RoosterFS
CS260 Project Writeup

Rob Bowden, David Holland, Eric Lu

May 8, 2017

Abstract

Formal verification of file systems is critical to ensure data integrity and file system consistency, both during regular operation and in the event of an unexpected crash. While a handful of verified file systems have been developed in the past few years, none of them support fully concurrent file system write operations. In this work, we introduce concurrent crash Hoare logic (CCHL) as a means of reasoning about system correctness in the presence of of both concurrency and crashes. We have implemented the logic in Coq proof assistant and have begun proving it sound according to the semantics of an imperative language with locks, heap, and disk operations. We intend to use this language to implement a fully concurrent file system that supports buffered writes and an asynchronous disk, and we will then use concurrent crash Hoare logic to prove this file system correct according to specifications that we have already begun forming.

1 Introduction

File systems that store data persistently are a critical component of operating systems. As such, their correctness is fundamental to the integrity and reliability of operating systems and thus of all higher-level application software running on those systems. Data loss and corruption bugs are regularly found in existing file systems, even widely used ones [17], and loss of persistent data is inherently more serious and has broader ramifications than kernel bugs that merely require rebooting. Therefore, proving the functional correctness of file systems is highly desirable.

Formal verification of software systems is already a nontrivial task in general[1], but for file systems an additional difficulty arises: file systems have persistent state on disk, and that state must remain consistent even in the face of system crashes. Strategies like journaling and soft updates are used to keep crashes from causing corruption and data loss, but in the absence of verification bugs can, have, and will continue to defeat these techniques.

To reason about the correctness of a file system one must reason both about transient in-memory state and persistent on-disk state. Crashes (arising for example from power failures, hardware faults, or kernel panics in unverified components) erase the transient state. This includes data in the kernel's buffer cache and, with modern disks, potentially also unwritten data cached within the disk itself. This can happen at any time and after any step of execution; unlike when reasoning about, for example, interrupts (already difficult enough), it is not possible to block crashes in order to execute a critical section.

Reasoning about state in the presence of uncontrollable external interrupts needs specific support in the program logic. This support is primarily useful for reasoning about persistent state in the presence of system crashes, but it can be

[1] "seL4 took *twenty person-years*!!!"

used for other things. For example, one might use it to reason about (transient) program state in the presence of imprecise floating point exceptions.

After a crash, the system reboots, and runs a procedure known as *recovery*, whose goal is to examine the on-disk state and correct it as necessary to resume normal functioning. This also requires support in the program logic, because (in general) the recovery procedure starts with the persistent disk state in an otherwise illegal intermediate condition that does not meet the preconditions for any ordinary operation. The program logic must splice the conditions enforced on the persistent state during operation ("crash conditions") to the precondition of the recovery procedure, and the postcondition of the recovery procedure ("recovery conditions") to the preconditions for normal operations. The full specification of a file system includes recovery conditions that describe the guarantees made to applications about the state of files and other file system objects after a crash occurs.[2].

Chen et al. [5] introduced the concept of *crash Hoare logic* for reasoning about persistent state in the presence of arbitrary crashes. This work introduced the concept of crash conditions, but it does not have explicit recovery conditions: their file system (FSCQ) is entirely synchronous (so no data is buffered past an operation completing) and their program logic does not support concurrency, so only one operation can be in progress at a time and their recovery scheme either fully completes or fully aborts it.

While other file systems besides FSCQ have been formally verified as crash-safe, none to our knowledge support concurrent execution and many do not support write-back caching. The ultimate intended goal of this project was to develop a file system, or at least a file system model, that supports both concurrent execution and write-back caching, and prove it correct in the presence of crashes with a relatively strict set of post-crash guarantees. The actual contribution of this project so far is, first, to extend crash Hoare logic into *concurrent* crash Hoare logic, to allow for reasoning about concurrent operations even in the face of crashes, and second, to take first steps at proving the logic sound and provide some simple example code demonstrating that the logic is also useful. Ultimately, after having proven the logic sound, we intend to use the concurrent crash Hoare logic to build a fully concurrent file system.

In the remainder of this paper we first discuss related work (Section 2), then introduce the language and the small-step operational semantics we use as a basis (Section 3). In Section 4 we describe the program logic. Then in Section 5 we discuss our implementation, example, and an evaluation of the usefulness of the logic. Finally we discuss future work (Section 6) and in Section 7 conclude.

2 Related Work

2.1 Separation logic

Separation logic [14] introduced an intuitive way to *reason locally* about programs that update heap values. The program heap is a mapping from addresses to values. A judgement on the heap takes the form $h \vDash P$. The judgement holds if h satisfies the assertion P. An example assertion is $10 \mapsto 20$, which asserts that address 10 contains (maps to) 20 (in other words, $h(10) = 20$).

In the classical formulation, the assertion P is only true if h is *exactly* the heap described by P, and nothing more. (In the intuitionistic formulation, satisfying P means h is a superset of the heap described by P, as opposed to exactly that heap. This is useful for reasoning about programs that have automatic garbage collection as opposed to manual memory management.)

