge$ other $s \wedge coh s \wedge coh s'$.

The FCSL metatheory requires that transitions also satisfy several properties. For example, $\text{marknode_trans}$ and $\text{nullify_trans}$ preserve the other-component and the coherence predicate, as immediately apparent from their definitions. They also preserve the footprint of the underlying state, \textit{i.e.}, they don’t add or remove any pointers. Adding and removing heap parts can be accomplished by communication between concurroids, as we briefly discuss in Section 4.

The coherence predicate, the transitions, and the proofs of their properties are packaged into a \textit{dependent record}ootnote{A type-theoretic variant of a C\texttt{struct}, where fields can contain proofs.} \texttt{SpanTree} $sp$, which encapsulates all that’s important about a concurroid. Thus, we use the power of dependent types in an essential way to build mathematical abstractions, such as concurroids, that are critical for reusing proofs.

\textbf{Atomic actions}. We next illustrate the mechanism for defining atomic actions in FCSL. The role of atomic actions is to perform a single physical memory operation on the real heap, simultaneously with an arbitrary modification of the auxiliary part of the state. In FCSL, we treat the real and auxiliary state uniformly as they both satisfy the same PCM laws. We specify their effects in one common step, but afterwards prove a number of properties that separate them. For instance, for each atomic action we always prove the \textit{erosure property} that says that the effect of the action on the auxiliary state doesn’t affect the real state.

Specifically, the effect of the \texttt{trymark} action is defined by the following relation between the input pointer $x$, the pre-state $s1$, post-state $s2$ and the return result $r$ of type \texttt{bool}.

\textbf{Definition} \texttt{trymark_step} $x : \text{ptr} s1 s2 (r : \text{bool}) :=$ exists $g : \text{graph (joint s1)}$,  
x $\in$ dom (joint $s1$) $\wedge$ other $s2$ = other $s1$ $\wedge$  
if $\text{mark g x}$  
then $r = \text{false} \wedge$ joint $s2 = \text{joint s1} \wedge$ self $s2$ = self $s1$  
else $r = \text{true} \wedge$ joint $s2 = \text{mark_node g x} \wedge$  
self $s2$ = $\#x \wedge$ self $s1$.

The relation requires that $x$ is a node in the pre-state graph ($x \in$ dom (joint $s1$)). If $x$ is unmarked in this graph, then the action returns true, together with marking the
node physically in the real state (employing the function mark_node already used in marknode_trans). Otherwise, the state remains unchanged, and the action’s result is \texttt{false}. Notice that when restricted to the real heap, i.e., if we ignore the auxiliary state in \texttt{self s1} and other \texttt{s1}, the relation essentially describes the effect of the \texttt{CAS} command on the mark bit of \texttt{x}. Thus, trymark \textit{erases} to \texttt{CAS}.

There are several other components that go into the definition of an atomic action. In particular, one has to prove that transitions are \texttt{total}, \texttt{local}, and \texttt{frameable} in the sense of Separation Logic, and then ascribe to each action a stable specification. However, the most important aspect of action definitions is to identify their behavior with some transition in the underlying concurroid. For example, trymark behaves like marknode_trans transition of SpanTree if it succeeds, and like \texttt{idle} if it fails. Actions may also change state of a number of concurroids simultaneously, as we will discuss in Section 4. In the interest of brevity, we omit the formal definition of all these properties here, but they can be found in the accompanying Coq files.

\textbf{Scoped concurroid allocation and hiding.} The \texttt{span tp} type from Figure 4 operates under \textit{open-world assumption} that \texttt{span} runs in an environment of interfering threads, which, however, respect the transitions of the SpanTree concurroid. If one wants to protect \texttt{span} from interference, and move to \textit{closed-world assumption}, the top-most call must be enclosed within \texttt{hide}. We next show how to formally do so.

