

Automating Separation Logic for Concurrent C Minor

William Mansky

Princeton University, May 23, 2008

Abstract

In this paper, I demonstrate the implementation of several tools for program analysis in a machine-checked environment. I begin by detailing the implementation of the separation logic for Concurrent C Minor proposed by Hobor, Appel, and Nardelli in the Coq proof assistant. Formalizing the syntax and semantics of CCm and extending the set of semi-automated tactics in the base implementation of C Minor leads to a framework for building proofs of the correctness and safety of concurrent programs. I then describe a thread-modular shape analysis algorithm proposed by Gotsman et al. for inferring lock invariants in concurrent programs, and demonstrate its use as a transformation from a simple C-like language to proved-correct CCm. By reinterpreting the analysis in terms of the application of the rules of concurrent separation logic, I prove its soundness without resorting to lattice theory. Finally, I explain a two-part algorithm, based on the analysis and implemented in Coq, for the construction of CCm programs and proofs of safety from programs in the source language of the analysis.

1 Introduction

With the increasing complexity of modern-day software systems comes an increased demand for program correctness and reliability. Computer programs are commonly used to manage dangerous or security-critical systems, where even the slightest possibility of internal error is unacceptable. Applications such as web services must remain up and running in potentially adversarial environments. Multipart systems are sufficiently complex that exhaustive testing of all possible cases is unfeasible. The field of automated theorem proving offers an answer to these demands in the

form of machine-checked verification. While demonstrating the correctness of a program is often more time-consuming than testing, it can provide guarantees of correctness rather than simply likelihoods. Furthermore, with the development of proof assistants and other tools for program analysis, the task of proving a program has become (and is continuing to become) significantly less burdensome.

Over the past decades, various logical frameworks for analyzing programs have been advanced, starting from Church’s lambda calculus. One such logic, the Calculus of Constructions (CoC), underlies the commonly used Coq proof assistant. From a user’s perspective, Coq consists of two parts. The first is an ML-based programming language called Gallina, which is used to specify logical constructs such as inductive definitions and recursive functions; the second is a system of *tactics* used to semi-automate the proof process [10]. Gallina is designed so that it is possible to reason about entities defined in the language using the Calculus of Constructions; thus, rather than learn the intricacies of the underlying logic, programmers can build CoC predicates using familiar programming language structures. Tactics provide an analogous function when creating proofs; using pattern matching and the basic axioms of the logic, they serve to automatically complete proof steps or simplify complex terms, forming a layer of abstraction over the details of the underlying logic. These abstractions provide an interface to the CoC usable by programmers as well as logicians, as well as speeding up the proof process through automation.

A series of projects, beginning with Leroy’s CompCert [9], have sought to adapt real-world programs to the Coq framework. The result of the CompCert project was an intermediate language called C Minor (Cm), along with a compiler front-end from C to Cm and a back-end from Cm to PowerPC machine code. C Minor has many of the fundamental characteristics of C (e.g. integer representation), and both the language semantics and most of the compiler are defined and proved sound in Coq. Various projects have resulted in the creation of further certified compiler front-ends, for high-level languages including Java and ML. Thus, Cm serves as a fully formalized and machine-verifiable intermediate step between common high-level languages and machine code – it can be used to guarantee properties of compiled code in terms of the source program. Essentially, the CompCert project has become the basis of a framework in which real-world programs can be

analyzed and verified in Coq.

Once the semantics of a programming language are fully formalized, various methods exist for proving properties of programs. One of the earliest and simplest methods is Hoare logic [4], which provides axiomatic semantics for the effects of programming statements on program state in terms of logical assertions. Separation logic, an extension of Hoare logic to deal with dynamic shared memory and pointers developed by Reynolds and O’Hearn [8], is sufficiently expressive to create proofs about correctness of Cm programs. Appel and Blazy have formalized separation logic on Cm in Coq [2], allowing proof writers to take advantage of the machine checking and automation provided by the proof assistant. More recently, Hobor, Appel, and Nardelli have extended both separation logic and Cm to handle concurrency [5], creating a Cm variant called Concurrent C Minor (CCm) which more closely models the behavior of modern multithreaded programs.

By incorporating concurrency, CCm significantly increases the range of real-world programming constructs that can be formally modeled and proven. Based on the concurrency model of C-threads, it has the potential to serve as a platform for proving the correctness of real-world concurrent programs. The first step to realizing this potential is the implementation of CCm in the machine-checked environment of a theorem prover. Since the base Cm language was created in the Coq proof assistant, Coq is a natural choice for this extended version. In addition, just as the Calculus of Constructions is made into a useful real-world tool through the interface of the tactics provided by Coq, concurrent separation logic can be made usable through the development of automated tools and tactics. In this paper, I describe the implementation of CCm and concurrent separation logic in Coq, as well as several tools designed to make the language and the logic accessible to programmers. Ideally, these tools will form the beginning of an interface to theorem proving accessible to anyone familiar with basic C-style programming paradigms.

2 Separation Logic

Separation logic has its origins in Hoare’s seminal paper “An axiomatic basis for computer programming” [4], which proposed a simple set of rules for expressing the effects of a program on the

environment of its local variables. A proposition in Hoare logic¹ takes the form $\{P\}s\{Q\}$, where s is one of several basic statements of a programming language, such as an assignment or a control flow statement, and P and Q are logical predicates on numbers and variables. P is called the precondition, and Q the postcondition, of the statement; each axiom and statement of Hoare logic consists of these three elements, known as the Hoare triple. In Hoare logic, P and Q are expressions in predicate logic; they can be thought of as *assertions* on the state of the program, where the state here consists of the assignment of values to variables. The underlying meaning of the statement $\{P\}s\{Q\}$ is that starting from a state in which P is true, the execution of s will produce a state in which Q is true. More specifically, $\{P\}s\{Q\}$ implies a property of *partial correctness*²: starting from state P , executing the command s will result in either an infinite loop or termination in state Q . The logical expression **true** corresponds to the most general state, in which nothing is known about the assignments of variables, since in classical logic $A \Rightarrow \mathbf{true}$ for all propositions A .

It is important to note the difference between programs that enter an infinite loop and programs that “get stuck”. In an infinite loop, the program continues to execute forever according to the semantics of the language; while the program does not proceed beyond the loop, the statements in the loop body continue to execute correctly. “Getting stuck”, on the other hand, reflects an error in the program, such as an unhandled exception – a case in which the semantics of the language provide no means for the program to proceed. Thus, while partial correctness does not ensure that a program will terminate, it does ensure that it will not enter an invalid state; such a program is called *safe*. Then $\{\mathbf{true}\}s\{Q\}$ expresses the proposition that starting from no information, s will either run to completion and reach state Q , or enter an infinite loop during execution, but not crash or get stuck. $\{\mathbf{true}\}s\{\mathbf{true}\}$ expresses the more general proposition that starting from no information, s is safe.

The usefulness of basic Hoare logic is limited, however, by its range of statements. In particular, there is no support for shared memory; while the value of one variable may be assigned to another, this establishes no further connection between the two variables. A solution to this problem was

¹In Hoare’s paper, this was written as $P\{s\}Q$, but the placement of brackets has been changed in modern notation.

²Other formulations of Hoare logic provide guarantees of total correctness by ensuring that loops will terminate; this paper will deal with only logics of partial correctness.

Figure 1 The rules of separation logic

$$\begin{array}{c}
 \frac{}{\vdash \{P[w/e]\} w := e \{P\}} \text{assignment} \qquad \frac{\vdash \{P\} s_1 \{Q'\} \quad \vdash \{Q'\} s_2 \{Q\}}{\vdash \{P\} s_1; s_2 \{Q\}} \text{composition} \\
 \\
 \frac{\vdash \{B \wedge P\} s \{P\}}{\vdash \{P\} \text{while } B \text{ s } \{\neg B \wedge P\}} \text{iteration} \qquad \frac{\vdash \{B \wedge P\} s_1 \{Q\} \quad \vdash \{\neg B \wedge P\} s_2 \{Q\}}{\vdash \{P\} \text{if } B \text{ then } s_1 \text{ else } s_2 \{Q\}} \text{conditional} \\
 \\
 \frac{}{\vdash \{e \mapsto -\} [e] := e' \{e \mapsto e'\}} \text{mutation} \qquad \frac{w \text{ is not free in } e \text{ or } e'}{\vdash \{e \mapsto e'\} w := [e] \{w = e' \wedge e \mapsto e'\}} \text{lookup} \\
 \\
 \frac{\vdash \{P\} s \{Q\} \quad P' \Rightarrow P}{\vdash \{P'\} s \{Q\}} \text{pre} \qquad \frac{\vdash \{P\} s \{Q\} \quad Q \Rightarrow Q'}{\vdash \{P\} s \{Q'\}} \text{post} \\
 \\
 \frac{\vdash \{P\} s \{Q\} \quad \text{no free variables in } R \text{ are modified by } s}{\vdash \{P * R\} s \{Q * R\}} \text{frame}
 \end{array}$$

first proposed by Owicki and Gries in 1976 [7]. In 2002, Reynolds and O'Hearn discovered a more modular solution with the invention of separation logic, an extension of Hoare logic that allows reasoning about pointers and shared memory, using the abstraction of a heap [8]. The key innovation of separation logic is separating conjunction, expressed with the $*$ operator. The construction $P * Q$ expresses the assertion that both P and Q are true, but of different, nonoverlapping sections of the heap. Thus, $x \mapsto 3 * y \mapsto 3$ means that x is a pointer to a heap cell containing the value 3, and y is a pointer to a heap cell containing the value 3, and x and y do not point to the same cell. In Hoare logic, it is true that for any assertion R whose free variables are not changed by the statement s , $\{P\} s \{Q\}$ implies $\{P \wedge R\} s \{Q \wedge R\}$ – that is, the only parts of the precondition that are affected by a statement are those involving the free variables of the statement. However, this is clearly not true when pointers come into play: $\{x \mapsto 4\} [x] := 3 \{x \mapsto 3\}$ does not imply $\{x \mapsto 4 \wedge y \mapsto 4\} [x] := 3 \{x \mapsto 3 \wedge y \mapsto 4\}$ when x and y point to the same heap cell. However, it is true in separation logic that $\{P\} s \{Q\}$ implies $\{P * S\} s \{Q * S\}$; in the above example, $x \mapsto 4 * y \mapsto 4$ guarantees that modifying the value at x will have no effect on the value at y . This result, known as the *frame rule*, allows the logic to be applied to large-scale programs by separating out the information irrelevant to a particular statement. The full set of rules of separation logic³ is

³Here and throughout this paper, the letter s is used for *statements*, e for *expressions*, w for *variables*, v for *values*,

shown in figure 1. With the proper extensions of the set of assertions, separation logic can express operations on complex data structures.

