# 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 FCSL. 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*, FCSL 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 FCSL by example, and report on our experience of using FCSL to verify a number of concurrent programs.

***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

***General Terms*** Algorithms, Theory, Verification

***Keywords*** Compositional program verification, concurrency, separation logic, mechanized proofs, dependent types.

## 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 to the 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) **Helping.** This 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 "helping" happens when a thread is scheduled for a task involving the resource, but the task is then accomplished by *another* thread; however, the result of the work, once the task is completed, is 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 asser-

*PLDI'15*, June 13–17, 2015, Portland, OR, USA.
Copyright © 2015 ACM 978-1-4503-3468-6/15/06. . . $15.00.
http://dx.doi.org/10.1145/2737924.2737964

tions to global invariants, accounting for specific threads' contributions into a resource [27].

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 [11, 14–16, 23, 28, 37, 44, 51, 53, 55], 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 helping. 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 is too easy for a human prover to forget about a piece of resource-specific invariant or to miss an intermediate 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 [6, 39, 40, 48], or implementing a standalone verification tool [8, 24, 29]. 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.* [37].[1] FCSL is a library and an embedded domain-specific language (DSL) in the dependently-typed language of Coq Proof Assistant [10]. 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 helping 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) [2]. 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 [38] 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 functional languages, such as 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 some libraries might be large, they are done just once.

```
1   span (x : ptr) : bool {
2     if $x$ == null then return false;
3     else
4       $b \leftarrow$ CAS($x$->$m$, 0, 1);
5       if $b$ then
6         $(r_l, r_r) \leftarrow$ (span($x$->$l$) || span($x$->$r$));
7         if $\neg r_l$ then $x$->$l$ := null;
8         if $\neg r_r$ then $x$->$r$ := null;
9         return true;
10      else return false; }
```

**Figure 1:** Concurrent spanning tree construction procedure.

- **Interactive:** FCSL benefits from the infrastructure, provided by Coq's fragment for mechanized reasoning, enhanced by Ssreflect extension [17]. 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 *wrt.* a version of denotational semantics for concurrent programs in the spirit of Brookes [4]. Moreover, since FCSL program specs are encoded as Coq types, the soundness result scales to the *entire programming language* of Coq, not just a toy core calculus. This ensures the absence of bugs in the whole verification tool and, as a consequence, correctness of any program, which is verified in it.

Our implementation of FCSL, unanimously accepted by PLDI'15 AEC, is available online with examples and the user manual [46].

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 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 atomic snapshot structure and non-blocking universal constructions. We conclude with a comparison to related logical frameworks and tools, and a discussion on future work.

## 2. Overview

We introduce verification in FCSL by informally explaining a fine-grained concurrent program for computing in place a spanning tree of a binary directed graph [22, 44].

### 2.1 A Graph Spanning Tree Algorithm and its Intuition

The recursive procedure `span` (Figure 1) takes an argument $x$, which is a pointer to a node in the graph, and constructs the spanning tree rooted in $x$ by traversing the graph and removing redundant edges. The graph is implemented as a memory region where each pointer's target value is a triple. The triple's first component is a bit $m$ indicating whether the node is *marked*; the second and third components are pointers to 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

[1] Hereafter, we will be using the acronym FCSL to refer both to the Nanevski *et al.*'s logical framework and to our implementation of it.

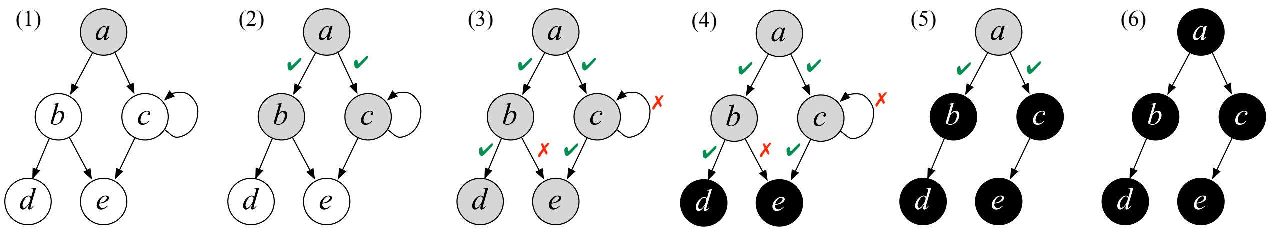

**Figure 2:** Stages of concurrent spanning tree construction. A node is painted grey right after a corresponding thread successfully marks it (line 4 of Figure 1). It is painted black right before the thread returns `true` (line 9). A black subtree is *logically ascribed* to a thread that marked its 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 $e$; (4) the processing of $d$ and $e$ is done; (5) the redundant edges $b \to e$ and $c \to c$ are removed by the corresponding parent threads; (6) the initial thread joins its children and terminates.

(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 2 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 *final* graph's topology (*i.e.*, the tree cannot be extended with additional nodes). Lines 7 and 8 of `span` preserve the edges from $x$ to $x.l$ and $x.r$; thus $x$ becomes a root of a tree with subtrees pointed to by $x.l$ and $x.r$ (Figure 2(6)). If $r_l =$ `true` and $r_r =$ `false`, the recursive call has computed a tree from the subgraph rooted at $x.l$, but $x.r$ has been found marked. The edge to $x.r$ is removed in line 8, but that to $x.l$ is preserved in line 7; $x$ becomes a root of a tree with left subtree rooted in $x.l$, and right subtree empty (Figure 2(5)). The case $r_l =$ `false`, $r_r =$ `true` is dual. The case $r_l = r_r =$ `false` results in a subtree containing just the root $x$ (Figure 2(4)).