The hide construct allocates a new lexically-scoped concurroid from a local state of a particular thread.\textsuperscript{4} The description of how much local heap should be “donated” to the concurroid creation is provided by the user-supplied predicate \(\Phi\), called \textit{decoration} predicate. In addition to the heap, the predicate scopes over the auxiliary \textit{self} value, while the auxiliary \textit{other} is fixed to the PCM unit, to signal that there’s no interference from outside threads. In the case of \texttt{span}, the decoration predicate is as follows.

\begin{verbatim}
Definition graph_dec sp (g : heap * ptr_set) s :=
  exists (pf : graph g.1), s = sp --> [g.2, g.1, Unit] \& coh s.
\end{verbatim}

We can now write out a new type \texttt{span_root tp}, to specify the top-most call to \texttt{span}, under the closed-world assumption that there’s no interference. Parametrizing \texttt{wrt.} the locally-scoped variable \texttt{h1 : heap} that snapshots the initial heap, the type is the following one.

\begin{verbatim}
Definition span_root tp (x : ptr) :=
  [g1 : graph h1], STsep [Priv pv]
  (* precondition predicate *)
  (fun s1 => (forall y, ~((mark g1 y)) \&
    pv_self s1 = h1 \& x \in dom h1 \& connected g1 x,
    (* postcondition predicate *)
    fun (t : bool) s2 => exists (g2 : graph (pv_self s2) t),
      (forall x, (edgl g2 x \in [: null; edgl g1 x]) \&
        (edgr g2 x \in [: null; edgr g1 x]) \&
    tree g2 x t \& dom t = i dom h1).
\end{verbatim}

The precondition says that the argument \texttt{x} is the root of the graph \texttt{g1} stored in \texttt{h1}, and all the nodes of \texttt{g1} are reachable from \texttt{x}. The postcondition says that the final heap’s topology is a tree \(t\), whose edges are a subset of the edges of \texttt{g1}, but whose nodes include \textit{all} the nodes of \texttt{g1}. Thus, the tree is a spanning one. The program satisfying this spec is a call to \texttt{span}, wrapped into \texttt{hide}, annotated with the decorating functions. We also supply \texttt{h1} as the initial heap, and \texttt{Unit} of the PCM of finite sets (hence, the empty set), as the initial value for \texttt{self}, which indicates that \texttt{span} is invoked with the empty set of marked nodes.

\begin{verbatim}
Program Definition span_root x : span_root tp x :=
  Do (priv_hide pv (graph_dec sp) (h1, Unit) [span sp x]).
\end{verbatim}

Coq will emit a proof obligation that the pre and post of \texttt{span tp} can be weakened into those of \texttt{span_root tp} under the closed-world assumption that other \texttt{s2 = Unit}. This proof is in the development, accompanying this paper.

\section{More examples}

We next briefly illustrate two additional features of FCSL that our implementation uses extensively: concurroid composition and reasoning about higher-order concurrent structures with work stealing.

\textbf{Composing concurrent resources.} The \texttt{span} algorithm uses only one concurroid \texttt{SpanTree}, allocated by hide out of the concurroid \texttt{Priv} for thread-local state. In general, FCSL specs can span multiple primitive concurroids, of the same or different kinds, which are \textit{entangled} by interconnecting special \textit{channel}-like transitions \cite{Nanevski:14}. The interconnection implements synchronized communication, by which concurroids exchange heap ownership. Entangling several concurroids yields a new concurroid. Omitting the formal details of the entanglement operators, let us demonstrate a program whose spec uses a composite concurroid.

\begin{verbatim}
Definition alloc : {h : heap}, STsep [entangle (Priv pv) ALock]
  (fun s1 => pv_self s1 = h, fun r s2 => exists B (r \& B), pv_self s2 = r --> y \& \& h) :=
  ffix (fun (loop : unit --> alloc tp) (_ : unit) =>
    Do (res <-- try_alloc;
      if res is Some r then loop tt else loop tt)).
\end{verbatim}