With its more extensive set of assertions, separation logic can provide a wide range of guarantees about programs with shared memory. As an extension of Hoare logic, separation logic also proves properties of partial correctness, and proving $\{\mathbf{true}\}s\{\mathbf{true}\}$ is sufficient to guarantee that s is safe. As a consequence of the semantics of the rule for lookup, in separation logic, $\{\mathbf{true}\}s\{\mathbf{true}\}$ implies that s is memory-safe, that is, it never dereferences a pointer to an unallocated heap cell. In real-world programs, this property is often both desirable and difficult to attain; however, the semantics of the rule for load ensure that it holds for all programs that have been proved correct with respect to separation logic. It is important to note the difference between the separation logic assertions **true** and **emp**. As in Hoare logic, **true** provides no information about the current state; for any separation logic assertion A , $A \Rightarrow \mathbf{true}$. **emp**, on the other hand, guarantees that the heap is empty, but, like **true**, imposes no conditions on variables. Then in separation logic, $\{\mathbf{emp}\}s\{\mathbf{emp}\}$ implies that s contains no memory leaks; starting with an empty heap, it ends with an empty heap. Both memory safety and the absence of memory leaks are useful real-world properties that can be guaranteed through separation logic, though as will be shown later, not all formulations of separation logic can provide such strong guarantees against memory leaks.

Concurrency adds a new set of challenges in proving a program’s correctness; when multiple threads access and modify the same memory, it is even more difficult to ensure memory safety. O’Hearn [6] proposes an extension of separation logic to support the notion of resources, each of which protects a set of variables and memory locations. O’Hearn then demonstrates the application of this extended separation logic to common idioms of concurrency, including locks and semaphores. This not only extends separation logic’s guarantees of memory safety to concurrent programs, but also allows guarantees against race conditions: if a program can be proved correct with respect to concurrent separation logic, then no two threads access the same resource at the same time. However, the model of concurrency used in this formulation of concurrent separation logic differs from the common multithreading paradigm used in packages such as C-threads, making it difficult to

and capital letters for *assertions*.

extend to real-world programming languages. There are three main features of real-world languages that are absent in O’Hearn’s model. First, the structure of parallelism is determined statically; these rules for concurrent separation logic cannot be applied unless the user indicates in advance the number of threads active at each point in the program. C Minor requires a more dynamic concurrency model, in which threads are created and destroyed at runtime. Second, O’Hearn’s model expresses shared memory through the sharing of local variables across threads (with the appropriate resource restrictions), while threads in most languages share memory but not local variables. Finally, a programmer does not always know exactly which memory locations, or even which variables, should be contained in each resource invariant. While the concurrent extension for Cm addresses all of these issues to some extent, the last is of particular interest, and I will return to it later.

Figure 2 The rules of concurrent separation logic

$$\begin{array}{c}
\frac{\text{resource}(e, R) \quad R \Leftrightarrow (\text{hold } e R * S)}{\Gamma \vdash \{e \mapsto 0\} \text{make_lock } e R \{e \xrightarrow{\pi_{lock}} R * \text{hold } e R\}} \\
\\
\frac{}{\Gamma \vdash \{e \xrightarrow{\pi_{lock}} R * \text{hold } e R\} \text{free_lock } e \{e \mapsto 0\}} \qquad \frac{}{\Gamma \vdash \{e \xrightarrow{\pi} R\} \text{lock } e \{e \xrightarrow{\pi} R * R\}} \\
\\
\frac{}{\Gamma \vdash \{e \xrightarrow{\pi} R * R\} \text{unlock } e \{e \xrightarrow{\pi} R\}} \qquad \frac{R \Leftrightarrow (\text{hold } e R * S)}{\Gamma \vdash \{R\} \text{unlock } e \{\text{emp}\}} \\
\\
\frac{\text{precise } (R)}{\Gamma \vdash \{e : \{R\}\{S\} * R(\vec{e})\} \text{fork } e \vec{e} \{e : \{R\}\{S\}\}}
\end{array}$$

As part of the ongoing Concurrent C Minor project, which aims at extending Leroy’s C Minor with support for concurrency and proofs in separation logic, Hobor, Appel, and Nardelli [5] propose an alternative formulation of concurrent separation logic, modeled after the locks paradigm of C-threads. The rules of CCm separation logic⁴ are shown in figure 2. Threads are created dynamically from functions through the fork command; resource invariants are protected by locks, areas of memory set aside (again, dynamically) as signals between processes. Each lock l is created with

⁴In Hobor et al.’s formulation, 100% is used to indicate the lock share produced by `make_lock` and consumed by `free_lock`. To avoid confusion between this 100% share and a full share, I will instead use π_{lock} throughout.

Figure 3 A concurrent program with a memory leak

```

void main() with pre:{emp} and post:{emp} {
  p := call malloc_and_zero(4);
  fork f(p);
  return ();
}

void f(p) with pre:{p ↦ -} and post:{p ↦ -} {
  while (true) { skip; }
}

```

some lock invariant R , an assertion of separation logic in which l is a free variable. Locking the lock adds R to the environment of what is known, and unlocking it removes R . By assuring that a lock can only be unlocked when the lock invariant is satisfied, this formulation of separation logic provides a guarantee that whenever the lock is unlocked, the invariant holds. R can include conditions on the contents of memory through assertions about pointers, as well as conditions on the lock itself. The lock assertion $e \bullet^{\pi} R$ comes with a share of visibility π , allowing shares of a lock to be passed out to various threads – if $\pi = \pi_1 \oplus \pi_2$, then $e \bullet^{\pi} R \Leftrightarrow e \bullet^{\pi_1} R * e \bullet^{\pi_2} R$. Through this mechanism, a program can give threads access to locks and the memory they protect by passing them as parameters. Note that there are two rules for the `unlock` statement; the first is a straightforward reversal of the rule for `lock`, while the second is more general but less intuitive. The second `unlock` rule is useful in the situation where a lock’s visibility is part of the lock invariant, but in most other cases the first, premise-free rule is sufficient.

The behavior of the `fork` function has important implications for the use of this version of concurrent separation logic. Rather than incorporate the newly spawned thread as part of the current program, as in O’Hearn’s model, CCm separation logic treats the new thread as a separate entity, with its own set of assertions on the current state. The state is divided between the parent and child threads at the moment of the thread’s creation, and thereafter the threads are analyzed sequentially and separately, making the analysis valid for any interleaving of the two threads⁵.

⁵This approach is based on an assumption of *sequential consistency*, which guarantees that the actual execution of a multithreaded program will consist of some interleaving of the threads. On modern multiprocessors, this may not be a valid assumption; the *oracle semantics* of CCm defined by Hobor, Appel, and Nardelli [5] assure sequential

The threads can only communicate by manipulating the locks visible to both of them; otherwise, they have no knowledge of each other's state. One important consequence of this fact is that while $\{\mathbf{true}\}s\{\mathbf{true}\}$ still implies that the thread s is memory safe and free from race conditions, proving $\{\mathbf{emp}\}s\{\mathbf{emp}\}$ does not ensure that s has no memory leaks. It is possible for a program to allocate memory, pass it to a forked thread, and then terminate before the memory is freed, as in the example in figure 3. Here, $\{\mathbf{emp}\}\text{call main}()\{\mathbf{emp}\}$ is provable, but `main` still contains a memory leak.

3 Implementation of CCm Separation Logic

Hobor, Appel, and Nardelli [5] present a theoretical framework for concurrent separation logic in C Minor, extending both the language and the logic to handle C-style concurrency. Cm and its sequential separation logic are defined and proved sound in the Coq proof assistant, allowing the creation of machine-checked proofs based on Coq's extensive library of tactics, as well as several custom tactics such as those created by Appel [1]. The creation of a concurrent extension for Cm then consists of the following steps:

1. The addition of concurrent statements to the definition of the Cm language.
2. The extension of the certified Cm compiler to produce and consume CCm.
3. The formalization of the rules of concurrent separation logic.
4. The extension of the proof of soundness of Cm separation logic to include the new concurrent rules.
5. The extension of the existing framework of tactics to support proofs involving concurrent programs.

In this section, I address items 1, 3, and 5, and use this partial implementation of CCm to prove the correctness of a sample concurrent program by Hobor et al.

consistency on multiprocessors.

In CCm, statements are redefined to include five new operations:

$$s : \text{stmt} ::= \dots \mid \text{lock } e \mid \text{unlock } e \mid \text{fork } e(\vec{e}) \mid \text{make_lock } e R \mid \text{free_lock } e$$

The `make_lock` statement turns a pointer into a lock pointer with a lock invariant R , while `free_lock` turns a lock pointer back into an ordinary pointer; `lock` and `unlock` manipulate the state of the provided lock. From an execution perspective, `make_lock` and `free_lock` have no effect, and `lock` and `unlock` call on the underlying implementation of the locking mechanism: `lock` causes the current thread to wait for the lock to become available, while `unlock` makes it available for the next waiting thread. From a separation logic perspective, as shown in figure 2 above, `make_lock` transforms a state described by a maps-to assertion into a state described by the conjunction of two new assertions: a lock assertion and a hold assertion. The lock assertion $e \bullet^{\pi} R$ indicates that the expression e evaluates to a pointer to a lock with invariant R , and is visible to the current thread with visibility π . The assertion `hold` $e R$ gives the current thread the right to unlock the lock pointed to by e , and implicitly contains a share of the lock's visibility. The other lock statements manipulate these assertions to produce the desired lock behavior.

