FSCQ Final Report

Daniel Ziegler

Overview

I've been working hard on FSCQ since December, and I've been able to contribute a lot to the project. I started out by designing a record system able to automatically serialize and deserialize simple data structures into fixed-length words. Using that, I was able to complete the first versions of the inode layer and block allocator that packed multiple pieces of data into a single disk block. To prove them correct, I had to prove many facts about arithmetic with fixed-length words, most of them to show that a particular calculation did not overflow. This required a number of new lemmas and a lot of cumbersome manipulation of expressions. Rather than bashing through each proof manually, I decided to follow a strategy similar to the one used in Bedrock: goals and hypotheses about words are converted into goals and hypotheses about natural numbers, for which Coq has much better standard library support, including tactics like `omega` which are decision procedures for large classes of equations and inequalities. Generally, the conversion requires proving that no overflow will occur for every operation in the expression. However, I realized that in some cases, the context implies that an overflow actually would not matter. Thus, I made my automation more clever than Bedrock's to eliminate these unnecessary proof obligations.

Once we had established that our logic and automation were sufficiently powerful to build up the file system, we decided to move to a more realistic model of the disk supporting asynchronous IO. This means that, after a crash, an arbitrary selection of unsynced writes could take effect on disk. So I started modifying the logging system to support asynchronous IO, as well as building up the transaction in memory before it is committed. This required very careful selection of the representation invariants describing the state of the transaction in various points of its execution. Not only did the pre- and post-conditions have to line up, but the invariants had to be as permissive as possible to allow for all possible states that could arise after a crash. It required quite a few iterations to get this right. Also, as I was writing the log, we added some more requirements that required changes to large parts of the code and proofs, such as running on top of a buffer cache and supporting a configurable disk layout specified by a superblock. I learned a lot in the process, and in the following sections I'll give more technical details about what I did and explain the conclusions I drew.

Asynchronous logging

The core of FSCQ is the write-ahead logging layer, which groups writes into all-or-nothing transactions, exposing a simple interface on top of the real disk, which does not always persist writes before reporting they have completed and may thus lose writes when the entire system suddenly crashes. This section details the failure model and describes the design of the log.

Failure model

The failure model for a program using the disk is a fail-stop one: a program can finish without crashes, or a program can transition into a crashed state at any time between two disk operations. Once the program enters the crash state, it stops executing until a reboot, at which point it executes a specified recovery procedure – in this case, log recovery. We assume that individual disk operations are atomic: they take effect completely or not at all (i.e., no partial block writes). This model is a simplistic one, since some real disks can corrupt the block being written during a power failure; we leave it to future work to incorporate such failures into our model. Because writes are asynchronous, the model has to represent the fact that, after a crash, not every issued write will be reflected in the disk state. To model this, the infrastructure represent each block as a set of possible values, with a special "last write" value representing the last write to this block. Reading from a block returns the "last write" value, since even if there are outstanding writes, a read should return the last write. Writing to a block adds the new value to this set, and marks the new value as the "last write". Finally, the sync operation discards all values in the set except for the "last write". When a system crashes, the new disk state chooses a value for each disk block non-deterministically from the set of all possible values for that block before the crash.

Modular log layers

The first time we built the fully-functional log layer on top of the asynchronous disk model, we did so monolithically, combining the representation invariants and logic into one big mess. In the latest revision, which also aimed to improve the proof automation, I split the log into three components: one to maintain the on-disk log, one to build up the transactions in memory and present an abstract memory interface, and one to update the data portion of the disk before the log is cleared. So far, I have only implemented the first one. Unlike the original log layer, which had eight states (six of which were relevant for crash recovery and not just intermediate states), the new disk log layer only has four crash-relevant states, for two operations: extending the on-disk list by appending to it, or shortening it to a shorter length.