The \texttt{alloc} procedure implements a pointer allocator. Its postcondition says that the initial heap \texttt{h} is augmented by a new pointer \texttt{r} storing some value \texttt{v} (\texttt{r} \rightarrow \texttt{v}). The heap \texttt{h} is part of the \texttt{Priv} concurroid, as evident by the projection \texttt{pv_self} in the precon. The pointer \texttt{r} is logically transferred from the concurroid \texttt{ALock} which implements a coarse-grained (\textit{i.e.}, lock-protected) concurrent allocator. Hence, the whole procedure \texttt{alloc} uses the composed concurroid \texttt{[entangle (Priv pv) ALock]}. The body of \texttt{alloc} implements a simple spin-loop, trying to acquire the pointer by invoking the \texttt{try.alloc} procedure, omitted here.

Whereas separation logic \cite{Reynolds:02} always assumes allocation as a primitive operation, the above example illustrates that in FCSL, allocation is definable. One can also define a new variant of the \texttt{STsep} type that automatically en-
tangles the underlying concurroid with ALock, thus enabling allocation without the user having to explicitly do so herself. **Higher-order specifications.** Due to embedding in Coq, FCSL is also capable of specifying and verifying higher-order concurrent data structures, which we illustrate by an example of a universal non-blocking construction of flat combining by Hendler et al. [2010].

A flat combiner (FC) is a higher-order structure, whose method flat_combine takes a sequential state-modifying function \( f \) and its argument \( v \), and works as follows. While for the client, invoking \( \text{flat}_{-}\text{combine}(f, v) \) looks like a sequence \( \text{lock}; f(v); \text{unlock} \), in reality, the structure implements a sophisticated concurrent behavior. Instead of expensive locking and unlocking, the calling thread doesn’t run \( f \), but only registers \( f \) to be executed on \( v \). One of the threads then becomes a combiner and executes the registered methods on behalf of everyone else. Since only the combiner needs exclusive access to the data structure, this reduces contention and improves cache locality. This design pattern is known as work stealing or helping: a thread can complete its task even without accessing the shared resource.

To specify FC, we parametrize it by a sequential data structure and a validity predicate \( fc_{-}R \), which relates a function \( f \) (from a fixed set of allowed operations), the argument of type \( fc_{-}\text{inT} f \), result of type \( fc_{-}\text{outT} f \) and the contribution of type \( fc_{-}\text{pcm} \). The last entry is a description of what \( f \) does to the shared state, expressed in abstract algebraic terms as a value from a user-supplied PCM.

\[ \text{Variable} \ fc_{-}R : \forall f, fc_{-}\text{inT} f \rightarrow fc_{-}\text{outT} f \rightarrow fc_{-}\text{pcm} \rightarrow \text{Prop}. \]

The spec of the \( \text{flat}_{-}\text{combine} \) is then given in the context of three entangled concurroids: \( \text{Priv} \) for thread-local state, a lock-based allocator \( \text{Alloc} \), adapted from the previous example (since a sequential function \( f \) might allocates new memory), and a separate concurroid \( \text{FlatCombine} \).

\[ \text{Definition} \ PA := (\text{entangle} (\text{Priv} pv) \text{Alloc}). \]
\[ \text{Definition} \ W := (\text{entangle} (PA \text{FlatCombine} fc)). \]
\[ \text{Program Definition} \ flat_{-}\text{combine} f (v : fc_{-}\text{inT} f) : \text{Step} [W] (\text{fun} s1 := pv_{-}\text{self} s1 = \text{Unit} \wedge fc_{-}\text{self} s1 = \text{Unit},
\quad \text{pv}_{-}\text{self} s2 = \text{Unit} \wedge fc_{-}\text{self} s2 = g / fc_{-}R f v w g) := \ldots \]

The precondition says that \( \text{flat}_{-}\text{combine} \) executes in the empty initial heap \( (pv_{-}\text{self} s1 = \text{Unit}) \), and hence by framing, in any initial heap. Similarly, the initially assumed effects of the calling thread on the shared data structure are empty \( (fc_{-}\text{self} s1 = \text{Unit}) \), but can be made arbitrary by applying FCSL’s frame rule to the spec of \( \text{flat}_{-}\text{combine} \). The postcondition says that there exists an abstract PCM value \( g \) describing the effect of \( f \) in terms of PCM elements \( (fc_{-}R f v w g) \). Moreover, the effect of \( g \) is attributed to the invoking thread \( (fc_{-}\text{self} s2 = g) \), even though in reality \( f \) could be executed by the combiner, on behalf of the calling thread. In our Coq implementation, we instantiated the FC structure with a sequential stack, showing that the result has the same spec as a concurrent stack implementation.