We have a binary separating conjunction operator $(*)$ and a unit emp (which describes the

[2] That said, we have not yet gotten to the point of formalizing recovery conditions for a concurrent file system in CCHL

empty heap) with the following properties[3]:

$$h * emp \vDash P \to h \vDash P$$

$$\begin{aligned} h \vDash P * R \leftrightarrow h_1 \vDash P \wedge \\ h_2 \vDash R \wedge \\ h = h_1 \cup h_2 \wedge \\ h_1 \perp h_2 \end{aligned}$$

where $h_1 \perp h_2$ is true if and only if h_1 and h_2 are disjoint heaps.

Using the separating conjunction, we can create a judgement of the form:

$$h \vDash (10 \mapsto 20) * (15 \mapsto 5)$$

This judgement holds if and only if h is exactly the heap with two allocated addresses, 10 pointing to 20 and 15 pointing to 5.

Separation logic extends traditional Hoare logic [8] with the frame rule:

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}$$

Intuitively, if we think of C as a procedure call that has P as its heap pre-condition and Q as the resulting heap, then we can call C by "framing out" the rest of the heap (R) that C doesn't touch, and then framing it back in after the procedure call.

2.2 Concurrent Separation Logic

The ability to reason locally about the heap naturally led to attempting to reason about shared state in concurrent programs. This gave rise to concurrent separation logic [13]. Given a binary operator $C_1 \mid\mid C_2$ that runs commands C_1 and C_2 in parallel, we have the rule:

$$\frac{\{P\}\, C_1\, \{Q\} \quad \{P'\}\, C_2\, \{Q'\}}{\{P * P'\}\, C_1 \mid\mid C_2\, \{Q * Q'\}}$$

[3] Traditional separation logic also has a separating implication operator ($-\!\!*$), but our logic does not use it.

We can freely run C_1 and C_2 concurrently if the heaps they use are disjoint, because they cannot interact with one another. But this rule alone cannot be used to model programs that communicate or share state.

The original concurrent separation logic had an atomic *with* r *when* b *do* c control flow operator, and associated logic rule. This runs command c atomically (using an implicit lock associated with r), spinning until condition b becomes true. We do not use this model. Instead, we model locks explicitly, as was done in Hobor et al.'s work on formulating a concurrent separation logic for C [9]. As such, we have rules for both acquiring and releasing a lock. Loosely, the rules look like:

$$\frac{}{\{\ell \mapsto R\}\, \texttt{getlock}(\ell)\, \{(\ell \mapsto R) * R\}}$$

$$\frac{}{\{(\ell \mapsto R) * R\}\, \texttt{putlock}(\ell)\, \{\ell \mapsto R\}}$$

Here, R represents an assertion (a "predicate on the heap"). What exactly it means for a predicate (something meant to be discussed only by the logic) to live in the heap and be pointed to is somewhat problematic. Our logic currently avoids doing this by not allowing for dynamic lock creation and deletion. But that issue aside, intuitively, acquiring a lock allows us to access new heap addresses referenced in R that the lock was protecting.

For example, R might look like $\exists x, 10 \mapsto x \wedge \texttt{isEven}(x)$. After acquiring the lock, we now can read address 10. We do not know what x is until we actually perform the read, but we do know that it is even. (This is all the information we will ever have when trying to prove something in the logic, since we cannot directly read the address to get a concrete value in the logic.)

We can change x arbitrarily while we hold the lock, but when we release the lock, we must guarantee x is once again even. Otherwise we do not meet the pre-condition of putlock, which includes R.

2.3 Other Logics

In parallel with concurrent separation logic, other methods arose to reason about concurrent programs. For example, rely-guarantee [10] explicitly models interference (whereas concurrent separation logic is premised on the assumption that most programs rarely interfere). The Views metaframework [6] is a general framework that can be used to prove the soundness of various compositional reasoning strategies for concurrent programs, and has been shown to generalize both rely-guarantee and concurrent separation logic.

Most similar to our own efforts, Ntzik et al. extend the Views framework to allow for concurrent fault-tolerant resource reasoning [12]. They split pre-conditions and post-conditions into separate volatile (memory-related) and durable (disk-related) conditions, with separate, framerulable crash invariants. They use their logic to reason about ARIES [12], a write-ahead logging recovery algorithm. However, their logic rules differ from our own, aligning much more closely with regular concurrent separation logic. Furthermore, they do not demonstrate an actual implementation of their logic or proofs in code. It is not immediately clear to us how their rules relate to our own, and we are interested in looking into the relationship further.

The Verified Software Toolchain [2] is an ongoing effort to allow for functional verification of real C code in Coq that can be immediately compiled by the CompCert [11] verified C compiler to machine code. While their separation logic framework has been used to prove relatively substantial programs written in C, their fledgling concurrent separation logic does not yet allow for reasoning about something as complex as a file system. Our concurrent separation logic does not need to concern itself with the full semantics of the (supported subset of) the C language, nor does it need to adhere to any preexisting semantics as VST must do with CompCert. Nevertheless, we were able to reuse VST's underlying separation algebra machinery in the implementation of our logic.

In extending a file system to allow for concurrency and buffered writes, we run into a fundamental issue with concurrent separation logic: proving the absence of low-level races is relatively easy, but reasoning about functional correctness becomes difficult. With multiple concurrent operations and updates being buffered, the state of the file system before and after any one operation is no longer readily expressible. Blom et al. argue that this problem may be handled with the use of traces [3], in a manner similar to how distributed systems implemented using the Verdi framework prove functional correctness [16]. This is the approach that VST is following in the beginnings of their concurrent separation logic, and it is the one we take in our specifications.