Why does `span` construct a *spanning* tree? Consider the *front* of the constructed tree in the initial graph (*i.e.*, the nodes immediately reachable in the initial graph, from the nodes of the constructed tree. For example, in Figure 2(5), the front of $b$ are nodes $d, e$ in Figure 2(1)). We will show in the next section that this front must contain only marked nodes, as otherwise `span` would have proceeded with node marking, instead of terminating. Thus the front of the tree constructed by the top-level call to `span` must be included in the very same tree. Otherwise, there exists a node which is marked but not in the tree. Therefore, this node must have been marked by another thread, thus contradicting assumption (2). Since, by (1), the initial graph is connected, a tree which contains its front must contain all the nodes, and thus be a spanning one.

## 2.2 Infrastructure for Mechanizing the Proof in FCSL

To flesh out the above informal argument and mechanize it in FCSL, we require a number of logical concepts. We describe these next, and tie them to the `span` algorithm.

### 2.2.1 Concurroids

FCSL requires an explicit description of the common assumptions that threads operating on the shared resource observe. Such agreement between threads is needed so that one thread's changes match other threads' expectations. In the case of `span`, for example, one assumption we elided above is that an edge out of a node $x$ can be removed only by a thread that marked $x$. This thread can rely on the property that edges to $x.l$ and $x.r$ won't be nullified by another thread. These assumptions are formalized by STSs of a special form, which are called *concurroids*.

The state of each concurroid is divided into three components $[\mathit{self} \mid \mathit{joint} \mid \mathit{other}]$. 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 [33]. We also note that the *self* and *other* components have to be elements of a PCM, *i.e.*, a set $\mathbb{U}$ with an associative and commutative *join* operation $\bullet$, and a unit element $\mathbb{1}$. PCMs enable general and compositional representation of thread-owned state (auxiliary or real), which can be split between parallel threads [33]. Intuitively, commutativity and associativity account for those of a parallel composition of threads, and partiality captures the fact that not all splits/combinations of state are valid.

All three state components of a concurroid may contain real state, *i.e.* heap, or *auxiliary state* [35, 42], which is kept for logical specification, but is erased before execution. In the case of the `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 $\uplus$ 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 the threads are allowed to do. The concurroid for `span`, named `SpanTree` in the sequel, has two non-trivial transitions, which we call `marknode_trans` and `nullify_trans`. Additionally, every concurroid has the trivial identity transition `idle`. A thread performs `marknode_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 `nullify_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 previously marked $x$ can take this transition.

```
Program Definition span : span_tp :=
 ffix (fun (loop : span_tp) (x : ptr) =>
  Do (if x == null then ret false else
      b <-- trymark x;
      if b then
        xl <-- read_child x Left;
        xr <-- read_child x Right;
        rs <-- par (loop xl) (loop xr);
        (if ~~rs.1 then nullify x Left else ret tt);;
        (if ~~rs.2 then nullify x Right else ret tt);;
        ret true
      else ret false)).
```

**Figure 3:** FCSL implementation of the span procedure.

### 2.2.2 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.*, CAS. 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 span in FCSL.

The 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 CAS on line 4 of Figure 1. Logically, if successful, it corresponds to a marknode_trans transition in the concurroid. If unsuccessful, it corresponds to the concurroid's idle transition. The nullify action invoked with an argument $x$, and a two-valued indicator side (Left or Right), sets the $x.l$ (or $x.r$, depending on side) pointer to null, but emits a precondition that $x$ is in *self.* Operationally, it corresponds to the assignment of null on lines 7 and 8 of Figure 1. Logically, it corresponds to taking the nullify_trans transition. Finally, read_child atomic action, invoked with arguments $x$ and side, returns the pointer $x.l$ (or $x.r$, depending on 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 3 shows how the actions are used to translate the span procedure from Figure 1 into FCSL.

### 2.2.3 Hoare Specifications as Types and Stability

In Figure 3, span is ascribed the type span_tp. While Section 3 defines it formally, here we provide some basic intuition for it.

Among other components, the type span_tp contains the formal pre- and postconditions, ascribed to span. Hence, it is a user-defined type, rather than inferred by the system. Also, span_tp is declared as the type of the fixpoint combinator ffix's argument loop, and thus serves as the "loop invariant" as well. The components of span_tp provide the following information: (a) The precondition in span_tp ensures that the input node $x$ is either null or points to a node in the heap. (b) If span returns false, the postcondition ensures that $x$ is either null or is marked in the graph, and the thread hasn't marked any other nodes during the call. (c) If span returns true, the postcondition states that $x \neq$ 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 *stable wrt.* interference, *i.e.*, they remain valid no matter which transitions of the span concurroid the interfering threads take. Proving stability is an important component of FCSL. Typically, every spec used in FCSL will be 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 span, including span_tp.

```
Definition span_tp := forall (x : ptr),
 {i (g1 : graph (joint i))}, STsep [SpanTree sp]
   (* precondition predicate *)
   (fun s1 => i = s1 /\
             (x == null \/ x \in dom (joint s1)),
   (* postcondition predicate *)
    fun (r : bool) s2 => exists g2 : graph (joint s2),
     subgraph g1 g2 /\
     if r then x != null /\ exists t,
       self s2 = self i \+ t /\ tree g2 x t /\
       maximal g2 t /\ front g1 t (self s2 \+ other s2)
     else (x == null \/ mark g2 x) /\ self s2 = self i).
```

**Figure 4:** Specification span_tp of the span procedure.

### 2.2.4 Hiding from External Interference

The type span_tp specifies the calls to span in the loop, but the top-most call to 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 span actually constructs a *spanning* tree, so we need to enforce it.