5. **Elements of FCSL infrastructure**

In this section we sketch two important parts of FCSL machinery, used to simplify construction of proofs. **Extracting concurroid structure via getters.** When working with compositions of multiple concurroids, as in examples listed in Section 4, one frequently has to select the \( self, \) \( joint \) or other components that belong to one of the composed concurroids.

A naive way of doing this is to describe the state space of the composition concurroid using existentials that abstract over the concurroid-specific fields. E.g., in the case of flat combiner, which composes three concurroids \( \text{Priv}, \text{Alloc} \) and \( \text{FlatCombine} \), we could use three existentials to abstract over \( self \) \( joint \) \( other \) fields for \( \text{Priv} \), another three for \( \text{Alloc} \), and three more for \( \text{FlatCombine} \). To access any of the fields, we have to destruct all nine of the existentials. This quickly becomes tedious and results in proofs that are obscured by such existential destruction.

Our alternative approach develops a systematic way of projecting the fields associated with each concurroid, based on the concurroid’s label. Thus, for example, we can write \( self \) \( pv \) \( \alpha \) to obtain the \( self \) component of \( \alpha \), associated with a concurroid whose label is \( pv \). The identifier \( pv_{-}\text{self} \) we used in the spec for \( \text{span}_{-}\text{root} \) and for \( \text{flat}_{-}\text{combine} \) is a notational abbreviation for exactly this projection. While this is a simple and obvious idea, its execution required a somewhat involved use of dependently-typed programming, and an intricate automation by canonical structures and lemma overloading [Gonthier et al. 2011; Mahboubi and Tassi 2013].

**Structural lemmas.** The proofs in FCSL are structured to facilitate systematic application of Floyd-style structural rules, one for each program command. All the rules are proved sound from first principles, and are applied as lemmas to advance the verification. As the first step of every proof, the system implicitly applies the weakening rule to the automatically synthesized weakest pre- and strongest postconditions [Dijkstra 1975], essentially converting the program into the continuation-passing style (CPS) representation and sequentializing its structure. Every statement-specific structural rule “symbolically evaluates” the program by one step, and replaces the goal with a new one to be verified.

For example, the following lemma \( \text{step} \), corresponding to the rule of sequential composition, reduces the verification of a program \( (y \leftarrow e_{1};(e_{2} y)) \) with continuation \( k \), to the verification of the program \( e_{1} \) and the program \( e_{2} y k \), where \( y \) corresponds to a symbolic result of evaluating \( e_{1} \), constrained according to \( e_{1} \)’s postcondition. One can apply it several times until \( e_{1} \) is reduced to some primitive action, at which point one can apply the structural rule for that action.

**Lemma** \( \text{step} \) \( W \) \( A \) \( B \) \( (e_{1} : ST W A) \) \( (e_{2} : A \rightarrow ST W B) \) \( (x : \text{cont} B) \):
\[ \text{verify} i e_{1} (\text{fun} y m \Rightarrow \text{verify} m (e_{2} y) k) \Rightarrow \text{verify} \ y (\leftarrow e_{1}; e_{2} y) \ k \]

\( ST \) is a type synonym for \( ST_{\text{step}} \), hiding its pre’s and post’s.

---

3 For simplicity, we present here a specification that is much weaker than what we have actually verified in our implementation.
Table 1: Statistics for implemented programs: lines of code for program-specific libraries (Libs), definitions of concurrent programs and decorations (Conc), actions (Acts), stability lemmas (Stab), spec and proof sizes of the main functions (Main), total LOC count (Total), and build times (Build).