Crash Hoare logic [5] is an extension of Hoare logic that, in addition to the regular pre- and post-conditions, has an additional crash condition that must hold *at every step* of the execution. If a given procedure satisfies some crash condition C, then in the event of a crash, we are guaranteed that C accurately describes the state of the world just before the crash. Crash Hoare logic is used in the correctness proofs of the FSCQ [5] file system.

2.4 Verified File Systems

FSCQ [5], written in the Coq proof assistant, is the first file system proven to meet a specification that includes crashes, using the aforementioned crash Hoare logic. The logic makes use of separation logic to ensure that procedures modify only the intended data. Using the separating conjunction ($*$), a specification for the writeblock call can be written along the lines of:

$$\{(\mathit{diskblock}\ b \mapsto\ ?) * \mathit{otherblocks}\}$$
$$\texttt{writeblock}(b, 10)$$
$$\{(\mathit{diskblock}\ b \mapsto 10) * \mathit{otherblocks}\}$$

In the above, the appearance of $* \ \mathit{otherblocks}$ on both sides of the Hoare triple indicates that the write does not touch the other blocks on disk. Without separation logic, a write call could theoretically be allowed to zero out all other blocks on disk and just not mention that fact in its Hoare triple.

Our work is directly inspired by FSCQ's use of crash Hoare logic, but there are three issues with FSCQ that we aim to fix. First, it does not support full concurrency. Chajed [4] extends FSCQ to allow for asynchronous reads: if a file system call needs to read from disk, it issues the read and restarts the entire call, allowing another file system call to proceed while the first waits for the disk read to complete. However, extending FSCQ to fully concurrent file system calls seems to require fundamental changes to its infrastructure.

Second, every write operation to disk requires an explicit disk sync before returning[4]. Writes are not buffered in memory, and the postcondition of a write call would not be able to state anything about the disk containing new data without a sync.

Finally, related strictly to the logic (and not a restriction on the actual implementation), the specification does not seem to be able to make full use of the power of separation logic. In the specification for `writeblock` above, usually separation logic would allow for leaving out $* \, otherblocks$ from the pre- and post-conditions, and it would be up to a caller to frame it out before making the call. Our logic aims to fix these three concerns.

Other verified file systems suffer from similar deficiencies, especially with respect to concurrency. Cogent [1] is a domain-specific language that certifiably compiles to C, which can be further compiled to machine code by CompCert. Proofs in Cogent are at a higher level than low-level C code, and thus the verification burden is lowered. Cogent is used to prove (subsets of) two Linux file systems, BilbyFS and ext2, rewritten in the language. However, Cogent's semantics is sequential, and thus while it supports asynchronous I/O it does not handle concurrency.

Yggdrasil [15] is similarly a higher-level toolkit for implementing file systems based on crash refinement. It uses Z3 as its underlying verifier, removing most of the proof burden from the developer. Again, file systems implemented using Yggdrasil are limited to single-threaded code.

[4] In fact, it requires four separate syncs.

3 Language and Semantics

In order to support concurrency we need a language where executions can be finely interleaved, and a small-step semantics so that these interleavings can be formally defined and reasoned about. In order to mechanize the logic and proofs in Coq this means defining an abstract syntax and a semantics, and then writing our concurrent file system code in the abstract syntax. (By comparison, in FSCQ, the only steps involve updating the disk and everything in between is pure-functional Coq code.)

Our language CHIMP (Concurrent/Crash Hoare IMP) is basic IMP extended with a minimal set of features necessary for writing nontrivial concurrent code:

- procedures and procedure calls/returns;
- a `start` operation to fork a new thread;
- local variables declared in procedures;
- `load` and `store` to access the heap;
- `getlock` and `putlock` to handle locks;
- and simple explicit types.

Local variables are private to the procedures they appear in. Forking a new thread always invokes a procedure with a new set of local variables.

Expressions are pure so we can have a big-step semantics for evaluating them. This simplifies a number of things greatly. Both the load and procedure call operations always assign the results to a variable directly; they are commands and not expressions.

Locks can be thought of as safe handles for global shared objects that live in the heap; the logic prevents accessing heap variables without holding the locks that protect them. This is discussed in Section 4.

The disk, or in fact potentially several disks, are treated as additional heaps and accessed via the same load and store operations as inmemory variables. Disk addresses can be protected by locks, but locks may not themselves live on disks.

We do not currently have condition variables. As seen in e.g. STM Haskell [7], condition variables are a performance optimization. (It might be an interesting exercise to formulate a more sophisticated abstract machine with condition variables and an explicit concept of sleeping threads, and prove a refinement.)

3.1 Semantics

As noted above eXpressions are pure (they can only access local variables, which are not shared, and are read-only) so the semantics for expression evaluation can be large-step: an expression just evaluates to a value. (The supply of expressions we support is somewhat limited, because originally program values were arbitrary Coq values and thus any Coq expression could be trivially embedded.)

The rest of the language semantics are written in four layers: one for ordinary commands/statements (including those that access the heap), one for the call and return stack, one for threads (which is currently pass-through but might not be in the future), and one for the complete machine with threads executing in parallel. Lock operations happen at the command level, because (at least for semantic purposes) the lock state, though separate from heap data, is rolled in with the heap. This might or might not have been a mistake. (The semantics of the lock operations have changed repeatedly to suit the needs of the logic.) Starting a new thread is effectively calling a procedure in a new thread context, so it happens mostly at the stack level: a new stack is created and this is then reflected up to the machine level as a new thread. At the machine level, the machine nondeterministically chooses a thread to run and steps it once. Acquiring a lock can only progress if the lock is available, so threads waiting for locks cannot step. If the system deadlocks, which the logic does not currently attempt to prevent, the machine will become unable to step. (This can be construed as a form of crash.)