The encapsulation of interference is achieved in FCSL by the program constructor hide. For instance, writing $\mathsf{hide}_{\Phi,\emptyset}\{\,\mathtt{span}(x)\,\}$, makes it apparent to the type system of FCSL that the top-most call to span runs without interference on the shared graph. More precisely, the call, span(x), within hide will execute relative to the protocol implemented by the 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 span protocol. The surrounding threads thus cannot interfere with the inside call to 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 span's caller onto which the span concurroid should be installed. In the case of span, $\Phi$ merely describes the nodes of the graph we want to span. $\emptyset$ indicates that span is initially invoked with the empty auxiliary state, *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 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 [46].

### 3.1 The Definition of the Type span_tp

The type span_tp is described in Figure 4. It is an example of a *dependent type*, as it takes formal arguments in the form of variables x, i and g1, that the body of the type can use, *i.e.*, *depend on*. The roles of the variables differ depending on the keyword that binds them. For example, the Coq keyword forall binds the variable x of type ptr, and indicates that span_tp is a specification for a procedure that has x as input. Indeed, span is exactly such a procedure, as apparent from Section 2. Using forall to bind x allows x to be used in the body of span_tp, *but also in the body* of span (Figure 3). On the other hand, i and g1 are bound by FCSL binder {...}. This binding is different; it allows i and g1 to be used in the body of span_tp, *but not in the procedure* span. In terminology of Hoare-style logic, i and g1 are *logical variables* (aka. *ghosts*), which are used in specs, but not in the code. 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 SpanTree sp in the brackets is the concurroid whose protocol span_tp respects. We will define SpanTree shortly. Finally, the parentheses include the precondition and the postcondition (defined as Coq's functions) that we want to ascribe to span. The precondition is a predicate over the pre-state s1. The postcondition is a predicate over the boolean result r and post-state s2. As customary in many programming languages, Coq included, we omit the types of various variables when the system can infer them (*e.g.*, the variables i, s1 and s2 are all of type state).

The precondition says that the input x is either null (since 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 joint out of the input state s1, which i snapshots. The projections self and other are sets of marked nodes, belonging to the caller of span and to its environment, respectively.

The postcondition says that in the case the return result is r = false, the pointer x was either null or already marked. Otherwise, there is a set of nodes t which is freshly marked by the call to span; that is, self s2 is a *disjoint union* (\+) of t with the set of nodes marked in the pre-state self i. The set t satisfies several important properties. First, t is a subtree in the graph, g2, of the post-state s2, with root x. Second, the tree t is *maximal*, *i.e.*, it cannot be extended into a larger tree by adding more nodes from g2, as all the edges between t and the rest of the graph have been severed by span. Third, all the nodes immediately reachable from t in the *initial* state i (*i.e.*, t's front) are marked in g2 either by this or some other thread (self s2 \+ other s2). That is, span did not leave any reachable nodes unmarked; if such nodes existed, span would not have terminated. Finally, in both cases, subgraph g1 g2 states that the final graph g2 is obtained by marking nodes and removing edges from the initial graph g1; no new edges are added, no nodes are un-marked.

We close the description of span_tp by noting its interesting *bi-directional* nature. It contains properties such as tree and maximal, stated over the post-state graph g2 (forward direction), but also the property front which is stated of the pre-state graph g1, and can be stated only in relation to s2 (backward direction). The backward direction is a crucial component in the proof that the top-most call to span, shielded from interference by hide, indeed marked all the nodes and, hence, constructed a spanning tree.

## 3.2 Representing Graphs in a Heap

Next we define the predicate graph h, which appears in span_tp (Figure 4), and says when a heap h represents a graph. It does so if every pointer x in h stores some triple (b, xl, xr), where b is the "marked" bit, and xl, xr are pointers in the domain of h (and, hence, are x itself or other nodes), or null if x has no successors.

```
Definition graph (h : heap) := valid h /\
  forall x, x \in dom h -> exists (b : bool) (xl xr : ptr),
      h = x :-> (b, xl, xr) \+ free x h /\
      {subset [:: xl; xr] <= [:: null] ++ dom h}.
```

The conjunct valid h says that the heap h doesn't contain duplicate pointers. The notation \+ is overloaded and used for disjoint union of sets of nodes in span_tp, and for disjoint union of heaps in graph. In general, we use \+ for any PCM • operation. free x h is the heap obtained by deallocating x from h. Finally, the last line is concrete syntax for $\{\texttt{xl}, \texttt{xv}\} \subseteq \{\texttt{null}\} \cup \mathrm{dom}\,\texttt{h}$.