<table>
<thead>
<tr>
<th>Program</th>
<th>Libs</th>
<th>Conc</th>
<th>Acts</th>
<th>Stab</th>
<th>Main</th>
<th>Total</th>
<th>Build</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS-lock</td>
<td>63</td>
<td>291</td>
<td>509</td>
<td>358</td>
<td>27</td>
<td>1248</td>
<td>1m 1s</td>
</tr>
<tr>
<td>Ticketed lock</td>
<td>58</td>
<td>310</td>
<td>706</td>
<td>457</td>
<td>116</td>
<td>1647</td>
<td>2m 46s</td>
</tr>
<tr>
<td>CG increment</td>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>44</td>
<td>70</td>
<td>8s</td>
</tr>
<tr>
<td>CG allocator</td>
<td>82</td>
<td>-</td>
<td>-</td>
<td>192</td>
<td>274</td>
<td>14s</td>
<td></td>
</tr>
<tr>
<td>Pair snapshot</td>
<td>167</td>
<td>233</td>
<td>107</td>
<td>80</td>
<td>51</td>
<td>638</td>
<td>4m 7s</td>
</tr>
<tr>
<td>Treiber stack</td>
<td>56</td>
<td>323</td>
<td>313</td>
<td>133</td>
<td>155</td>
<td>980</td>
<td>2m 41s</td>
</tr>
<tr>
<td>Spanning tree</td>
<td>348</td>
<td>215</td>
<td>162</td>
<td>217</td>
<td>305</td>
<td>1247</td>
<td>1m 11s</td>
</tr>
<tr>
<td>Flat combiner</td>
<td>92</td>
<td>442</td>
<td>672</td>
<td>538</td>
<td>281</td>
<td>2025</td>
<td>10m 55s</td>
</tr>
<tr>
<td>Seq. stack</td>
<td>65</td>
<td>-</td>
<td>-</td>
<td>125</td>
<td>190</td>
<td>1m 21s</td>
<td></td>
</tr>
<tr>
<td>FC-stack</td>
<td>50</td>
<td>-</td>
<td>-</td>
<td>114</td>
<td>164</td>
<td>44s</td>
<td></td>
</tr>
<tr>
<td>Prod/Cons</td>
<td>365</td>
<td>-</td>
<td>-</td>
<td>243</td>
<td>608</td>
<td>2m 43s</td>
<td></td>
</tr>
</tbody>
</table>

6. Evaluation and experience

The Coq proof assistant serves as a tool for implementing FCSL’s metatheory and as a language for writing and verifying concurrent programs. The formalization of the metatheory, which includes the semantic model, structural lemmas and a number of useful libraries (e.g., getters, theory of PCMs, heaps, arrays, etc.), is about 17.2 KLOC size.

We evaluated FCSL by implementing, specifying and verifying a number of characteristic concurrent programs and structures. The simplest fine-grained structure is a lock, and we implemented two different locking protocols: CAS-based spinlock and a ticketed lock [Dinsdale-Young et al. 2010]. Both locks instantiate a uniform abstract lock interface, and are used by coarse-grained programs, performing concurrent incrementation of a pointer and memory allocation. In addition to the spanning tree algorithm and the flat combining construction, we also implemented such fine-grained programs as an atomic pair snapshot [Qadeer et al. 2009; Liang and Feng 2013] and non-blocking stack [Treiber 1986], both given specs via a PCM of time-stamped action histories [Sergey et al. 2015] in the spirit of linearizability [Herlihy and Wing 1990], as well as several client programs: a sequential stack (obtained from Treiber stack via hiding), FC-based stack, and a Treiber stack-based concurrent Producer/Consumer.