3.2 Typing

We originally wanted to use arbitrary Coq values as program values, for a wide range of reasons. Doing that while not requiring a separate variable environment for every type requires wrapping up a Coq type and value of that type inside a value object; extracting these and using the values requires proving to Coq that the type wrapped up in a value is the type it ought to be. This in turn requires a set of soundness propositions on the abstract syntax. These propositions were not a type system in the usual sense, but they were structurally equivalent to one. Later on we gave up on using arbitrary Coq values; the existing soundness propositions became very similar ones implementing a simple type system.

The motivation for worrying about typing, and, indeed, proving soundness of the typing in terms of preservation and progress, was not typing per se (largely irrelevant to the project goals) but using the types as a lever for debugging the semantics. The soundness of the *logic* is of critical importance; the logic is sound only with respect to the semantics; and the logic soundness proofs are far more complicated and harder than type system proofs. Problems with the semantics that make the logic proofs just not work can be detected and understood in the context of the type system.

The experience of updating the typing proofs as the abstract syntax and semantics have shifted under the demands of the logic has been valuable in its own right, in a long-term sense.

4 Crash Concurrent Separation Hoare Logic

4.1 Notation

Because our language is imperative, we are able to take some inspiration from the Verified Software Toolchain for the Hoare logic rules. On the one hand, we are able to simplify, since we do not have to support the broader C semantics that VST does. On the other, we need to extend the logic to support both concurrency and crash conditions.

A Hoare judgement for a statement of our language is a septuple of the form:

$$(R^c, R^r); (L^c, L^r) \vdash \{\{CP\}\}\, \{P\}\, s\, \{Q\}\, \{\{CQ\}\}$$

P and Q are the pre- and post-conditions, respectively, of the statement s, as in traditional Hoare logic. CP and CQ, on the other hand, are the crash pre-condition and crash post-condition.

Note that these are the pre-form and post-form of the crash condition (the condition that must be true when crashing), and in particular the "crash post-condition" is not the post-condition of crashing, or of crashing and recovering. This terminology could be construed as confusing.

The crash condition (which in FSCQ appears as a single condition) is a statement about invariants. Every step taken must preserve the crash condition. In particular, the crash pre-condition demands that every step taken in s preserves the invariants in CP. The crash post-condition allows s to demand that every step taken after s completes abides by the invariants in CQ. These are potentially separate (and different) because s might do things that affect what crash invariants must be considered.

(L^c, L^r) and (R^c, R^r) are described in the "Lock rules" subsection (4.2) and and "Procedure rules" subsection (4.5), respectively. We will write these as just L and R except when directly discussing the individual components.

4.2 Lock rules

How can a statement take in an invariant that is supposed to hold at every step of execution, and ultimately require a *different* invariant after it executes? Without any additional infrastructure, it makes sense that s can add further invariants to CP, such that CQ is ultimately a superset of CP. However, the logic can only scalably reason about large programs if we can take invariants *away*.

As described in regular concurrent separation logic, acquiring a lock can pull a resource invariant into the pre-condition of a statement, and releasing a lock can pull a resource invariant out of the post-condition. Analogously, we model lock acquisition to pull a separate resource invariant into the crash pre-condition and lock release to pull it out of the crash post-condition.

We call the resource invariant pulled into the regular pre- and post-conditions a *weak invariant*. The invariant only needs to hold at the time the lock is acquired and when it is released. In between lock acquisition and release, the invariant may be violated. We call the (possibly different) resource invariant pulled into the crash pre- and post-conditions a *strong invariant*. The invariant must be true at the time the lock is acquired; it must be true at the time the lock is released; and it must remain true at each step of execution in between the acquisition and release.

In the Hoare septuple above, L^c is a map from a lock to its associated strong invariant that is pulled into and out of the crash pre-condition and post-condition, and L^r is the traditional map from a lock to its associated weak invariant that is pulled into and out of the regular pre-condition and post-condition. Given all this, the rules for lock acquisition and release[5] become

[5] Notice that we avoid having to discuss the storage of

what is written in Figure 1.

4.3 Frame rules

What happens if we try to acquire two completely independent locks, ℓ_1 and ℓ_2, in sequence? Unfortunately, when we try to acquire ℓ_2, we could have a non-empty crash pre-condition arising from $L^c(\ell_1)$, and as such would not meet the crash pre-condition of `getlock`(ℓ_2).

We would like to use a frame rule to "frame out" $L^c(\ell_1)$ from the crash pre- and post-conditions before calling `getlock`(ℓ_2). Unfortunately, it cannot be as general as the original frame rule of separation logic: here, to frame constraints out of the crash pre- and post-conditions, we must also frame knowledge of the constrained portions of the heap out of the regular pre- and post-conditions. Otherwise, we could frame out a strong invariant and retain the ability to modify the heap in a way that violates the strong invariant. As such, the additional frame rule carries some restrictions, as seen in Figure 2.