The final concurrent statement is `fork`, which spawns a new thread from the provided function; from the perspective of separation logic, it acts as a function call that never returns, taking the part of the current state corresponding to its function precondition and disappearing. As per the CCm concurrency model, once a thread is forked, it has no interaction with the parent thread, except by changing the state of mutually visible locks. It is important to note that the lock assertion contains no information about the current state of the lock. The memory protected by the lock is invisible to the current thread when the lock is unlocked, and becomes visible when the current thread locks the lock; this is the only information a thread receives from the `lock` and `unlock` commands. This reflects the fact that it is impossible for a thread that does not currently hold the lock to determine whether it is locked or unlocked, since at the moment of determination its state might be simultaneously changed by another thread.

The lock invariant in the `make_lock` statement takes the form of a separation logic assertion, making it the only statement to refer directly to the logic. In effect, this embeds the set of assertions

of the logic in the set of statements of the language⁶. While this does not result in a problem of recursive definition, since CCm assertions do not refer directly to statements, it does break the abstraction barrier between the language and the logic. While programmers in the original C Minor could write programs without any knowledge of the logic used to prove it correct, CCm programmers must be able to provide separation logic formulas describing the resources protected by each lock. While this does not interfere with the demonstration of a machine-checked proof of a sample CCm program where the invariants are provided, it does present a problem of usability, and the next section of this paper further investigates this problem and provides a partial solution.

The implementation of the extended statement set and separation logic of CCm is sufficient theoretical framework to prove a sample program. However, the construction of the proof itself is difficult given only Coq’s built-in tactics. The Calculus of Constructions is a *deductive* logic, so a proposition once shown to be true is true throughout the proof. Separation logic, on the other hand, is a *linear* logic; moving through a program consumes facts as the state changes from statement to statement. Appel [1] has created tactics for sequential Cm that provide a natural interface for separation logic reasoning in Coq. The most significant of these tactics, and the one requiring the most significant extension for CCm, is the **forward** tactic.

The **forward** tactic is used to move forward through atomic statements, that is, statements that execute a single operation, as opposed to those involving control flow or stack manipulation. The atomic statements of Cm are store (mutation) and assignment, which can be further subdivided into load (lookup) and heap-independent assignment. **forward** can be used when the next statement in a program is an atomic operation, and the current precondition can be shown to be satisfied. It operates by reducing the current proof obligation (goal) to a simpler *subgoal* on the remainder of the program. In sequential Cm, **forward** can be applied in the following cases [1]:

- when the goal is $\{P\}w := e; C\{Q\}$ where w is not free in P or e and e contains no memory

⁶While embedding the assertion type in the statement type in Coq does not result in a recursive definition, it does raise another implementation issue: that of the sort of the statement type. While statements are of sort **Set**, assertions are of sort **Type**, making them too large to embed in a **Set**. Coq offers two possible resolutions of this problem. For the purposes of this section, the concurrent statements are added as axioms, allowing them to be used interchangeably with sequential statements. In the next section, the other approach will be used; concurrent statements will be defined as a separate, larger type, containing both assertions and sequential statements. The former provides greater ease of use, while the latter is more logically sound, and makes possible the structural induction that will be used in the proofs of the next section.

Figure 4 An example of **forward** and other separation logic tactics

Lemma. If e_1 and e_2 contain no memory references, and i and j do not occur free in e_1 or e_2 , $\{e_1 \mapsto 0 * e_2 \mapsto 1\}i := [e_1]; j := i; [e_2] := j\{e_1 \mapsto 0 * e_2 \mapsto 0\}$.

Proof in Coq.

Tactic	Proof Obligation
intros.	$\{e_1 \mapsto 0 * e_2 \mapsto 1\}i := [e_1]; j := i; [e_2] := j\{e_1 \mapsto 0 * e_2 \mapsto 0\}$
forward.	$\{e_1 \mapsto 0 * e_2 \mapsto 1 * i = 0\}j := i; [e_2] := j\{e_1 \mapsto 0 * e_2 \mapsto 0\}$
forward.	$\{e_1 \mapsto 0 * e_2 \mapsto 1 * i = 0 * j = i\}[e_2] := j\{e_1 \mapsto 0 * e_2 \mapsto 0\}$
assert_subst i.	$\{e_1 \mapsto 0 * e_2 \mapsto 1 * j = 0\}[e_2] := j\{e_1 \mapsto 0 * e_2 \mapsto 0\}$
forward.	$e_1 \mapsto 0 * e_2 \mapsto j * j = 0 \Rightarrow e_1 \mapsto 0 * e_2 \mapsto 0$
assert_subst j.	$e_1 \mapsto 0 * e_2 \mapsto 0 \Rightarrow e_1 \mapsto 0 * e_2 \mapsto 0$
sep_trivial.	Proof complete. □

references; the remaining subgoal is $\{P * (w = e)\}C\{Q\}$.

- when the goal is $\{P_1 * (e \mapsto e') * P_2\}w := [e]; C\{Q\}$ where w is not free in e , e' , P_1 , or P_2 ; the remaining subgoal is $\{P_1 * (e \mapsto e') * (w = e') * P_2\}C\{Q\}$.
- when the goal is $\{P_1 * (e \mapsto e') * P_2\}[e] := e_2; C\{Q\}$ where e contains no memory references; the remaining subgoal is $\{P_1 * (e \mapsto e_2) * P_2\}C\{Q\}$.

The usefulness of this tactic can be seen in a proof such as that in figure 4. Here, two other separation logic tactics are also used: **assert_subst**, which replaces all occurrences of a variable with its value, and **sep_trivial**, which automatically proves trivial propositions about assertion implication. The built-in Coq tactic **intros** serves to introduce the variables and premises. Thanks to the **forward** tactic, the proof is straightforward.

For most tactics (such as **assert_subst**), the desired functionality is unchanged in CCM; extending them is simply a matter of adding cases corresponding to the new statements and assertions. The **forward** tactic, however, requires further modification. The new lock manipulation statements (**make_lock**, **free_lock**, **lock**, and **unlock**) are also atomic, and thus **forward** should be extended to handle the most common cases of their use. **forward** can be extended to cover the concurrent atomic statements through the addition of four new cases:

- when the goal is $\{P_1 * e \mapsto 0 * P_2\}\text{make_lock } e \ R; C\{Q\}$ and R is a valid lock invariant for e ; the remaining subgoal is $\{P_1 * e \xrightarrow{\pi_{\text{lock}}} R * \text{hold } e \ R * P_2\}C\{Q\}$.

- when the goal is $\{P1 * e \xrightarrow{\pi_{lock}} R * \text{hold } e R * P2\} \text{free_lock } e; C\{Q\}$; the remaining subgoal is $\{P1 * e \mapsto 0 * P2\} C\{Q\}$.
- when the goal is $\{P1 * e \xrightarrow{\pi} R * P2\} \text{lock } e; C\{Q\}$; the subgoal is $\{P1 * e \xrightarrow{\pi} R * R * P2\} C\{Q\}$.
- when the goal is $\{P1 * R * P2\} \text{unlock } e; C\{Q\}$ and R is a valid lock invariant for e ; the subgoal is $\{P1 * P2\} C\{Q\}$.

Conditions for valid lock invariants are defined in the axioms of CCm separation logic (see figure 2 for details).

Using this implementation of CCm and concurrent separation logic, along with the newly extended tactics, it is possible to construct a machine-checked proof of the example in appendix B of Hobor et al.⁷[5] The lemma in the example can be stated formally as

Theorem 3.1. $\forall l. \Gamma \vdash \{l \xrightarrow{\pi_f} R(l)\} \text{call } f(l) \{\mathbf{emp}\} \wedge \Gamma \vdash \{\mathbf{emp}\} \text{call } \text{main } () \{\mathbf{emp}\}$

where $\Gamma(f) = f : \{\mathbf{emp}\} \{\mathbf{emp}\}$.

Figures 5 and 6 show the program annotated with separation logic pre- and postconditions at each line. Examination should show that each line, together with its precondition and postcondition, forms a valid and provable separation logic triple⁸. Note that the assertions in `main` refer to a lock share π_{main} that does not appear in the program; at the `fork` statement, a share π_f of the lock visibility is split off and passed to `f`, and π_{main} is the share that remains, i.e., the share such that $\pi_f \oplus \pi_{main} = \pi_{lock}$.

As an example of the reasoning used in the proof, I will outline the proof of what may be the least obvious step: the statement `lock l` in `f`.

Lemma 1. $\{l \xrightarrow{\pi_f} R(l)\} \text{lock } l \{l \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * \exists v. (l+1) \mapsto v * (l+2) \mapsto v * (l+3) \mapsto 1\}$.

Proof.

1. By the lock rule, $\{l \xrightarrow{\pi_f} R(l)\} \text{lock } l \{l \xrightarrow{\pi_f} R(l) * R(l)\}$.

⁷As mentioned earlier, the percentage notation for lock shares has been modified for clarity; the share indicated by 50% in Hobor et al. is labeled as π_f here, and 100% is written as π_{lock} .

⁸Actually, as explained by Appel and Blazy [2], the separation logic “triple” contains six elements; in addition to the standard Hoare triple and the global context Γ , there are also a return context R and block-exit context B . Here R and B are trivial, and are omitted in the presentation of separation logic propositions.