The graph predicate illustrates *certified programming* in Coq [7], *i.e.*, the ability to use propositions as types, and pass variables such as g1 and g2 that stand for *proofs* of the graph property, as inputs to other types (*e.g.*, span_tp) or functions. This ability enables formally defining *partial* functions over heaps that are undefined when the heap doesn't encode a valid graph. An alternative to this somewhat unique capability of dependent types is to encode partial functions as relations, but that usually results in increase in proof tedium and size.

Here are a few examples of such partial functions. Given a node (*i.e.*, a pointer) x and a proof that the heap h represents a graph (written (g : graph h)), we name mark g x, edgl g x and edgr g x the three components stored in the pointer x in the heap (*i.e.*, the "marked" bit, left, right successor), and write cont g x for the whole triple. By default, these values are false, null, null if x is not in the heap. Each of these functions takes h as an argument; *i.e.*, one could also write mark h g x *etc*., but we omit h as it can be inferred from g's type, following Coq's standard notational abbreviation.

We can now define the remaining predicates used in span_tp in Figure 4. For all of the definitions, we assume that variables h and (g : graph h) are in scope, and omit them. We also use ptr_set as an alias for finite maps from pointers to the unit type.[2]

First, we define the function edge, which represents the incidence relation for the graph g.

```
Definition edge (x y : ptr) := (x \in dom h) &&
  (y != null) && (y \in [:: edgl g x; edgr g x]).
```

Second, tree x t requires that t contains x, and for any node $y \in t$, there exists a unique path (*i.e.*, a list of nodes) p from x to y via edge's links, which lies within the tree (*i.e.*, the nodes p are a subset of t). Note how edge is *curried*, *i.e.*, passed to path as a function, abstracted over arguments. This illustrates that even simple mathematical mechanizations require higher-order functions in order to work.

```
Definition tree (x : ptr) (t : ptr_set) := x \in dom t /\
  forall y, y \in dom t -> exists !p,
    path edge x p /\ y = last x p /\ {subset p <= dom t}.
```

Third, front t t', determines if the nodes reachable from t in *zero or one step* are included in t'.

```
Definition front (t t' : ptr_set) :=
  {subset dom t <= dom t'} /\
  forall x y, x \in dom t -> edge x y -> y \in dom t'.
```

Fourth, a tree t is maximal if it includes its front. A graph is connected if there's a path from x to every other node y in it.

```
Definition maximal (t : ptr_set) := front t t.
Definition connected (x : ptr) (t : ptr_set) := forall y,
  y \in dom t -> exists p, path edge x p /\ last x p = y.
```

Finally, subgraph codifies a number of properties between pre-state s1 and post-state s2, and their graphs g1, g2. In particular: g1, g2 contain the same nodes (=i is equality on lists modulo permutation), the set of self-marked and other-marked nodes only increase, edges out of a node y can be changed only if the node is marked, and the only change to the edges is nullification (that is, removal).

```
Definition subgraph s1 s2
  (g1 : graph (joint s1)) (g2 : graph (joint s2)) :=
  dom (joint s1) =i dom (joint s2) /\
  {subset dom (self s1) <= dom (self s2)} /\
  {subset dom (other s1) <= dom (other s2)} /\
  (forall y, ~~(mark g2 y) -> cont g1 y = cont g2 y) /\
  (forall x, (edgl g2 x \in [:: null; edgl g1 x]) /\
          (edgr g2 x \in [:: null; edgr g1 x])).
```

[2] This is a bit expedient way of implementing finite sets, but it saves work by reusing an extensive library of finite maps, also used for formalizing heaps.

We close the description of the predicates used in `span_tp`, by listing two important lemmas that relate them. The first lemma, `max_tree2`, says that if `y1` and `y2` are successors of `x` (*i.e.*, `edge x` equals the set `[:: y1; y2]` modulo permutation), and `ty1` and `ty2` are maximal trees rooted in `y1` and `y2`, and moreover, `ty1` and `ty2` are disjoint, then the set of nodes built from `x`, `ty1` and `ty2` by disjoint union (`\+`) is a tree itself, *i.e.*, no edges connect `ty1` and `ty2` (the notation `#x` is concrete syntax for the singleton finite map containing node `x`). This lemma is essential in proving that `span` produces a tree, as mentioned in Section 2 for the case $r_l = r_r =$ `true`.

```
Lemma max_tree2 x y1 y2 ty1 ty2 :
  edge x =i [:: y1; y2] -> tree y1 ty1 -> maximal ty1 ->
  tree y2 ty2 -> maximal ty2 -> valid (ty1 \+ ty2) ->
  tree x (#x \+ ty1 \+ ty2).
```

The second lemma shows that `subgraph` is monotone *wrt.* the stepping of environment threads in the `SpanTree` concurroid.

```
Lemma subgraph_steps s1 s2
  (g1 : graph (joint s1)) (g2 : graph (joint s2)) :
  env_steps (SpanTree sp) s1 s2 -> subgraph g1 g2.
```

We used this lemma as the main tool in establishing a number of stability properties in Coq, related to the conjuncts 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 changes via environment interference.

### 3.3 `SpanTree` Concurroid

Next we define the `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: *labels* (semantically, natural numbers) that differentiate instances of the concurroid. Thus the definition of `SpanTree` is parametrized by the variable `sp`, which makes it possible to use several instances of `SpanTree` with different labels in a specification of a single program. For example, say we want to run two `span` procedures in parallel on disjoint heaps. Such a program could be specified by a Cartesian product of `SpanTree sp1` and `SpanTree sp2`, where the different labels `sp1` and `sp2` instantiate the variable `sp`.