Intuitively, if statement s does not rely in any way whatsoever on lock ℓ in order to execute, then we can completely frame everything related to it out of existence (including the lock itself). s cannot possibly violate constraint $L^c(\ell)$, because everything that the acquisition of ℓ pulled in has been framed out.[6]

A more granular rule is expected to remain sound. A strong invariant cannot be violated as long as the resource it constrains is removed from the weak invariant. Then, as long as we simultaneously frame the same referenced addresses and all constraints on those addresses out of both the crash and regular conditions, we do not need to frame *everything* related to the lock out at once. With this, we can combine the two frame rules into one, as seen in Figure 3.

There is one important thing that the weaker (more granular) rule handles that the stronger rule cannot. Consider an example of a lock ℓ_1 protecting a structure that contains another lock ℓ_2 (for example, in a linked list hand-over-hand lock coupling program). The crash pre-condition of `getlock`(ℓ_2) requires everything related to ℓ_1 to have been framed out. But the mapping $a_2 \mapsto \ell_2$, which is a part of the statement's pre-condition, is itself a resource in the invariants of ℓ_1! The weaker frame rule would allow for removal of all *other* resources and constraints, which still allows for acquisition of the lock.

We intend to switch to (a form of) the alternative frame rule, but we currently have implemented the two separate frame rules. See Section 6 for a discussion of why the frame rule (and more fundamentally the shape of our Hoare triple) might still change further, whereas we expect (but have not yet proven) that our two current frame rules are sound.

4.4 Rule of consequence

Ignoring separation logic and crash post-conditions for a moment, if we currently have crash pre-condition $CP' = A \wedge B$, under what circumstances can we execute a statement s with pre-condition CP? We can clearly do so if $CP = CP'$. What if $CP = A$? Then statement s does not adhere to invariant B, and we cannot execute it. On the other hand, if $CP = A \wedge B \wedge C$,

invariants in the heap. Instead, analogously to locks in C, we just need to have a pointer to the lock in our heap in order to acquire and release it.

[6] Note that this implies a constraint on the possible dynamic crash conditions that we can support: a crash condition should *only* constrain the use of the resource that the lock protects. It does not make sense for the acquisition of a lock to insert a strong invariant about some global variable not protected by that lock, since any thread is in danger of breaking that invariant when not holding the lock. Furthermore, it is frustrating that we have to specify the mapping of the lock in the crash conditions. This would appear to imply that this must *always* be the mapping of the lock. We could just indicate that there exists some mapping for the lock, but that would still imply that the lock is always allocated. This is fine in our current world where we do not have dynamic memory or lock operations, but is an issue once those are added into the language. See the future work section for a discussion of our proposed solution to this issue.

$$\frac{}{\forall a, R; (L^c, L^r) \vdash \{\{a \mapsto \ell\}\}\, \{a \mapsto \ell\}\, \texttt{getlock}(\ell)\, \{a \mapsto \ell * L^r(\ell)\}\, \{\{a \mapsto \ell * L^c(\ell)\}\}}$$

$$\frac{}{\forall a, R; (L^c, L^r) \vdash \{\{a \mapsto \ell * L^c(\ell)\}\}\, \{a \mapsto \ell * L^r(\ell)\}\, \texttt{putlock}(\ell)\, \{a \mapsto \ell\}\, \{\{emp\}\}}$$

Figure 1: Rules for getlock and putlock

$$\frac{R; (L^c, L^r) \vdash \{\{CP\}\}\, \{P\}\, s\, \{Q\}\, \{\{CQ\}\}}{\forall a, R; (L^c, L^r) \vdash \{\{CP * L^c(\ell)\}\}\, \{P * a \mapsto \ell * L^r(\ell)\}\, s\, \{Q * a \mapsto \ell * L^r(\ell)\}\, \{\{CQ * L^c(\ell)\}\}}$$

Figure 2: Frame rule for strong invariants

then s adheres to a superset of the invariants that we need, and we can proceed. In general, we can execute s if $CP \to CP'$.

Now focusing on crash post-conditions, we have crash post-condition $CQ' = A \wedge B$, and statement s has crash post-condition CQ. If $CQ = A \wedge B \wedge C$, then s requires that future steps adhere to C, which we do not promise to adhere to. On the other hand, if $CQ = A$, then s requires that all future steps adhere only to A, which is a subset of what we already intended to adhere to after this step. In general, we can execute s if $CQ \to CQ'$.

The full rule of consequence, combined with that of traditional Hoare logic, is as shown in Figure 4. Interestingly, the implications in the crash rules are the *converse* of the implications in the traditional rules!

In traditional Hoare logic, a Hoare triple cannot be established for a sequence where the second statement has a pre-condition of False. (Unless the first statement was able to establish a post-condition of False.) Similarly, having proven a Hoare triple with a post-condition of False means we can sequence with a statement that has any other pre-condition (and thus prove anything), because we can use the rule of consequence to change the post-condition to match whatever pre-condition we like.

What does it mean for a statement's crash pre-condition to be True? By the rule of consequence, it means we cannot establish a Hoare septuple for a sequence to this statement (unless the current crash post-condition happens to also be True). Similarly, a crash pre-condition of True may be established for any statement, since any statement's more precise crash pre-condition may imply True. Intuitively, a True crash pre-condition releases the Hoare septuple's statement from adhering to *any* invariants; a Hoare septuple with a True crash pre-condition does not claim the statement adheres to any invariant.