Figure 5 Sample concurrent program and assertions – part 1

```

void main() with pre:{emp} and post:{emp} {
  {emp}
  L := call malloc_and_zero(4);
  {L  $\mapsto$  0 * (L + 1)  $\mapsto$  0 * (L + 2)  $\mapsto$  0 * (L + 3)  $\mapsto$  0}
  i := 0;
  {L  $\mapsto$  0 * (L + 1)  $\mapsto$  0 * (L + 2)  $\mapsto$  0 * (L + 3)  $\mapsto$  0 * i = 0}
  make_lock L R(L);
  {L  $\xrightarrow{\pi_{lock}}$  R(L) * hold L R(L) * (L + 1)  $\mapsto$  0 * (L + 2)  $\mapsto$  0 * (L + 3)  $\mapsto$  0 * i = 0}
  fork f(L);
  {L  $\xrightarrow{\pi_{main}}$  R(L) * hold L R(L) * (L + 1)  $\mapsto$  0 * (L + 2)  $\mapsto$  0 * (L + 3)  $\mapsto$  0 * i = 0}
  *(L+3) := 1;
  {L  $\xrightarrow{\pi_{main}}$  R(L) * hold L R(L) * (L + 1)  $\mapsto$  0 * (L + 2)  $\mapsto$  0 * (L + 3)  $\mapsto$  1 * i = 0}
  block {loop {
    {L  $\xrightarrow{\pi_{main}}$  R(L) * R(L) *  $\exists v'.i = v'$ }
    if (*(L+3)==0) exit 0 else skip;
    {L  $\xrightarrow{\pi_{main}}$  R(L) * hold L R(L) *  $\exists v.(L + 1) \mapsto v * (L + 2) \mapsto v * (L + 3) \mapsto 1 * \exists v'.i = v'$ }
    *(L+1) := i; *(L+2) := i;
    {L  $\xrightarrow{\pi_{main}}$  R(L) * hold L R(L) *  $\exists v.(L + 1) \mapsto v * (L + 2) \mapsto v * (L + 3) \mapsto 1 * \exists v'.i = v'$ }
    unlock L;
    {L  $\xrightarrow{\pi_{main}}$  R(L) *  $\exists v'.i = v'$ }
    i := i+1;
    {L  $\xrightarrow{\pi_{main}}$  R(L) *  $\exists v'.i = w$ }
    lock L;
    {L  $\xrightarrow{\pi_{main}}$  R(L) * R(L) *  $\exists v'.i = v'$ }
  }}
  {L  $\xrightarrow{\pi_{lock}}$  R(L) * hold L R(L) *  $\exists v.(L + 1) \mapsto v * (L + 2) \mapsto v * (L + 3) \mapsto 0$ }
  free_lock L;
  {L  $\mapsto$  0 *  $\exists v.(L + 1) \mapsto v * (L + 2) \mapsto v * (L + 3) \mapsto 0$ }
  call free(L,4);
  {emp}
  return ();
  {emp}
}

```

Figure 6 Sample concurrent program and assertions – part 2

```

void f(l) with pre:  $\{l \bullet \xrightarrow{\pi_f} R(l)\}$  and post:  $\{\mathbf{emp}\}$  {
   $\{l \bullet \xrightarrow{\pi_f} R(l)\}$ 
  loop {
     $\{l \bullet \xrightarrow{\pi_f} R(l)\}$ 
    lock l;
     $\{l \bullet \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * \exists v. (l+1) \mapsto v * (l+2) \mapsto v * (l+3) \mapsto 1\}$ 
     $* (l+1) := * (l+1) * 2;$ 
     $\{l \bullet \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * \exists v. (l+1) \mapsto v * 2 * (l+2) \mapsto v * (l+3) \mapsto 1\}$ 
     $* (l+2) := * (l+2) * 2;$ 
     $\{l \bullet \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * \exists v. (l+1) \mapsto v * (l+2) \mapsto v * (l+3) \mapsto 1\}$ 
    if  $(* (l+1) > 10)$  {  $* (l+3) := 0;$  unlock l; return (); }
    else skip;
     $\{l \bullet \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * \exists v. (l+1) \mapsto v * (l+2) \mapsto v * (l+3) \mapsto 1\}$ 
    unlock l;
     $\{l \bullet \xrightarrow{\pi_f} R(l)\}$ 
  }
   $\{\mathbf{emp}\}$ 
}
 $S(l, P) = (\exists v. ((l+1) \mapsto v * (l+2) \mapsto v) * ((l+3) \mapsto 1) \vee ((l+3) \mapsto 0) * l \bullet \xrightarrow{\pi_f} P))$ 
 $R(l) = \mu P. (\text{hold } l P) * S(l, P)$ 

```

2. By the definition of R , $R(l) \Leftrightarrow \text{hold } l R(l) * S(l, R(l))$.
3. Then $l \bullet \xrightarrow{\pi_f} R(l) * R(l) \Leftrightarrow l \bullet \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * S(l, R(l))$.
4. $S(l, R(l)) = \exists v. ((l+1) \mapsto v * (l+2) \mapsto v * ((l+3) \mapsto 1 \vee ((l+3) \mapsto 0 * l \bullet \xrightarrow{\pi_f} R(l)))$.
5. Now, among the clauses of the separating conjunction in the postcondition, we have both $l \bullet \xrightarrow{\pi_f} R(l)$ and $((l+3) \mapsto 1 \vee ((l+3) \mapsto 0 * l \bullet \xrightarrow{\pi_f} R(l)))$. However, a share can never be joined to itself, and so $l \bullet \xrightarrow{\pi_f} R(l) * l \bullet \xrightarrow{\pi_f} R(l) \Rightarrow \mathbf{false}$.
6. Then the right branch of the disjunction cannot hold, and so the left branch must hold.
7. Then the postcondition can be rewritten as $l \bullet \xrightarrow{\pi_f} R(l) * \text{hold } l R(l) * \exists v. (l+1) \mapsto v * (l+2) \mapsto v * (l+3) \mapsto 1$. □

The rest of the proof proceeds along similar lines. Here the Coq implementation of CCm and concurrent separation logic, along with the extended tactics described above, provides a framework for the construction of a complete machine-checked proof of the memory safety and freedom from race conditions of this sample real-world program.

4 Thread-Modular Shape Analysis

One of the most difficult aspects of applying formal program analysis methods to real-world concurrent programs is the problem of lock invariants. While most programmers are familiar with the lock paradigm of C-threads, they rarely consider exactly what conditions a lock enforces. In the case of CCm, `make_lock` is the only statement that contains a separation logic assertion, breaking the abstraction that allows programmers to use the language without understanding the underlying proof mechanisms. However, it is far more difficult to reason about programs with locks when the lock invariants are not known at the start of the analysis. While it is not possible to use separation logic as presented here without providing lock invariants in code, for a certain subset of CCm, it is possible to infer the lock invariants using proven program analysis methods.

Gotsman et al.[3] have proposed a thread-modular shape analysis for discovering the lock invariants of concurrent programs, precisely the tool to make concurrent separation logic accessible and restore the abstraction between statements and assertions. As a thread-modular analysis, it operates separately on each thread of the program, without requiring a model of concurrency that tracks the interleaving of threads. The analysis infers lock invariants in terms of separation logic assertions, making it particularly suited for this application. While the paper cited proposes a general framework for such analyses, defined in terms of lattice theory, its demonstrative example demonstrates the application of the framework to a C-like programming language, with locks much like those used in CCm. Thus, by implementing the analysis in the machine-checked environment of CCm, we obtain a tool that can be used to translate a simple subset of CCm, with a `make_lock` statement that requires no information about the lock invariant, to a memory-safe program in full CCm. The analysis and the associated tool will be the subject of this section.

The tactics used in the sample proof of the previous section were an example of semi-automated

proof tools. A user creates a machine-checked proof in Coq by combining built-in and custom tactics, using his judgement to determine the appropriate tactic for each step, and occasionally providing information that the proof assistant cannot determine automatically. The shape analysis, on the other hand, is a fully automated algorithm, which produces lock invariants without human guidance. As a result, some elements of CCm may be too complex for the algorithm to handle. Both the set of concurrent statements and the set of separation logic assertions must be restricted to eliminate any ambiguity in the analysis. While some generality is lost through the fully automated approach, there is a corresponding increase in usability. Without knowing anything about concurrent separation logic, a user can write a program in the algorithm's input language and have it automatically transformed into a valid CCm program with a guarantee of memory safety.

Figure 7 Sample program for thread-modular shape analysis

<pre> void thread1(<i>l</i>) { while(<i>nondet</i>()) { lock <i>l</i>; <i>n</i> := malloc_and_zero(3); *<i>n</i> := data; next := *(<i>l</i>+1); *(<i>n</i>+1) := next; *(<i>n</i>+2) := <i>l</i>; *(<i>l</i>+1) := <i>n</i>; next := *(<i>n</i>+1); *(next+2) := <i>n</i>; unlock <i>l</i>; } return (); } </pre>	<pre> void thread2(<i>l</i>) { while(<i>nondet</i>()) { lock <i>l</i>; <i>n</i> := *(<i>l</i>+1); if(<i>n</i> != <i>l</i>) { prev := *(<i>n</i>+2); next := *(<i>n</i>+1); *(prev+1) := next; *(next+2) := prev; data := *<i>n</i>; call free(<i>n</i>,3); } else skip; unlock <i>l</i>; } return (); } </pre>	<pre> void main() <i>l</i> := malloc_and_zero(3); make_lock <i>l</i>; *(<i>l</i>+1) := <i>l</i>; *(<i>l</i>+2) := <i>l</i>; unlock <i>l</i>; fork thread1(<i>l</i>); fork thread2(<i>l</i>); return (); } </pre> <div style="display: flex; justify-content: space-around; margin-top: 20px;"> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td><i>n</i></td><td><i>data</i></td></tr> <tr><td><i>n</i> + 1</td><td><i>next</i></td></tr> <tr><td><i>n</i> + 2</td><td><i>prev</i></td></tr> </table> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td><i>l</i></td><td><i>lock</i></td></tr> <tr><td><i>l</i> + 1</td><td><i>next</i></td></tr> <tr><td><i>l</i> + 2</td><td><i>prev</i></td></tr> </table> </div>	<i>n</i>	<i>data</i>	<i>n</i> + 1	<i>next</i>	<i>n</i> + 2	<i>prev</i>	<i>l</i>	<i>lock</i>	<i>l</i> + 1	<i>next</i>	<i>l</i> + 2	<i>prev</i>
<i>n</i>	<i>data</i>													
<i>n</i> + 1	<i>next</i>													
<i>n</i> + 2	<i>prev</i>													
<i>l</i>	<i>lock</i>													
<i>l</i> + 1	<i>next</i>													
<i>l</i> + 2	<i>prev</i>													

Figure 7 shows a sample program for the thread-modular shape analysis, adapted to CCm from Gotsman et al.[3] `main` forks off threads `thread1` and `thread2`; `thread1` adds nodes to the circular doubly-linked list at `l`, while `thread2` removes nodes from the list as long as more than one node remains. With the addition of sequential code to generate and process the data in the list, this models a common paradigm in concurrent programming. The most significant differences between

this program and the example in the previous section are the `make_lock` statement in `main`, which does not declare the lock invariant for l , and the use of temporary variables `prev` and `next` to separate store and load operations. These and other special features of this program will be examined in further detail as the analysis is explained.