The state space of `SpanTree` is defined by the following state predicate `coh`, which we call *coherence predicate*.

```
Variable sp : nat.
Definition coh s := exists g : graph (joint s),
  s = sp ->> [self s, joint s, other s] /\
  valid (self s \+ other s) /\
  forall x, x \in dom (self s \+ 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 \+ other s)` says that the *self* and *other* components of the auxiliary state are disjoint; their union is a finite map which is `valid`, *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 *iff* it's marked in the joint graph.

The metatheory of FCSL [37, §4] requires the coherence predicates to satisfy several properties that we omit here, but prove in our implementation. The most important property is the *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.

`SpanTree sp` contains two non-idle transitions. The first transition `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 second transition `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 `mark_node` and `null_edge` that describe the physical changes performed by the two transitions to the underlying shared graph. These can be found in the accompanying Coq code [46].

```
Definition marknode_trans x s s' :=
 exists g : graph (joint s), ~~(mark g x) /\
   joint s' = mark_node g x /\ self s' = #x \+ self s /\
   other s' = other s /\ coh s /\ coh s'.

Definition nullify_trans x (c : side) s s' :=
 exists g : graph (joint s), x \in dom (self s) /\
   joint s' = null_edge g c x /\ self s' = self s /\
   other s' = other s, coh s /\ coh s'.
```

The FCSL metatheory requires that transitions also satisfy several properties. For example, `marknode_trans` and `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, *i.e.*, they don't add or remove any pointers. Adding and removing heap parts can be accomplished by *communication* between concurroids, which we will briefly discuss in Section 4 of this paper.

The coherence predicate, the transitions, and the proofs of their properties are packaged into a *dependent record*[3] `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.

### 3.4 Defining 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 *erasure property* that says that the effect of the action on the auxiliary state doesn't affect the real state.

Specifically, the effect of the `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 `bool`.

```
Definition trymark_step (x : ptr) s1 s2 (r : bool) :=
 exists g : graph (joint s1),
   x \in dom (joint s1) /\ other s2 = other s1 /\
   if mark g x
   then r = false /\ joint s2 = joint s1 /\
        self s2 = self s1
   else r = true /\ joint s2 = mark_node g x /\
        self s2 = #x \+ 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 `false`. Notice that when restricted to the real heap, *i.e.*, if we ignore the auxiliary state in `self s1` and `other s1`, the relation essentially describes the effect of the `CAS` command on the mark bit of `x`. Thus, `trymark` *erases* to `CAS`.

[3] A type-theoretic variant of a C `struct`, where fields can contain proofs.

There are several other components that go into the definition of an atomic action. In particular, one has to prove that transitions are *total*, *local*, and *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 `idle` if it fails. Actions may also change state of a number of concurroids simultaneously, as we will discuss in Section 4. We omit the formal definition of all these properties, but they can be found in the accompanying code [46].

### 3.5 Scoped Concurroid Allocation and Hiding

The `span_tp` type from Figure 4 operates under *open-world assumption* that `span` runs in an environment of interfering threads, which, however, respect the transitions of the `SpanTree` concurroid. If one wants to protect `span` from interference, and move to *closed-world assumption*, the top-most call must be enclosed within 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. The thread-local state is modelled in FCSL by a basic concurroid `Priv pv` with a label `pv` [37, §4], and its *self*/*other* components are retrieved via `pv_self` and `pv_other` projections. The description of how much local heap should be "donated" to the concurroid creation is provided by the user-supplied predicate $\Phi$, called *decoration* predicate. In addition to the heap, the predicate scopes over the auxiliary *self* value, while the auxiliary *other* is fixed to the PCM unit, to signal that there's no interference from outside threads. In the case of `span`, the decoration predicate is the following one.

```
Definition graph_dec sp (g : heap * ptr_set) s := coh s /\
  exists (pf : graph g.1), s = sp ->> [g.2, g.1, Unit].
```

We can now write out a new type `span_root_tp`, to specify the top-most call to `span`, under the closed-world assumption that there's no interference. Parametrizing *wrt.* the locally-scoped variable `h1 : heap` that snapshots the initial heap, the type is as follows.

```
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 _ 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).
```

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

```
Program Definition span_root x : span_root_tp x :=
 Do (priv_hide pv (graph_dec sp) (h1, Unit) [span sp x]).
```

Coq will emit a proof obligation that the pre and post of `span_tp` can be weakened into those of `span_root_tp` under the closed-world assumption that `other s2 = Unit`. This proof is in the development, accompanying this paper [46].

## 4. 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 helping.

### 4.1 Composing Concurrent Resources

The `span` algorithm uses only one concurroid `SpanTree`, allocated by hide out of the concurroid `Priv` for thread-local state. In general, FCSL specs can span multiple primitive concurroids, of the same or different kinds, which are *entangled* by interconnecting special *channel*-like transitions [37]. 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.

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

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

Whereas separation logic [45] 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 `STsep` type that automatically entangles the underlying concurroid with `ALock`, thus enabling allocation without the user having to explicitly do so herself.