By similar reasoning, if a statement's crash pre-condition is False, then it supports *all* possible invariants. For example, an operation that does not modify any state can have a crash pre-condition of False.

If a statement's crash pre-condition is emp, then that asserts and requires that its heap is permanently empty. (The only way we could add something to the heap is by acquiring a lock, which would require having access to something on the heap in the first place.) This could be useful for specifying completely pure functions that do not touch the heap.

On the other side of the septuple, what about the crash post-condition (assuming we meet the pre-conditions)? If a statement's crash post-condition is True, then we can sequence to a statement with any crash pre-condition. Again, True implies respect for no invariant, and so

$$\frac{R; L \vdash \{\{CP\}\}\,\{P\}\,s\,\{Q\}\,\{\{CQ\}\}}{R; L \vdash \{\{CP * CF\}\}\,\{P * F\}\,s\,\{Q * F\}\,\{\{CQ * CF\}\}}\ \forall a, (a \mapsto\ ?) \in CF \to (a \mapsto\ ?) \notin P$$

Figure 3: Alternative frame rule

$$\frac{CP \to CP' \quad P' \to P \quad R; L \vdash \{\{CP\}\}\,\{P\}\,s\,\{Q\}\,\{\{CQ\}\} \quad Q \to Q' \quad CQ' \to CQ}{R; L \vdash \{\{CP'\}\}\,\{P'\}\,s\,\{Q'\}\,\{\{CQ'\}\}}$$

Figure 4: Rule of Consequence

if the statement ends without further requirements for respected invariants, we can just add more invariants to get the invariant that the sequenced statement promises to respect.

On the other hand, if the statement's crash post-condition is False, we can never execute another statement (unless the next statement's crash post-condition is also False).

Finally, if the statement's crash post-condition is emp, the heap must remain empty after the statement returns. This is not strictly permanent, though, since we could always have framed out part of the heap prior to executing the statement, and frame it back in after.

Interestingly, even applying the rule of consequence seems to work in the opposite direction for crash conditions. Take the rule for the `skip` statement as an example:

$$\frac{}{R; L \vdash \{\{CP\}\}\,\{P\}\,\texttt{skip}\,\{P\}\,\{\{CP\}\}}$$

If the current state of our judgement looks like:

$$\frac{}{R; L \vdash \{\{\mathit{True}\}\}\,\{\mathit{False}\}\,\texttt{skip}\,\{Q\}\,\{\{CQ\}\}}$$

(which is a very possible state after applying the Return Rule, described in the next subsection), how can we use the skip rule to prove this judgement? As described earlier, we can use a crash pre-condition of True to reach any crash condition, and a regular pre-condition of False to reach any regular condition.

In this case, we can weaken our regular precondtion from False to Q using the rule of consequence. But to make the crash conditions match, we go in the opposite direction. We use the rule of consequence on the crash *post-condition* to bring the state from CQ to True. Now we can apply the skip rule with $P = Q$ and $CP = True$, and we are all set.

4.5 Procedure rules

In our language, a statement can call another procedure and can return from within a procedure. Ultimately, a procedure judgement is of roughly the form:

$$L \vdash \{\{CP\}\}\,\{P\}\,\texttt{proc}(s, e)\,\{Q\}\,\{\{CQ\}\}$$

This is to say, a procedure with body s with takes e as an argument must satisfy the specified pre- and post-conditions given the (global) lock invariants. Notably missing is the return condition, R, found in the statement judgements. In order to prove the procedure judgement sound, we just need to prove that statement s satisfies:

$$(CQ(e), Q(e)); L \vdash \{\{CP(e)\}\}\,\{P(e)\}\,s\,\{\mathit{False}\}\,\{\{\mathit{True}\}\}$$

First, the procedure's conditions are all parameterized by the input argument. Second, the procedure's post-conditions are pulled into the *return* conditions of the body, the post-condition is False, and the post crash-condition is True. False and True enforce that the body must always have a return statement, which, as we will see, is a rule that can satisfy those constraints (after hitting a return, we should be able to trivially prove the rest of the procedure, which is often just “skip”).

The return rule takes the form:

$$\frac{R^c(e) \rightarrow CP \quad P \rightarrow R^r(e)}{(R^c, R^r); L \vdash \{\{CP\}\}\,\{P\}\,\texttt{return}\; e\,\{\mathit{False}\}\,\{\{\mathit{True}\}\}}$$

(R^c, R^r) came directly from the crash and regular post-conditions of the procedure. They were already parameterized by the input argument, and are now further parameterized by the return value (such that the input argument and the return value can be related in the pre- and post-conditions for the procedure). If the current crash condition implies the procedure’s crash post-condition and the current regular condition implies the procedure’s post-condition (again, the crash condition is the reverse of the regular condition), then the judgement is sound.

If the body of the procedure is proven sound according to the procedure’s pre- and post-conditions, then in proving the correctness of any other procedure that calls the proven procedure, assuming that the current pre-conditions match those of the proven procedure with the input argument passed in, we can simply drop in the procedure’s post-conditions, parameterized with both the input argument and the return value.