To simplify the separation-logic representation of the data structures in the program, Gotsman et al. define two new assertions, `node` and `dll`. `node(i, n, p)` asserts that i is a node in a doubly-linked list as shown in figure 7, with next pointer n and previous pointer p . `dll(f, p, n, l)` asserts that the heap forms a doubly-linked list with first node f and last node l , where p is the previous pointer of f and n is the next pointer of l . Formally,

- $\text{node}(i, n, p) ::= \exists d. i \mapsto d * (i + 1) \mapsto n * (i + 2) \mapsto p$
- $\text{dll}(f, p, n, l)$ is the least assertion such that $\text{dll}(f, p, n, l) \Leftrightarrow (f = n \wedge p = l \wedge \mathbf{emp}). \vee \exists x. (\text{node}(f, x, p) * \text{dll}(x, f, n, l))$

The demonstration of the analysis on the sample program will make use of these definitions.

The basis of the algorithm is an underlying sequential analysis, which is run repeatedly on each thread to build up increasingly accurate approximations of the lock invariants of the program. On the sample program, the first step is to analyze the `main` function and find an initial approximation for each lock invariant. The line `make_lock l` signals to the analysis that l is a lock for which the invariant must be inferred. The analysis then tracks the state of the program until the `unlock` statement, at which point part of the state is split off and used as an approximation of the lock invariant. Recall that in the simple separation logic rule for `unlock`, as shown in figure 2, the effect of an `unlock` statement is to subtract the lock invariant from the current state. Similarly, at an `unlock` statement, the analysis subtracts a portion of the current state and uses it as its approximation of the lock invariant. In this case, the state at the line `unlock l` includes $(l + 1) \mapsto l * (l + 2) \mapsto l$, which is used as the first approximation I_0 of the invariant of l .

The algorithm then runs the sequential analysis on one of the threads. The order chosen has no effect on the outcome of the analysis; in this case, we will assume that it starts with `thread1`. Entering the `while` loop, it iterates over the loop body to determine the loop invariant. At the line

lock l , the analysis adds the current approximation of the lock invariant for l to the current state, just as the separation logic rule for **lock** adds the lock invariant to the current state. At the line **unlock** l , the analysis again subtracts a portion of the current state and adds it to the lock invariant as a disjunct. In this case, the state at the **unlock** statement includes $(l + 1) \mapsto \mathbf{n} * (l + 2) \mapsto \mathbf{n} * \mathbf{node}(\mathbf{n}, \mathit{data}, l, l)$. Since \mathbf{n} is a local variable, it must be existentially quantified for use in the lock invariant of l , yielding the new disjunct $I_1 = \exists x. ((l + 1) \mapsto x * (l + 2) \mapsto x * \mathbf{node}(x, \mathit{data}, l, l))$. The approximation of the lock invariant of l is now $I_0 \vee I_1$, and this new invariant will be used in the analysis of the next thread.

Next the analysis moves to the code of **thread2**. At lock l , both possibilities for the lock invariant of l must be considered. Under the approximation I_0 , $(l + 1) \mapsto l$, so the code inside the **if** clause is not executed, and no further information is discovered. Using the approximation I_1 results in a state containing $(l + 1) \mapsto l * (l + 2) \mapsto l$ at the **unlock** statement, which is equal to I_0 , so again no new information is discovered, and this completes the first iteration over the threads of the program. The second iteration performs the same process with the newly discovered case I_1 of the lock invariant. In **thread1**, another new possibility is discovered, which can be abstracted to $I_2 = \exists x, y. ((l + 1) \mapsto x * (l + 2) \mapsto y * \mathbf{dll}(y, l, l, x))$. In **thread2**, this leads to the rediscovery of the invariant case I_1 , and so no further information is added. In the third iteration, trying the new case I_2 in each thread leads to no new cases of the lock invariant. The analysis has reached a fixed point, and $I_0 \vee I_1 \vee I_2$ is the final approximation of the lock invariant of l .

If the algorithm is sound, a program on which it completes is guaranteed to have the following three properties: the approximated lock invariants are valid, each thread in the program is memory safe, and the program contains no race conditions. These conditions are also those guaranteed for a program p for which $\{\mathbf{true}\}p\{\mathbf{true}\}$ has been shown. Furthermore, much of the sequential analysis consists of finding separation logic assertions on the program state at each **lock** and **unlock** statement. This suggests that the entire algorithm may be expressible in terms of the application of the rules of separation logic. Then the soundness of the algorithm would follow from the soundness of separation logic, and creating a Coq implementation of the algorithm for CCm would be straightforward. The rest of this section will deal with the details and the consequences of this

reinterpretation.

The analysis determines the program state at each lock manipulation statement through a separation logic assertion. However, the assertions used by the analysis are different from the general assertions of separation logic. While in separation logic, any condition on program state that satisfies certain requirements is a valid assertion⁹, the analysis uses a simplified set of assertions to capture only the relevant aspects of the program state. This set of simple assertions can be inductively defined as follows:

$$\begin{aligned}
A : \text{simple_assert} \quad ::= & \mathbf{emp} \mid \mathbf{true} \mid \mathbf{false} \mid A \wedge A \mid A \vee A \mid A * A \mid e = e \mid e \mapsto e \\
& \mid e : \{\overline{\text{is_lock } w \pi}\} \mid \text{defined}(e) \mid \text{is_lock } w \pi \mid \text{hold } w
\end{aligned}$$

The first line of cases are some of the standard terms and operators of separation logic assertions, and their meaning is unchanged in the context of the algorithm. $\text{defined}(e)$ asserts that e has a *defined* value, that is, that it can be evaluated in the program context to some value of the language; formally, $\text{defined}(e) ::= \exists(v : \text{value}). e = v$. $\text{is_lock } w \pi$ and $\text{hold } w$ correspond to the standard assertions $w \bullet^{\pi} R$ and $\text{hold } w R$ respectively, but with two important differences: the lock pointer must be an identifier (specifically a *global* identifier), and the lock invariant is not specified. The latter is necessary in an algorithm for finding lock invariants, while the former allows locks to be uniquely identified across multiple threads. Finally, $e : \{\overline{\text{is_lock } w \pi}\}$ asserts that e is a function with precondition equal to the separating conjunction of some number of is_lock assertions on its arguments; since the return values of forked threads are not checked, the postcondition can be assumed to be \mathbf{true} . In the original statement of the algorithm, the initial state of each thread was held to be \mathbf{emp} , and no mention is made of lock-related assertions; however, the is_lock and hold assertions are implicitly present throughout the algorithm, and this is reflected in this reduced set of assertions.

This simplified assertion set is accompanied by an implication relation analogous to the one on full separation logic assertions. In standard separation logic, $P \Rightarrow Q ::= P(st) \Rightarrow Q(st)$, that is, for all states in which P holds, Q also holds. This definition of implication, while powerful and general,

⁹For more on the precise definition of assertions, see Appel and Blazy [2]

is impossible to check algorithmically. Thus, the corresponding relation on simple assertions must have a more limited scope, while remaining powerful enough to produce the preconditions required for the various separation logic rules. Such an implication algorithm must be able to handle at least the following cases:

- associativity and commutativity of the $*$, \wedge , and \vee operators,
- substitution of expressions used in equalities,
- splitting off and possibly combining shares of lock visibility,
- deduction of **defined** assertions from other information about the state,
- expansion and collapsing of data structure assertions such as **node** and **dll**.

As long as the implication algorithm is sound, its range does not affect the soundness of the algorithm, but it does impact its versatility, and as such is an important topic for further research.

We can now formally define the language on which the algorithm operates, and in which the example in figure 7 is written. This language uses the expression syntax of CCm, but with a modified set of statements adapted to the requirements of the algorithm. This statement type is defined inductively as

$$\begin{aligned} ss : \text{simple_stmt} \quad ::= & w := e \mid [e] := e \mid \text{skip} \mid \text{return}() \mid \text{if } e \text{ then } ss \text{ else } ss \mid \text{while } e \text{ } ss \mid ss; ss \\ & \mid \text{malloc_and_zero}(n) \mid \text{make_lock } w \mid \text{free_lock } w \mid \text{lock } w \mid \text{unlock } w \mid \text{fork } e \vec{w} \end{aligned}$$

The first line of cases is a subset of the statements of sequential Cm, while the remainder are adapted from the concurrent statements of CCm. As noted above, lock manipulation statements are constrained to operate on global identifiers, the only arguments to a forked thread must be lock identifiers, and the **make_lock** statement does not declare a lock invariant. **make_lock** can only be used in the main function of a program, and not in any forked threads, so that the algorithm can identify initial lock invariants. Further, the **loop**, **block**, and **exit** commands of sequential Cm are subsumed by **while**, restricting the language to simple loops, and function calls are omitted. The

`return` command takes no arguments, since the return values of forked threads are ignored. Since function calls do not exist in the simplified language, standard function `malloc_and_zero` is made into a statement, and its argument is required to be a number rather than an expression, reducing the amount of computation required in the algorithm. cursory examination should show that the example program above consists entirely of these simple statements.