Table 1 presents some statistics wrt. implemented programs in terms of LOCs and build times. The program suite was compiled on a 2.7 GHz Intel Core i7 OS X machine with 8 Gb RAM, using Coq 8.4pl4 and Srreflect 1.4. We didn’t rely on any advanced proof automation in the proof scripts, which would, probably, decrease line counts at the expense of increased compilation times. Notably, for those programs that required implementing new primitive concurroids (e.g., locks or Treiber stack), a large fraction of an implementation is due to proofs of properties of transitions and actions, as well as stability-related lemmas, while the sizes of proofs of the main programs’ specs are relatively small.

Our development is inherently compositional, as illustrated by the dependency diagram on Figure 5. For example, both lock implementations are instances of the abstract lock interface, which is used to implement and verify the allocator, which is then employed by a Treiber stack, used as a basis for sequential stack and producer/consumer implementations. In principle, we could implement an abstract interface for stacks, too, to unify the Treiber stack and the FC-stack, although, we didn’t carry out this exercise.

As hinted by Table 1, not every concurrent program requires implementing a new primitive concurroid: typically this is done only for libraries, so library clients can reason out of the specifications. Table 2 shows that the reuse of concurroids is quite high, and most of the programs make consistent use of the concurroid for thread-local state and locks (abstracted through the corresponding interface), as well as of those required by the used libraries (e.g., Treiber or FC).

7. Related and future work

Using the Coq proof assistant as a uniform platform for implementation of logic-based program verification tools is a well-established approach, which by now has been success-
fully employed in a number of projects on certified compilers [Leroy 2006; Appel et al. 2014] and verified low-level code [Shao 2010; Chlipala 2011; Jensen et al. 2013], although, with no specific focus on abstractions for fine-grained concurrency, such as protocols and auxiliary state.

**Related program logics.** The FCSL logic has been designed as a generalization of the classical Concurrent Separation Logic by O’Hearn [2007], combining the ideas of local concurrent protocols with arbitrary interference [Jones 1983; Feng 2009] and compositional auxiliary state [Ley-Wild and Nanevski 2013] with the possibility to compose protocols. Other concurrency logics, close to FCSL in their expressive power, are iCAP [Svendsen and Birkedal 2014], CoLoSL [Raad et al. 2014], and CoReSL [Turon et al. 2013].

iCAP leverages the idea, originated by Jacobs and Piessens [2011], of parametrizing specs for fine-grained concurrent data types by client-provided auxiliary code, which can be seen as a “callback”. A form of composition of concurrent resources can be encoded in iCAP using fractional permissions [Borndat et al. 2005] and view-shifts [Dinsdale-Young et al. 2013]. Since iCAP doesn’t have explicit subjective dichotomy of the auxiliary state, encoding of thread-specific contributions in it is less direct comparing to FCSL.

CoLoSL defines a different notion of thread-local views to a shared resource, and uses overlapping conjunction [Hobor and Villard 2013] to reconcile the permissions and capabilities, residing in the shared state between different threads. Overlapping conjunction affords a description of the shared structure mirroring the recursive calls in the structure’s methods. In FCSL, such machinery isn’t required, as self and other suffice to represent the thread-specific views, and joint state doesn’t need to be divided between threads. In our opinion, this leads to simpler specs and proofs. For example, CoLoSL’s proof that span constructs a tree involves abstractions such as shared capabilities for marking nodes and extension of the graph with virtual edges, none of which is required in FCSL. Moreover, CoLoSL doesn’t prove that the tree is spanning, which we achieve in FCSL via hiding.

CaReSL combines the Hoare-style reasoning and proofs about contextual refinement. Similarly to FCSL, CaReSL employs resource protocols, although, targeting the “life stories” of particular memory locations instead of describing a whole concurrent data structure by means of an STS. While FCSL isn’t equipped with abstractions for contextual refinement, in our experience it was never required to prove the desired Hoare-style specs for fine-grained data structures.

Reasoning in all the three alternative logics follows the tradition of Hoare-style logics, so the specs never mention explicitly the heap and state components. In contrast, FCSL assertions use explicit variables to bind heap and auxiliary state, as well as their components. In our experience, working directly with the state model is pleasant, and has to be done in Coq anyway, since Coq lacks support for contexts of bunched implications, as argued by Nanevski et al. [2010]. None of iCAP, CoLoSL or CaReSL features a mechanized metatheory, nor any of these logics has been implemented in a form of a mechanized verification tool.