4.6 Crash-preserving rules

Ultimately, the only statement of our language whose corresponding Hoare rule needs to be checked against the current crash condition is “store”. In our model, the disk is represented as a parallel version of the heap that can ultimately hold arbitrary values. A modern disk’s internal write-back cache can be represented, for example, by storing lists at each disk address, where the head of the list gives the most recent value written to disk, and the tail represents potential values that could appear on disk following a crash if the disk is not explicitly synced. The list can be trimmed to the single head element on a disk-level sync. (This could also be used to handle the operating system buffer cache, but that is more appropriately modeled explicitly.)

On a store, in addition to the regular concurrent separation logic rules requiring that the address is currently mapped in our accessible heap, we just need to be sure that the new value being stored at the heap address does not violate any strong invariants associated with that heap address in our current crash condition.

4.7 Soundness

We would like to demonstrate that our logic accurately describes the properties of programs under the semantics we have given. At a high level, soundness may be stated as: if we were able to prove a Hoare septuple about a program, then if the program is executed under the semantics, it will preserve the stated lock invariants, adhere to the stated crash pre-condition, and establish the stated normal post-condition given that the initial state of the heap and local variables satisfies the pre-condition. Note that the crash post-condition is not implicated in this high-level description of soundness: we do not check explicitly that requiring adherence to the crash post-condition that was proven in a Hoare septuple for some statement s on the part of the crash pre-conditions of statements sequenced onto s suffices for continued maintenance of the lock invariants of the Hoare septuple demonstrated for s. This is because showing that proving a Hoare septuple is enough to guarantee a statement’s adherence to the stated lock invariants means that the logic is already properly using the crash post-condition in the inference rule for sequencing to account for which statements may or may not be sequenced.

As of this writing, we are able to state a partial soundness condition which guarantees that a heap that contains only a finite list of locks adheres to the lock invariants for each of these locks. Intuitively, this is not an onerous restriction: a program certainly should only ever have a finite list of locks. This is also not a reversion to the pre-declared finite list of locks maintained by traditional concurrent separation logic, because these locks may still inhabit the heap, and the logic might account for growing and shrinking the list of extant locks when lock allocation and destruction are added to the language and logic. The soundness statement is partial because the logic currently underconstrains the heap, such that it is possible that an enumeration of the heaplets pointed to by each lock does not terminate. Future work may lift this restriction by moving the finite lock list stipulation from the partial soundness statement into the logic.

Proof of soundness for traditional concurrent separation logic proceeds by strengthening the high-level violation-freeness soundness statement into an invariant which is demonstrated to be preserved by the execution semantics. We expect that our proof of soundness will proceed in similar manner. We expect the resulting soundness invariant to be as follows:

We will say that a heap is good with respect to this Hoare septuple and a state of the local variables if it meets the following criteria:

1. it adheres to both the crash and normal pre-conditions of the Hoare septuple (given the state of the local variables as a parameter);
2. it contains a finite number of locks, which may be given by a list;
3. it adheres to the lock invariants of all the *unheld* locks in this list.

Suppose a statement s satisfies a Hoare septuple. Suppose that we then run s some number of steps starting from a good heap state h (and concomitant local variables ρ), reaching a new state s', h', ρ'. Then we stipulate that there exist new crash and normal pre-conditions such that a Hoare septuple may be proven for s'. The new heap state h' should be good with respect to ρ'. The new crash pre-condition should additionally constrain the heap state to adhere to the lock invariants of the *held* locks. (We do not have this as an assumption of the original heap state because we should not be able to prove a Hoare septuple where the preconditions already imply a violation of the lock invariants.)

We have not yet completed a proof of this soundness statement. However, the given expected form of the statement explains some features of the logic. For instance, it explains the converse behavior of the rule of consequence for crashes. The stipulation that the crash pre-condition CP should constrain the heap state to adhere to the lock invariants of held locks may be written as: $\neg(CP \implies \neg L^C)$; that is, we cannot derive a violation of L^C given that CP holds. If $CP \implies CP'$ then CP' will also meet this stipulation! Similarly, it indicates that the normal pre-condition should not be strengthened using the rule of consequence to the point that it can prove $\neg CP$. (Adherence of the normal pre-condition to the conditions set by the crash pre-condition is given as a condition on the heap $P \wedge CP$, but under the rule of inference "material implication," we can see that $P \wedge CP$ corresponds to $\neg(P \implies \neg CP)$, similar to the form of "crash pre-condition adherence to lock invariants" stated above.) Since the assumptions and conclusions made about the heap in the soundness execution invariant both constrain the heap to obey $P \wedge CP$, we expect that P and CP must share a heap footprint. This explains the form of the more granular frame rule suggested, which by ensuring that the same address is framed out of both the normal and crash pre-conditions would make certain that the footprints of the crash and normal pre-conditions are kept in sync.

5 Implementation

Our language, semantics, and logic have been formalized in the Coq proof assistant. There is also a type system for the language, as discussed above.

As of this writing we have partial proofs of the soundness of the type system (which serves mostly to validate the semantics), and a proof outline for the soundness of the logic with respect to the semantics, as described above.

We have used CCHL to (mostly) verify a simple example of a lock protecting a counter. The lock's strong invariant requires that the counter is always even (whenever we store into the counter, this property is verified). The lock's weak invariant requires that the counter is exactly 4 (only on lock release is this property reverified). The program acquires the lock, reads in the counter's current value, adds two to that value and stores it back into the counter (thus respecting the strong invariant but violating the weak invariant), subtracts two from the new value and stores it back into the counter (again respecting the strong invariant and restoring the weak invariant), and then releases the lock (which succeeds, since the weak invariant holds again).