Note that from this definition, not every simple assertion is a complete separation logic assertion, and not every simple statement is a complete CCm statement. However, we can define a straightforward transformation from simple assertions and statements to full assertions and statements. The incomplete assertions are `is_lock` and `hold`, while the incomplete statement is `make_lock`; all three are missing lock invariants, while all other cases of assertions and statements are valid in full separation logic and CCm. Then given a map θ from lock pointers to lock invariants, define A with θ as the recursive replacement of assertions of the form `is_lock w π` and `hold w` in A by $w \xrightarrow{\pi} \theta[w]$ and `hold w $\theta[w]$` respectively, and ss with θ as the recursive replacement of the statement `make_lock w` by `make_lock w $\theta[w]$` in ss . Then A with θ is a separation logic assertion, and ss with θ is a CCm statement. These transformations are undefined when θ does not contain a lock invariant for some lock pointer w in A or ss . Since the output of the thread-modular shape analysis is a map from the locks in a program to their lock invariants, the analysis can be seen as providing the missing link θ between the source language and full CCm.

Given these definitions, the action of the sequential analysis on a thread can be expressed in terms of the syntax-directed application of a set of inductive rules similar to those of separation logic, as shown in figures 8 and 9. The conclusions of these rules are judgements of the form $\Gamma \vdash \langle \theta_P, P \rangle s \langle \theta_Q, Q \rangle$, where P and Q are simple assertions, s is a simple statement, and θ_P and θ_Q are maps from global identifiers to simple assertions. As in the separation logic judgement $\Gamma \vdash \{P\} s \{Q\}$, P is an assertion on the state of the program before s and Q is an assertion on the state of the program after s . Similarly, θ_P is the set of approximated lock invariants before s is analyzed, and θ_Q is the set updated with the new information learned from s . $\theta[w]$ represents the approximation of the lock invariant for the lock at w . In many rules no new information is gained about the lock invariant; in these cases, θ_P and θ_Q are omitted, and are assumed to be identical.

Figure 8 The sequential analysis as inductive rules part 1: sequential statements

$$\begin{array}{c}
\frac{\text{imsv } ss \ R \quad \Gamma \vdash \langle P \rangle ss \langle Q \rangle}{\Gamma \vdash \langle P * R \rangle ss \langle Q * R \rangle} \text{frame} \qquad \frac{\text{pure } e \quad \text{inde } w \ e \quad P \Rightarrow \text{defined}(e)}{\Gamma \vdash \langle P \rangle w := e \langle \text{defined}(e) * w = e \rangle} \\
\\
\frac{\text{pure } e' \quad \text{inde } w \ e \quad \text{inde } w \ e' \quad P \Rightarrow e \mapsto e'}{\Gamma \vdash \langle P \rangle w := *(e) \langle e \mapsto e' * w = e' \rangle} \\
\\
\frac{\text{pure } e \quad \text{pure } e' \quad P \Rightarrow e \mapsto e' * \text{defined}(e_1)}{\Gamma \vdash \langle P \rangle *(e) := e_1 \langle e \mapsto e_1 \rangle} \qquad \frac{}{\Gamma \vdash \langle P \rangle \text{skip} \langle P \rangle} \\
\\
\frac{}{\Gamma \vdash \langle P \rangle \text{return } () \langle \text{false} \rangle} \qquad \frac{\Gamma \vdash \langle \theta_0, P \rangle ss_1 \langle \theta_1, Q \rangle \quad \Gamma \vdash \langle \theta_1, P \rangle ss_2 \langle \theta_2, Q \rangle \quad \text{pure } e}{\Gamma \vdash \langle \theta_0, P \rangle \text{if } e \text{ then } ss_1 \text{ else } ss_2 \langle \theta_2, Q \rangle} \\
\\
\frac{\Gamma \vdash \langle \theta_0, P \rangle ss_1 \langle \theta_1, Q \rangle \quad \Gamma \vdash \langle \theta_1, Q \rangle ss_2 \langle \theta_2, Q' \rangle}{\Gamma \vdash \langle \theta_0, P \rangle ss_1 ; ss_2 \langle \theta_2, Q' \rangle} \\
\\
\frac{P \Rightarrow \mathbf{emp} \quad n > 0}{\Gamma \vdash \langle P \rangle w := \text{malloc_and_zero}(n) \langle w \mapsto 0 * (w + 1) \mapsto 0 * \dots * (w + n - 1) \mapsto 0 \rangle}
\end{array}$$

Three new predicates appear in the premises of these rules: **pure**, **inde**, and **imsv**. These are defined as follows:

- **pure** e means that the expression e contains no memory references.
- **inde** $w \ e$ means that w is not free in e .
- **imsv** $ss \ P$ means that no free variables in P are modified by ss (**imsv** stands for “independent of modified statement variables”).

The truth of these properties can be determined algorithmically through case analysis. Two additional helper functions are used in the rule for **unlock**. **split_local** separates an assertion P into two parts: a *local* assertion P_{loc} , which is retained as the current state after the **unlock** statement, and a *frame* assertion P_r , which is taken as a new possibility for the lock invariant. This split is chosen such that $P \Leftrightarrow P_{loc} * P_r$. For a more detailed explanation of the split into local and frame assertions, see Gotsman et al.[3] The second helper function, **add_disj**, adds a disjunct to the current approximation of a lock invariant, as long as that particular disjunct is not already present in the invariant; it also existentially quantifies any local variables in the new disjunct, and abstracts

Figure 9 The sequential analysis as inductive rules part 2: while and concurrent statements

$$\begin{array}{c}
\frac{\Gamma \vdash \langle \theta_0, P \rangle ss \langle \theta_1, P \rangle}{\Gamma \vdash \langle \theta_0, P \rangle \text{while } e \text{ ss } \langle \theta_1, P \rangle} \qquad \frac{\Gamma \vdash \langle \theta_0, P \rangle ss \langle \theta_1, Q \rangle \quad \Gamma \vdash \langle \theta_1, Q \rangle \text{while } e \text{ ss } \langle \theta_2, Q' \rangle}{\Gamma \vdash \langle \theta_0, P \rangle \text{while } e \text{ ss } \langle \theta_2, Q' \rangle} \\
\\
\frac{P \Rightarrow w \mapsto 0}{\Gamma \vdash \langle \theta, P \rangle \text{make_lock } w \langle \text{add_disj } \theta[w] \text{ emp, is_lock } e \pi_{lock} * \text{hold } w \rangle} \\
\\
\frac{P \Rightarrow \text{is_lock } w \pi_{lock} * \text{hold } w}{\Gamma \vdash \langle P \rangle \text{free_lock } w \langle w \mapsto 0 \rangle} \qquad \frac{P \Rightarrow \text{is_lock } w \pi \quad \theta[w] = A}{\Gamma \vdash \langle \theta, P \rangle \text{lock } w \langle \theta, \text{is_lock } w \pi * \text{hold } w * A \rangle} \\
\\
\frac{P \Rightarrow \text{is_lock } w \pi * \text{hold } w * P_0 \quad \theta[w] = A \quad \text{split_local } P_0 = (P_{loc}, P_r)}{\Gamma \vdash \langle \theta, P \rangle \text{unlock } w \langle \text{add_disj } \theta[w] P_r, \text{is_lock } w \pi * P_{loc} \rangle} \\
\\
\frac{\Gamma(e) = e : \{R\} \quad P \Rightarrow R(\vec{w})}{\Gamma \vdash \langle P \rangle \text{fork } e \vec{w} \langle \text{emp} \rangle}
\end{array}$$

matching clauses into the associated data structure assertions.

Given these rules, the sequential analysis can be described as follows. At each statement in the thread under consideration, the analysis begins by applying its version of the frame rule to remove extraneous information. It then chooses the rule that matches the structure of the current statement, and uses its auxiliary functions (including the implication algorithm described above) to verify the rule's premises given the current state P and lock approximations θ_P . If any of these premises cannot be verified, the program is either incorrect or too complex, and the algorithm fails. Otherwise, the postcondition Q and new approximations θ_Q specified by the rule are used as the current state and approximations for the next statement in the thread. The two rules for while cause the algorithm to iterate over the body of a loop, accumulating cases of lock invariants, until the loop invariant reaches a fixed point; even if the loop is infinite in the execution of the program, as long as a stable loop invariant can be found, the analysis will proceed. Once no more statements remain, the analysis is complete. Note that the assertions constructed in the map θ are not valid lock invariants, as they do not satisfy the precondition of the separation logic rule for `make_lock` (see figure 2); to remedy this, we define a function `mklk` such that `mklk $\theta[l] \Leftrightarrow \text{hold } l (\text{mklk } \theta[l]) * \theta[l]$` for all $\theta[l]$, and the lock invariant for a lock l in the program is approximated as `mklk $\theta[l]$` at the end of the analysis. For simplicity, this replacement will be assumed to be performed by the with

function described above in further notation.

As a concrete example of this description, consider the analysis on its first iteration through the sample program in figure 7, at the line `lock l`. The precondition for `thread1` is `is_lock l π1` for some π_1 such that $\pi_1 \oplus \pi_2 = \pi_{lock}$, and the current approximation for the lock invariant of l is I_0 , so the current lock map θ_c has $\theta_c[l] = I_0$. The algorithm chooses the `lock` rule from figure 9 and uses the implication algorithm to verify that `is_lock l π1` \Rightarrow `is_lock l π1`. This leads to a conclusion of the form $\Gamma \vdash \langle \theta, P \rangle \text{lock } w \langle \theta, \text{is_lock } w \pi * \text{hold } w * A \rangle$. Substituting the values of P , θ , and w gives $\Gamma \vdash \langle \theta_c, \text{is_lock } l \pi_1 \rangle \text{lock } l \langle \theta_c, \text{is_lock } l \pi_1 * \text{hold } l * I_0 \rangle$, so the current map is unchanged, and the current state becomes `is_lock l π1 * hold l * I0`. This is consistent with the behavior of the analysis as described above, while maintaining the lock-related assertions appropriate to the current state of l .