### 4.2 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.* [20].[4]

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 `flat_combine(f, v)` looks like a sequence `lock; f(v); 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 *helping* or *work stealing*: 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_inT f`, result of type `fc_outT f` and the *contribution* of type `fc_pcm`. The

[4] For simplicity, we present here a specification that is much weaker than what we have actually verified in our implementation.

last entry is a description of what `f` *does* to the shared state, expressed in abstract algebraic terms as an element from a user-supplied PCM `fc_pcm`.

```
Variable fc_R : forall f,
         fc_inT f -> fc_outT f -> fc_pcm -> Prop.
```

The spec of the `flat_combine` is then given in the context of three entangled concurroids: `Priv` for thread-local state, a lock-based allocator `Alloc`, adapted from the previous example (since a sequential function `f` might allocate new memory), and a separate concurroid `FlatCombine`.

```
Definition PA := (entangle (Priv pv) Alloc).
Program Definition flat_combine f (v : fc_inT f) :
 STsep [entangle PA (FlatCombine fc)]
 (fun s1 => pv_self s1 = Unit /\ fc_self s1 = Unit,
  fun (w : fc_outT f) s2 => exists g, pv_self s2 = Unit /\
      fc_self s2 = g /\ fc_R f v w g) := ...
```

The precondition says that `flat_combine` executes in the empty initial heap (`pv_self s1 = 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_self s1 = Unit`), but can be made arbitrary by applying FCSL's *frame rule* to the spec of `flat_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_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 the ideas behind implementation of FCSL as an embedding into Coq, and describe important parts of FCSL machinery, used to simplify construction of proofs.

### 5.1 FCSL Model and Embedding into Coq

Programs in FCSL are encoded as their values in the denotational semantics of sets of *action trees* [37, Appendix F of extended version]. The trees are a structured version of *action traces* by Brookes [4]. They implement finite, partial approximations of the behavior of FCSL commands. Basic commands, such as `ret` and atomic actions are given semantics as finite sets of primitive trees. The semantics of program combinators, such as sequential or parallel composition, is defined as a set-comprehension over the elements of their constituents' semantics, and `ffix` combinator is defined by taking Tarski's fixed point of its argument function's monotone closure over the powerset lattice. Therefore, each FCSL program (*e.g.*, `span` in Figure 3) is a value in Coq, obtained by composing primitive commands and atomic actions via FCSL combinators and native Coq expressions (*e.g.*, functions and conditionals).

In our implementation, the definition of each FCSL command (*i.e.*, its denotational semantics) is packaged, as a dependent record, together with a specification, corresponding to its *weakest pre-* and *strongest postconditions* [12] *wrt.* the natural safety predicate,[5] and with a proof that such specification is a valid one for the corresponding program's semantics. We also defined a number of implicit coercions that inject a command into the corresponding record instance, containing the command's spec. This allowed us to exploit the type-checking machinery of Coq, which is capable of composing coercions, thus *synthesizing* weakest pre- and strongest postconditions for arbitrarily complex well-typed FCSL programs.

[5] The predicate states that a program can execute (possibly, concurrently) in a state, satisfying a given precondition, without crashing [37, §F.5].

### 5.2 Verification Conditions and Structural Lemmas

Even though each well-typed FCSL program has already the strongest specification inferred automatically by the system, such spec isn't what one would typically like to ascribe to a program and expose to the clients. Therefore, verification in FCSL reduces to *weakening* the derived strongest spec to the one of interest.

The verification conditions for weakening are encoded by wrapping a program into the `Do` constructor (*e.g.*, in Figure 3), which emits a corresponding proof obligation. Such proofs in FCSL are structured to facilitate systematic application of Floyd-style structural rules, one for each 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, 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 (or several ones) to be verified.