Proving this simple example correct was surprisingly nontrivial. First, the logic had to be adjusted several times, not to fix bugs, but rather to try to write the rules in such a way that made it easy for Coq to handle. Several rules create existential variables because of the possibility of a program variable getting overwritten, and so the example was rewritten in single static assignment (SSA) form to avoid the existentials. Constructing the correct frame rules is a miserable process when the the pre-condition contains a growing number of assertions (Props) at each step.

Even with the bit of pre-existing machinery that using VST separation algebra provides, much more automation (in the form of Ltac) is necessary before using CCHL will be tolerable. That said, seeing as how VST has managed to construct impressive tactics that make proving C programs relatively painless, we believe that this should be possible.

6 Future Work

The final goal is to use our concurrent crash Hoare logic to prove functional correctness and crash safety of a concurrent file system. But first, we still need to fully prove the logic sound with respect to our semantics of our imperative language.

That said, our language and logic must be extended in multiple ways to more fully model a concurrent file system. First, we need a permission model to allow for multiple readers of a heap address. Otherwise key file system objects that are shared among many or all threads (like vnodes) cannot be handled properly.

Additionally, our language does not currently include any dynamic heap allocation, including malloc, free, lock creation, and lock deletion. These are necessary to be able to correctly model operations such as file system mount and unmount. A fundamental issue is that, traditionally, dynamic locks introduce the issue of "predicates on the heap" mentioned earlier. Once locks are dynamic and predicates are placed on the heap, we lose the power to reason about what global invariants a program must abide by, and as such can no longer reason about a consistent state for recovery to act upon.

We have toyed with the idea that a solution to this problem is to give each lock an explicit type, and all locks of a given type must share the same resource invariant, parameterized by the address of the resource that the lock is protecting. In fact, this is how we have currently implemented the map from a lock to its invariant, and it has worked well in our static-lock language. For example, all vnode locks would share the same invariant, as applied to the individual vnode they protect. This seems like a reasonable proposition from a programming standpoint. In this case, we no longer have predicates on the heap, and the (L^c, L^r) mapping to the left of $\vDash$, parameterized by both lock type and address, once again contains all possible strong invariants. Here, the strong invariants associated with each lock type would constitute the global invariants to which the overall program adheres.

As mentioned earlier, there is an alternative frame rule we have not yet implemented that combines the two frame rules we currently have. A related issue is that any resource pulled into the pre-condition must also be pulled into the crash pre-condition, though the actual constraints (weak vs. strong) can differ between the two. Once dynamic memory is implemented, it seems to make even less sense for something like $\exists(x : vnode), 10 \mapsto x$ to appear in a strong invariant, since 10 can be deallocated and reallocated.

Our proposed solution to this problem is to combine the crash conditions into the regular conditions and maintain only a set of addresses, each of which would be tagged directly with both the weak and strong invariants by which it must abide (according to the lock type that protects the address). The strong invariant must be respected whenever we store to that address, and the weak invariant must be respected whenever we attempt to release the lock protecting that address. Now, we have an even simpler frame rule that can frame anything out of the current state, since framing out an address automatically frames out both the strong and weak invariants associated with it.

An additional set of complications, that pertain mostly to the language and not the logic, is that in order to have abstract compound objects like vnodes, we need to extend the type system of our language with at least record types and maybe other things. Going back to a model where we can use arbitrary Coq values as program values would eliminate this requirement (and was one of the primary motivations for wanting to do that), but in the long run we would like to be able to extract the file system written in our language, compile it to C (or something of the sort) and run it outside of Coq; that requires not depending directly on Coq internal phenomena.

7 Conclusions

We have introduced concurrent crash Hoare logic to modularly reason about system correctness in the presence of both concurrency and crashes. We still have a long way to go before we can use it to prove a file system correct, including developing sufficient tactic support and proving the logic's soundness. Nevertheless, our current results are promising. If and when proven sound, CCHL stands as the first verified logic for reasoning about the crash-correctness of concurrent programs.

References

[1] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, et al. Cogent: Verifying high-assurance file system implementations. In *ACM SIGPLAN Notices*, volume 51, pages 175–188. ACM, 2016.

[2] A. W. Appel, R. Dockins, L. Beringer, A. Hobor, J. Dodds, S. Blazy, X. Leroy, and G. Stewart. *Program logics for certified compilers*. Cambridge University Press, 2014.

[3] S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. In *Software Engineering and Formal Methods*, pages 84–98. Springer, 2015.

[4] T. Chajed. *Verifying an I/O-Concurrent File System*. PhD thesis, Massachusetts Institute of Technology, 2017.

[5] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37. ACM, 2015.

[6] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *ACM SIGPLAN Notices*, volume 48, pages 287–300. ACM, 2013.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP '05)*, pages 48–60, 2005.

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[9] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic (extended version). Technical report, Tech. report, Princeton University, 2008.

[10] C. B. Jones. *Development methods for computer programs including a notion of interference*. Oxford University Computing Laboratory, 1981.

[11] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[12] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. In *Asian Symposium on Programming Languages and Systems*, pages 169–188. Springer, 2015.

[13] P. W. Ohearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007.

[14] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[15] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 1, 2016.

[16] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, volume 50, pages 357–368. ACM, 2015.

[17] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 131–146. USENIX Association, 2006.