Since the rules of the sequential analysis are derived from those of separation logic, it is natural to expect that the results of the analysis on a thread also show something about the separation logic properties of the thread. Since the lock invariant approximation changes over the course of the analysis of a thread, it is not possible to assert that $\Gamma \vdash \langle \theta_P, P \rangle ss \langle \theta_Q, Q \rangle \Rightarrow \Gamma \vdash \{P\} ss \{Q\}$ in the general case. However, this property does hold during the last iteration on a thread, when the set of lock approximations is unchanged.

Theorem 4.1. $\Gamma \vdash \langle \theta, P \rangle ss \langle \theta, Q \rangle \Rightarrow \Gamma \vdash \{P \text{ with } \theta\} ss \text{ with } \theta \{Q \text{ with } \theta\}$.

Proof: by structural induction on the judgement $\Gamma \vdash \langle \theta, P \rangle ss \langle \theta, Q \rangle$.

The conclusion $\Gamma \vdash \langle \theta, P \rangle ss \langle \theta, Q \rangle$ can only be reached by the application of one of the rules of figures 8 and 9. Thus, the lemma can be proved by showing that each rule of the analysis is transformed by with to a valid rule of separation logic. The rules in figure 8 lead directly to cases of the rules of Cm separation logic¹⁰, as laid out by Appel and Blazy [2]; in each case, the rule is a composition of the pre rule with a specialization of the rule for the case of `ss`. The first and second `while` rules both match the separation logic rule for `while`; in the first case, the loop invariant is P ,

¹⁰One more complicated case is the `return` rule, which in standard Cm separation logic depends on the return context R . For the purposes of the algorithm, since values will never be returned and the postconditions of all threads are assumed to be **true**, R can be assumed to map the empty list to **true** in all threads, leading to the analysis rule shown above.

and in the second it is $P \vee Q \vee Q'$. The analysis rules for `make_lock`, `free_lock`, and `lock` also lead directly to compositions of pre and the corresponding separation logic rules, under the substitution defined by `with`. The case of `unlock` is the most difficult, and merits its own lemma:

Lemma 2. $\Gamma \vdash \langle \theta, P \rangle \text{unlock } w \langle \theta, Q \rangle \Rightarrow \Gamma \vdash \{P \text{ with } \theta\} \text{unlock } w \{Q \text{ with } \theta\}.$

Proof.

1. The substitution of lock invariants and application of the pre rule leaves the proof obligation $\Gamma \vdash \{w \bullet \xrightarrow{\pi} \text{mklk } \theta[w] * \text{hold } w \text{mklk } \theta[w] * P_0\} \text{unlock } w \{w \bullet \xrightarrow{\pi} \theta[w] * P_{loc}\}.$
2. By the definition of `split_local`, $P_0 \Rightarrow P_{loc} * P_r$, and since θ is unchanged by the application of the rule, $\theta[w]$ already includes the disjunct P_r .
3. Since $P \Rightarrow P \vee Q$ for all P , this implies that $P_r \Rightarrow \theta[w]$.
4. Applying the pre rule with this fact leaves the premise $\Gamma \vdash \{w \bullet \xrightarrow{\pi} \text{mklk } \theta[w] * \text{hold } w \text{mklk } \theta[w] * P_{loc} * \theta[w]\} \text{unlock } w \{w \bullet \xrightarrow{\pi} \theta[w] * P_{loc}\}.$
5. Applying the frame rule reduces this to $\Gamma \vdash \{w \bullet \xrightarrow{\pi} \text{mklk } \theta[w] * \text{hold } w \text{mklk } \theta[w] * \theta[w]\} \text{unlock } w \{w \bullet \xrightarrow{\pi} \theta[w]\}.$
6. By the definition of `mklk`, $\text{hold } w \text{mklk } \theta[w] * \theta[w] \Leftrightarrow \text{mklk } \theta[w]$, so this is equivalent to $\Gamma \vdash \{w \bullet \xrightarrow{\pi} \text{mklk } \theta[w] * \text{mklk } \theta[w]\} \text{unlock } w \{w \bullet \xrightarrow{\pi} \theta[w]\}.$
7. This is true by the first separation logic rule for `unlock`. □

Finally, the analysis rule for `fork` leads to the corresponding separation logic rule through the application of the global rule, which states that $\Gamma \vdash \{P * \Gamma\} s \{Q * \Gamma\} \Rightarrow \Gamma \vdash \{P\} s \{Q\}$ (this is true by the definition of the global context Γ). Thus, for all statements ss , $\Gamma \vdash \langle \theta, P \rangle ss \langle \theta, Q \rangle$ implies $\Gamma \vdash \{P \text{ with } \theta\} ss \text{ with } \theta \{Q \text{ with } \theta\}$, and the final iteration of the sequential analysis on a thread can be seen as the application of the rules of CCm separation logic.

This analysis is used to create the full thread-modular shape analysis by applying it iteratively over the set of threads in the program as described above. The starting state for `main` is **true**, and the starting states for all other threads are defined by their preconditions. An iteration of the

sequential analysis on `main` is complete when $\Gamma \vdash \langle \theta_P, \mathbf{true} \rangle \text{main} \langle \theta_Q, Q \rangle$ has been reached for some Q . Similarly, an iteration on a thread with name n and body t is complete when $\Gamma \vdash \langle \theta_P, R \rangle t \langle \theta_Q, Q \rangle$ is reached for some Q , where R is the precondition for n (i.e., $\Gamma(n) = n : \{R\}$). An iteration over a thread t is said to discover no new information if the algorithm determines that $\Gamma \vdash \langle \theta, P \rangle t \langle \theta, Q \rangle$ for some P and Q . Thus, the thread-modular analysis terminates when $\Gamma \vdash \langle \theta, P \rangle \text{ss} \langle \theta, Q \rangle$ is reached for `main` and each thread in the program in an iteration.

The analysis is a useful tool if it is sound, that is, if a program using the lock invariants produced by the analysis is memory-safe and free from race conditions. Gotsman et al.[3] provide a proof of soundness for the general thread-modular shape analysis framework, using the concepts of lattice theory. However, in keeping with the reinterpretation of the algorithm in the terms of separation logic, this particular instance of the algorithm may be proved sound more concisely by taking advantage of separation logic's soundness. Recall that the soundness of separation logic guarantees that if $\{\mathbf{true}\}p\{\mathbf{true}\}$ can be proven for a program p , then p is memory-safe and free from race conditions. More specifically, a concurrent program can be divided into a main function and a list of threads. Then such a program is safe if two properties hold. First, $\Gamma \vdash \{\mathbf{true}\}\text{call main}()\{\mathbf{true}\}$ must hold for the main function. This is provable only if the global context Γ contains some function assertion $t : \{R\}\{S\}$ for each forked thread. Then the second property is that for each thread with name n and body t in the list of threads, $\Gamma(n) = n : \{R\}\{S\} \Rightarrow \Gamma \vdash \{R\}t\{S\}$. Only if these two properties hold is the program guaranteed to be memory-safe and free from race conditions.

Thus, the algorithm can be proved sound by showing that these two properties hold on the programs it analyzes. Let $p = (\text{main}, \bar{n}, \bar{t})$ designate the program p consisting of main function `main` and threads with names \bar{n} and bodies \bar{t} . Define the judgement $\Gamma \vdash \text{tmsa } p = \theta$ to mean “the thread-modular shape analysis on the program P , run with global context Γ , produces a set of approximated lock invariants θ ”. Then the properties of soundness can be written as follows:

Theorem 4.2. $\forall \Gamma, p. \Gamma \vdash \text{tmsa } p = \theta \wedge p = (\text{main}, \bar{n}, \bar{t}) \Rightarrow (\Gamma \vdash \{\mathbf{true}\}\text{main with } \theta\{\mathbf{true}\} \wedge \forall (n_i, t_i) \text{ in } \bar{n}, \bar{t}. \Gamma(n_i) = n_i : \{R\} \Rightarrow \Gamma \vdash \{R \text{ with } \theta\}t_i \text{ with } \theta\{\mathbf{true}\}).$

Proof.

1. If the algorithm completed on p , then in the last iteration, the sequential analysis ran on `main`

and each thread (n_i, t_i) without discovering any new lock invariants, so by the definition of the analysis, $\Gamma \vdash \langle \theta, \mathbf{true} \rangle \text{main} \langle \theta, Q \rangle$ for some Q , and for each thread (n_i, t_i) , $\Gamma \vdash \langle \theta, R \rangle t_i \langle \theta, Q \rangle$ for some Q .

2. By theorem 4.1, this means that $\Gamma \vdash \{\mathbf{true}\} \text{main}$ with $\theta\{Q \text{ with } \theta\}$ for some Q and for each (n_i, t_i) , $\Gamma \vdash \{R \text{ with } \theta\} t$ with $\theta\{Q \text{ with } \theta\}$ for some Q .
3. $Q \Rightarrow \mathbf{true}$ for all Q , so by the post rule, the above implies that $\Gamma \vdash \{\mathbf{true}\} \text{main}$ with $\theta\{\mathbf{true}\}$, and similarly, for each thread, $\Gamma \vdash \{R \text{ with } \theta\} t$ with $\theta\{\mathbf{true}\}$. \square

Thus, this particular instantiation of the thread-modular shape analysis algorithm is sound as long as CCm separation logic is sound.

5 The Analysis in Coq

The thread-modular shape analysis forms a tool for transforming simple concurrent C-like programs without lock invariants into valid CCm. Since CCm is defined and implemented in the Coq proof assistant, the simplest way to create a machine-checked version of the analysis is to implement it in Coq as well. One of the fundamental constructs of Coq is the inductive definition, and so the simple assertion and statement types described above are easily transferred to Coq. A translation function from simple assertions to the corresponding expressions in full separation logic is similarly straightforward. The analysis itself is an inductive relation defined by the rules of figures 8 and 9; it holds when these rules can be applied to form a judgement $\Gamma \vdash \langle \theta_P, P \rangle ss \langle \theta_Q, Q \rangle$. Since the application of the rules in the analysis is directed by the structure of the statement ss , it should be possible to build an algorithmic function (Coq `Fixpoint`) that follows the semantics of the relation and completely models the behavior of the algorithm.