For example, the following lemma `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 step W A B (e1 : ST W A)
                 (e2 : A -> ST W B) i (k : cont B):
       verify i e1 (fun y m => verify m (e2 y) k) ->
       verify i (y <-- e1; e2 y) k.
```

`ST` is a type synonym for `STsep`, hiding its pre's and post's.

### 5.3 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 naïve way of doing this is to describe the state space of the composition concurroid using existentials that abstract over the concurroid-specific fields. For instance, in the case of flat combiner, which composes three concurroids `Priv`, `Alloc` and `FlatCombine`, we could use three existentials to abstract over *self*/*joint*/*other* fields for `Priv`, another three for `Alloc`, and three more for `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 s` to obtain the *self* component of `s`, associated with a concurroid whose label is `pv`. The identifier `pv_self` we used in the spec for `span_root` and for `flat_combine` is a notational abbreviation for exactly this projection. While this is an obvious idea, its execution required a somewhat involved use of dependently-typed programming, and an intricate automation by employing canonical structures and lemma overloading [18, 36].

## 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, implementation of getters, structural lemmas and a number of useful libraries (*e.g.*, theory of PCMs, heaps, arrays, *etc.*), is about 17.2 KLOC size.

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

**Table 1:** Statistics for implemented programs: lines of code for program-specific libraries (Libs), definitions of concurroids and decorations (Conc), actions (Acts), stability-related lemmas (Stab), spec and proof sizes of the main functions (Main), total LOC count (**Total**), and build times (**Build**). The "-" entries indicate the components that were not needed for the example.

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 [14]. 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 [34, 43] and non-blocking stack [52], both given specs via a PCM of time-stamped action histories [47] in the spirit of linearizability [21], 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.

The PCMs employed in formalizations of the case studies are: disjoint sets [37] (spanning tree, FC, ticketed lock), heaps [33] (thread-local state), natural numbers with addition and zero [33] (CG increment), mutual exclusion PCM [33, 37] (CAS-lock, FC), time-stamped histories [47] (pair snapshot, Treiber stack, producer/consumer), client-provided PCMs [33, 37] (FC, locks), *lifted* PCMs—products of basic PCMs [33, 37] (FC, locks). All these PCM instances are treated uniformly in the proofs conducted in FCSL, due to the unifying algebraic structure.

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 Ssreflect 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 always 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,

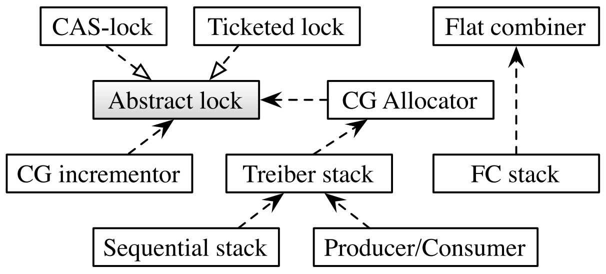

**Figure 5:** Dependencies between concurrent libraries.

| **Program** | Priv | CLock | TLock | ReadPair | Treiber | SpanTree | FlatCombine |
|---|---|---|---|---|---|---|---|
| CAS-lock | ✓ | ✓ | | | | | |
| Ticketed lock | ✓ | | ✓ | | | | |
| CG increment | ✓ | ✓$_L$ | ✓$_L$ | | | | |
| CG allocator | ✓ | ✓$_L$ | ✓$_L$ | | | | |
| Pair snapshot | | | | ✓ | | | |
| Treiber stack | ✓ | ✓$_L$ | ✓$_L$ | | ✓ | | |
| Spanning tree | ✓ | | | | | ✓ | |
| Flat combiner | ✓ | ✓$_L$ | ✓$_L$ | | | | ✓ |
| Seq. stack | ✓ | ✓$_L$ | ✓$_L$ | | ✓ | | |
| FC-stack | ✓ | ✓$_L$ | ✓$_L$ | | | | ✓ |
| Prod/Cons | ✓ | ✓$_L$ | ✓$_L$ | | ✓ | | |

**Table 2:** Primitive concurroids (in column headings) employed by different programs. Two lock concurroids, for CAS-based and ticketed locks, are interchangeable, as they implement the same abstract interface (indicated by ✓$_L$).

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 stack or Flat combiner).

## 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 successfully employed in a number of projects on certified compilers [1, 31] and verified low-level code [6, 25, 48], 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 [41], combining the ideas of local concurrent protocols with arbitrary interference [15, 26] and compositional (*i.e.*, subjective) auxiliary state [33] with the possibility to compose protocols. Other modern concurrency logics, close to FCSL in their expressive power, are iCAP [51], Iris [28], CoLoSL [44], and CaReSL [53].

Both iCAP and Iris leverage the idea, originated by Jacobs and Piessens [23], of parametrizing specs for fine-grained concurrent data types by client-provided auxiliary code, which can be seen as a "callback", and nowadays is usually referred to as a *view shift*. View shifts also serve purposes similar to FCSL's concurroid transitions. A form of composition of concurrent resources can be encoded by combining view shifts with fractional permissions [3] (in iCAP) or invariant masks (in Iris). Similarly to FCSL, Iris makes use of PCMs as a generic mechanism to describe state, which can be split between parallel threads. However, neither iCAP, nor Iris have explicit subjective dichotomy of the auxiliary state, which makes encoding of thread-specific contributions (*e.g.*, marked nodes from the graph example) in them less direct comparing to FCSL.

CoLoSL defines a different notion of thread-local views to a shared resource, and uses *overlapping conjunction* [22] to reconcile the permissions and capabilities, residing in the shared state between parallel 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. In our opinion, this leads to simpler specs and proofs. For example, the proof that `span` constructs a *tree* in CoLoSL 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 to specify thread interference, although, targeting the "life stories" of particular memory locations instead of describing a whole concurrent data structure by means of an STS. While FCSL is not 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.

Neither of the four mentioned logics features hiding as a lanugage constructor for controlling interference. Reasoning in iCAP, Iris, CoLoSL and CaReSL 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.* [40]. None of iCAP, CoLoSL or CaReSL features a mechanized metatheory, and neither of these logics has been implemented as a mechanized verification tool. The soundness of Iris' primitive proof rules has been mechanized in Coq, however, the logic itself has only been employed for paper-and-pencil verification of example programs.