**Related tools for concurrency verification.** SAGL and RGSep, the first logics for modular reasoning about fine-grained concurrency [Feng et al. 2007; Vafeiadis and Parkinson 2007], inspired creation of semi- and fully-automated verification tools: SmallfootRG [Calcagno et al. 2007] and Cave [Vafeiadis 2010]. These tools target basic safety properties of first-order code, such as data integrity and absence of memory leaks.

Chalice [Leino and Müller 2009] is an experimental first-order concurrent language, supplied with a tool that generates verification conditions (VCs) for client-annotated Chalice programs. Such VCs are suitable for automatic discharge by SMT solvers. For local reasoning, Chalice employs fractional permissions [Bornat et al. 2005], implicit dynamic frames [Smans et al. 2009], and auxiliary state [Leino et al. 2009], which, unlike the one of FCSL, is not a subject of PCM laws, and thus is not compositional, as its shape should match the forking pattern of the client program being verified. Chalice also supports a form of symmetric Rely/Guarantee reasoning, i.e., not allowing the threads to take different roles in a protocol (which is expressible in FCSL via self-enabled transitions). Chalice’s specification fragment is a first-order logic, whereas FCSL admits higher-order functions and predicates in specs, therefore, enabling program composition and proof reuse, as shown in Figure 5.

VCC [Cohen et al. 2009] is a tool for verifying low-level concurrent C code. VCC doesn’t support reasoning about custom interference or compositional auxiliary state, and, similarly to Chalice, allows specifications only in a first-order logic to support an SMT-based automation back-end.

VeriFast [Jacobs et al. 2011] is a tool for deductive verification of sequential and concurrent programs, based on separation logic [Reynolds 2002]. To specify and verify fine-grained concurrent algorithms, VeriFast employs fractional permissions and a form of (non-compositional) first-class auxiliary state [Jacobs and Piessens 2011]. VeriFast has been recently extended with Rely/Guarantee reasoning [Smans et al. 2014], although, without a possibility to compose resources. To the best of our knowledge, soundness of VeriFast core has been formally proved only partially [Vogels 2012].

Rely-Guarantee references (RGREFs) by Gordon [2014] are a mechanism to prove transition invariants of concurrent data structures. While the language of RGREFs is implemented as a Coq DSL, the system’s soundness is proved by hand using the Views framework [Dinsdale-Young et al. 2013]. Since RGREFs focuses on correctness of data structures wrt. specific protocols and doesn’t provide auxiliary state, it’s unclear how to employ it for client-side reasoning.

**Future work.** In the future, we plan to augment FCSL with the program extraction mechanism [Letouzey 2008] and implement proof automation for stability-related facts via
FCSL doesn’t support higher-order heaps ([Bertot and Castérane 2021]. This is due to the limitations of Coq’s model wrt. impredicativity, at this moment, FCSL doesn’t support higher order heaps (i.e., the possibility to reason about arbitrary storable effectful procedures). While programs requiring this feature are rare, we hope that it will be possible to encode and verify the characteristic ones, once the development is migrated to Coq 8.5, featuring universe polymorphism [Sozeau and Tabareau 2014].

8. Conclusion

Our experience with implementing a number of concurrent data structures in FCSL indicates a recurring pattern, exhibited by the formal proof development. Verification of a new library in FCSL starts from describing its invariants and evolution in terms of an STS. It’s common to consider parts of real or auxiliary state, which are a subject of the logical split between parallel threads, as elements of a particular PCM. Such representation of resources makes the verification uniform and compositional, as it internalizes the library protocol, so the clients can reason out of the specifications.

This observation indicates that STSs and PCMs can be a robust basis for understanding, formalizing and verifying existing fine-grained programs. We conjecture that the same foundational insights will play a role in future designs and proofs of correctness of novel concurrent algorithms.

References