The proof of soundness for the algorithm described above may also be translatable to a proof in Coq, using the built-in tactics along with the special CCm tactics described in section 3. However, a more indirect approach results in both a simpler proof of soundness and improved guarantees of safety. As shown in theorem 4.1, the final pass of the analysis can be seen as the syntax-directed application of certain facts of separation logic, derived from the base axioms to match the

Figure 10 The rules of the verification function part 1: sequential statements

$$\begin{array}{c}
\frac{\text{pure } e \quad \text{indec } w e \quad P \Rightarrow \text{defined}(e) * R \quad \text{imsv}(w := e) R}{\Gamma \vdash \{P\} w := e \{ \text{defined}(e) * w = e * R \}} \\
\\
\frac{\text{pure } e' \quad \text{indec } w e \quad \text{indec } w e' \quad P \Rightarrow e \mapsto e' * R \quad \text{imsv}(w := *(e)) R}{\Gamma \vdash \{P\} w := *(e) \{ e \mapsto e' * w = e' * R \}} \\
\\
\frac{\text{pure } e \quad \text{pure } e' \quad P \Rightarrow e \mapsto e' * \text{defined}(e_1) * R \quad \text{imsv}(*(e) := e_1) R}{\Gamma \vdash \{P\} *(e) := e_1 \{ e \mapsto e_1 * R \}} \\
\\
\frac{}{\Gamma \vdash \{P\} \text{skip} \{P\}} \qquad \frac{}{\Gamma \vdash \{P\} \text{return} () \{ \text{false} \}} \\
\\
\frac{\Gamma \vdash \{P\}_{s_1} \{Q\} \quad \Gamma \vdash \{P\}_{s_2} \{Q\} \quad \text{pure } e}{\Gamma \vdash \{P\} \text{if } e \text{ then } s_1 \text{ else } s_2 \{Q\}} \qquad \frac{\Gamma \vdash \{P\}_{s_1} \{Q\} \quad \Gamma \vdash \{Q\}_{s_2} \{Q'\}}{\Gamma \vdash \{P\}_{s_1; s_2} \{Q'\}} \\
\\
\frac{P \Rightarrow \mathbf{emp} * R \quad \text{imsv}(w := \text{malloc_and_zero}(n)) R \quad n > 0}{\Gamma \vdash \langle P \rangle w := \text{malloc_and_zero}(n) \langle w \mapsto 0 * (w + 1) \mapsto 0 * \dots * (w + n - 1) \mapsto 0 * R \rangle}
\end{array}$$

structure of the analysis. These facts can easily be expressed and proved correct in Coq using the implementation of CCm separation logic in section 3. Then, as long as the premises of these facts can be checked algorithmically, a function that applies the rules deterministically to a CCm program can be created and proved sound with respect to CCm separation logic, without involving the simple statement type or the semantics of the analysis. As mentioned in section 4, the truth of most of the premises can be determined by case analysis, and the implication algorithm can provide decisions on the truth of premises of the form $P \Rightarrow Q$. The construction of subsidiary checker functions to automate the case analyses, along with a Coq version of the implication algorithm, allows the creation of a proved-correct Coq verification function for the output of the analysis. Rather than prove the analysis sound in Coq, I take advantage of the correspondence between the analysis and separation logic to produce a verification function which both follows the structure of the analysis and can be easily proved correct from its definition.

Figures 10 and 11 show the facts of separation logic used by the verification function. These correspond directly to the analysis rules of figures 8 and 9 respectively, with the incorporation of the frame rule into each other rule¹¹. The proof of theorem 4.1 outlines the derivation of each of

¹¹In many cases, the premise $\text{imsv } s R$ is trivial, since only a few statements modify the values of variables. It is

Figure 11 The rules of the verification function part 2: while and concurrent statements

$$\begin{array}{c}
\frac{\Gamma \vdash \{P\}s\{P\}}{\Gamma \vdash \{P\}\text{while } e s\{P\}} \qquad \frac{\Gamma \vdash \{P\}s\{Q\} \quad \Gamma \vdash \{Q\}\text{while } e s\{Q'\}}{\Gamma \vdash \{P\}\text{while } e s\{Q'\}} \\
\\
\frac{P \Rightarrow w \mapsto 0 * R \quad \text{imsv}(\text{make_lock } w \text{ mklk } I) R}{\Gamma \vdash \{P\}\text{make_lock } w (\text{mklk } I)\{e \xrightarrow{\pi_{\text{lock}}} (\text{mklk } I) * \text{hold } w (\text{mklk } I) * R\}} \\
\\
\frac{P \Rightarrow w \xrightarrow{\pi_{\text{lock}}} A * \text{hold } w A * R \quad \text{imsv}(\text{free_lock } w) R}{\Gamma \vdash \{P\}\text{free_lock } w\{w \mapsto 0 * R\}} \\
\\
\frac{P \Rightarrow w \xrightarrow{\pi} A * R \quad \text{imsv}(\text{lock } w) R}{\Gamma \vdash \{P\}\text{lock } w\{w \xrightarrow{\pi} A * A * R\}} \qquad \frac{P \Rightarrow w \xrightarrow{\pi} A * A * R \quad \text{imsv}(\text{unlock } w) R}{\Gamma \vdash \{P\}\text{unlock } w\{w \xrightarrow{\pi} A * R\}} \\
\\
\frac{\Gamma(e) = e : \{A\} \quad P \Rightarrow A(\vec{w}) * R \quad \text{imsv}(\text{fork } e \vec{w}) R}{\Gamma \vdash \{P\}\text{fork } e \vec{w}\{R\}}
\end{array}$$

these facts from the axioms of concurrent separation logic. The verification function is then simply the syntax-directed application of these rules, starting with the precondition **true**; its soundness reduces to the soundness of each rule, which in turn reduces to the soundness of separation logic. If at any point the premises of a rule cannot be proved, or the statement under consideration does not match one of the rules, the verification fails. The result of the function is a proof of safety for a CCm program, and by the results of theorem 4.1, we expect that verification will succeed on any output of the thread-modular shape analysis, since the verification function effectively reproduces the final iteration of the analysis.

This two-part approach at first seems counterintuitive – after all, if the analysis can be proven sound, verifying the results seems unnecessary. However, the approach provides several notable benefits. First, the domain of the machine-checked proof involved is reduced from the inner workings of the analysis to the familiar ground of CCm separation logic; it is easy to show that the verification function produces valid results, since it consists entirely of the application of separation logic rules. Second, the verification function provides the user with both the proved-safe program and the proof itself, the correctness of which is guaranteed by the proof of soundness of CCm separation logic.

included here to make clear the derivation of each fact from the frame rule.

Finally, if the analysis results in a program that is not safe, through a hole in the informal proof or a flaw in the formalization or implementation, the verification function will fail and the user will not be presented with spurious results. Thus, the two-part model reduces the size of the trusted code base on which the validity of the results depends. The increased simplicity and security of the two-part model may be generalizable to various other machine verification problems, and a more detailed examination of the paradigm is a topic for further research.

6 Conclusions

Program analysis tools implemented in a theorem prover can benefit from both increased automation and machine-checked guarantees of correctness. To this end, the CompCert project [9] led to the development of C Minor, an intermediate language completely defined in the Coq proof assistant. Appel and Blazy [2] adapted the formalism of separation logic to C_m, providing a tool for proving the correctness and memory safety of C-like programs. By extending this base implementation with the syntax and semantics of Concurrent C Minor defined by Hobor, Appel, and Nardelli [5], I have defined a framework for the semi-automated construction of proofs for programs using a realistic model of concurrency. Using this framework, I demonstrated the correctness of a sample concurrent program in the machine-checked environment of Coq. This is a step towards the complete implementation of CC_m in Coq, which will ultimately allow the verification of real-world concurrent programs written in high-level languages such as C and Java.

Next, I explained the application of the thread-modular shape analysis framework proposed by Gotsman et al.[3] for automatically inferring lock invariants to CC_m. In basic CC_m, the lock creation statement `make_lock` brings separation logic assertions into the program code, requiring users to understand both the language and the logic to create concurrent programs. I defined a variant of CC_m without declared lock invariants on which the analysis can operate, and reinterpreted the analysis in terms of the rules of CC_m separation logic. This allowed a simplified proof of soundness for the analysis based on the soundness of separation logic, as well as the construction of a verification algorithm for the output of the analysis. Both the analysis and the verification algorithm are implemented in Coq, forming a machine-checked function from the source language

of the analysis to a corresponding CCm program accompanied by a proof of safety. This two-part model led to both a simplified proof of correctness and a reduction in the size of the trusted portion of the algorithm. These benefits may be generalizable to a larger set of static analyses, and the model merits further investigation.

Acknowledgments. I would like to thank Andrew Appel for all his advice and guidance in both this paper and the associated Coq development.

References

- [1] A. W. Appel. Tactics for Separation Logic. 2006.
- [2] A. W. Appel and S. Blazy. Separation Logic for Small-step C Minor. In *TPHOLs 2007: 20th International Conference on Theorem Proving in Higher-Order Logics*, September 2007.
- [3] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *Proceedings of the 2007 ACM Conference on Programming Languages Design and Implementation (PLDI'07)*, pp. 266-277, 2007.
- [4] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(583):576-580, 1969.
- [5] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. *European Symposium on Programming (ESOP)*, April 2008. Extended version with appendix.
- [6] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375(1-3):271-307, May 2007.
- [7] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319-340, 1976.
- [8] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pp. 55-74, 2002.

- [9] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the POPL 2006 symposium*, pp. 42-54, 2006.
- [10] “The Coq proof assistant”. <<http://coq.inria.fr>>, accessed May 2008.