***Related tools for concurrency verification*** SAGL and RGSep, the first logics for modular reasoning about fine-grained concurrency [16, 55], inspired creation of semi- and fully-automated verification tools: SmallfootRG [5] and Cave [54]. These tools target basic safety properties of first-order code, such as data integrity and absence of memory leaks.

Chalice [29] 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 [3], implicit dynamic frames, and auxiliary state [30], 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.*, it does not allow the threads to take different roles in a protocol (which is expressible in FCSL via *self*-enabled transitions, such as `nullify_trans` from Section 3.3). 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 [8] is a tool for verifying low-level concurrent C code. VCC relies on an object-based model to describe protocols and decomposition of state, and its two-state object invariants are similar to our resource protocols. However precise comparison is difficult, as FCSL is not bound to an object-based model and allows quantification over arbitrary state. For compositional reasoning, VCC employs a number of abstractions, such as *claims*, *ghost* and *volatile* fields, *ghost*, *owned*, *wrapped*, *nested* and *closed* objects. Because FCSL relies on standard type theory and PCMs, specialized abstractions and annotations are not required. VCC allows specifications only in a first-order logic to support automation for an SMT-based back-end. To the best of our knowledge, soundness of full VCC has not been proved formally [9].

VeriFast [24] is a tool for deductive verification of sequential and concurrent C and Java programs, based on separation logic [45]. To specify and verify fine-grained concurrent algorithms, VeriFast employs fractional permissions [3] and a form of first-class auxiliary code [23], enabling manipulation of (non-compositional) auxiliary state. VeriFast has been recently extended with Rely/Guarantee reasoning [49], although, without a possibility to compose resources. To the best of our knowledge, soundness of VeriFast core has been formally proved only partially [56].

Rely-Guarantee references (RGREFs) by Gordon [19] are a mechanism to prove transition invariants of concurrent data structures. While the language of RGREFs is implemented as an axiomatized Coq DSL, the system's soundness is proved by hand using the Views framework [13]. 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 [32] and implement proof automation for stability-related facts via lemma overloading [18]. 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 [50].

## 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.

***Acknowledgements*** We thank the anonymous reviewers from PLDI'15 PC and AEC for their feedback. We are also grateful to Steve Blackburn for his efforts as PLDI PC chair. This research is partially supported by Ramon y Cajal grant RYC-2010-0743 and the US National Science Foundation (NSF). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

## References

[1] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.

[4] S. Brookes. A semantics for concurrent separation logic. *Th. Comp. Sci.*, 375(1-3), 2007.

[5] C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, volume 4634 of *LNCS*, pages 233–248. Springer, 2007.

[6] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245. ACM, 2011.

[7] A. Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2013. Available from `http://adam.chlipala.net/cpdt`.

[8] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[9] E. Cohen, W. J. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *SOFSEM*, volume 7741 of *LNCS*, pages 1–27. Springer, 2013.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version V8.4*, 2014. `http://coq.inria.fr/`.

[11] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231. Springer, 2014.

[12] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.

[13] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300. ACM, 2013.

[14] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

[15] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327. ACM, 2009.

[16] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, volume 4421 of *LNCS*, pages 173–188. Springer, 2007.

[17] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Technical Report 6455, Microsoft Research – Inria Joint Centre, 2009.

[18] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175. ACM, 2011.

[19] C. S. Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

[20] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.

[21] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

[22] A. Hobor and J. Villard. The ramifications of sharing in data structures. In *POPL*, pages 523–536. ACM, 2013.

[23] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282. ACM, 2011.

[24] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[25] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *POPL*, pages 301–314. ACM, 2013.

[26] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Prog. Lang. Syst.*, 5(4):596–619, 1983.

[27] C. B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In *Reflections on the Work of C.A.R. Hoare*, pages 167–187. Springer London, 2010.

[28] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.

[29] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer, 2009.

[30] K. R. M. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

[31] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[32] P. Letouzey. Extraction in Coq: An Overview. In *Computability in Europe*, volume 5028 of *LNCS*, pages 359–369. Springer, 2008.

[33] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574. ACM, 2013.

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

[35] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory Vienna, 1968.

[36] A. Mahboubi and E. Tassi. Canonical structures for the working Coq user. In *ITP*, volume 7998 of *LNCS*, pages 19–34. Springer, 2013.

[37] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014. Extended version is available at `http://software.imdea.org/fcsl/papers/concurroids-extended.pdf`.

[38] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP*, pages 62–73. ACM, 2006.

[39] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240. ACM Press, 2008.

[40] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.

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

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

[43] S. Qadeer, A. Sezgin, and S. Tasiran. Back and forth: Prophecy variables for static verification of concurrent programs. Technical Report MSR-TR-2009-142, Microsoft Research, 2009.

[44] A. Raad, J. Villard, and P. Gardner. CoLoSL: Concurrent Local Subjective Logic. In *ESOP*, volume 9032 of *LNCS*, pages 710–735. Springer, 2015.

[45] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[46] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. Accompanying code and tutorial. Available from `http://software.imdea.org/fcsl`.

[47] I. Sergey, A. Nanevski, and A. Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, volume 9032 of *LNCS*, pages 333–358. Springer, 2015.

[48] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[49] J. Smans, D. Vanoverberghe, D. Devriese, B. Jacobs, and F. Piessens. Shared boxes: Rely-Guarantee reasoning in VeriFast. CW Reports CW662, KU Leuven, May 2014.

[50] M. Sozeau and N. Tabareau. Universe polymorphism in Coq. In *ITP*, volume 8558 of *LNCS*, pages 499–514. Springer, 2014.

[51] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.

[52] R. K. Treiber. Systems programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[53] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.

[54] V. Vafeiadis. RGSep Action Inference. In *VMCAI*, volume 5944 of *LNCS*, pages 345–361. Springer-Verlag, 2010.

[55] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[56] F. Vogels. *Formalisation and Soundness of Static Verification Algorithms for Imperative Programs*. PhD thesis, KU Leuven, 2012.