Disk log representation invariant

	Length	Descriptor	Start of data		Rest of data	
Synced	$\langle \texttt{old},\ \varnothing\rangle$	$\langle \texttt{old},\ \varnothing\rangle$	synced `old`		?	
ExtendedDesc	$\langle \texttt{old},\ \varnothing\rangle$	$\langle \texttt{old++new},\ \{\texttt{old}\}\rangle$	synced `old`		?	
Extended	$\langle \texttt{old} + \texttt{new},\ \{\texttt{old}\}\rangle$	$\langle \texttt{old++new},\ \varnothing\rangle$	synced `old`		synced `new`	?
Shortened	$\langle \texttt{cur},\ \{\texttt{cur} + \texttt{cut}\}\rangle$	$\langle \texttt{cur++cut},\ \varnothing\rangle$	synced `cur`	synced `cut`	?	

The table above shows the constraints that different states in the log's representation invariant impose on different parts of disk. Note that they must encompass not just the states relevant for the public interface (i.e. only Synced) but all possible intermediate states: If an extend operation crashes after it has written to the descriptor but before it has touched anything else, the disk no longer satisfies the Synced state, since the descriptor is unsynced. Likewise, right after the length block is written during either an extend or a shorten operation, it is unsynced. Interestingly, there are many intermediate states in the code that the representation invariant does not specify precisely. After the descriptor is written, for instance, the extend code writes the new data blocks one by one into the log, and then syncs them all one by one. Since none of the states specify the contents of the rest of the data portion, it doesn't matter (either for the public interface or for crash recovery) that this happens.

This kind of reasoning, for figuring out how the representation invariant should be specified, is not very straightforward: it requires thinking about all possible states the log could be in; the predicates need to be strong enough to satisfy the preconditions of the next function *and* the recovery procedure, yet those used in the recovery procedure need to be simultaneously weak enough to be satisfied by all crash states; and the states depend on each other in a circular manner (both across cycles of transaction execution and cycles of repeated crashes and recovery). As a result, I went through a number of iterations of trying to prove part of FscqLog correct, realizing I needed to change the representation invariant, and then having to adjust other proofs as a result. It took more effort to converge than I expected. If I had taken a step back and reasoned through the diagram above before actually trying to prove anything, I would have saved myself a lot of work.

Proof automation

The vast majority of the reasoning in proving FSCQ correct is quite uninteresting and very amenable to automation. I attempted to make as much use of Ltac as possible to simplify proofs, but Ltac's warts and Coq's bugs made automation rather painful to use and create. This section describes some of the different kinds of automation I created, and the lessons I learned as a result. In general, the tactics tend to perform as many rewrites as possible to simplify expressions and bring them into a normal form, and then apply some theorems and sub-tactics to take care of the resulting goals.

Word automation

Consider a hypothesis of the form $a < b * c$, where a, b, and c are fixed-length words. A simple transformation to a hypothesis about nats would apply a comparison conversion theorem to obtain $\#a < \#(b * c)$, where # denotes conversion from word to nat, and then rewrite the multiplication to obtain $\#a < \#\$(\#b * \#c)$, where \$ denotes conversion from nat to word. Then, it would convert this into $\#a < \#b * \#c$, generating an additional goal that $\#a * \#b$ is small enough not to overflow. However, this side condition should not be necessary. If the word multiplication overflows, it only strengthens the original condition: in that case, #a is of a valid size but #b * #c is not, so $\#a < \#b * \#c$ is trivially true. Also, in more complicated expressions such as $a + b * c$, it would be silly to prove that first $\#b * \#c$ is well-sized and then $\#a + \#b * \#c$ is well-sized, since the latter implies the former. To take advantage of these facts, the automation works according to the following procedure:

1. Naively convert all arithmetic operations on words into ones working on nats
2. Eliminate unnecessary round trips from nat to word and back using lemmas such as $\$(\#\$x+y) = \$(x + y)$ and $x < \#\$y \implies x < y$.
3. Only then, perform rewrites that generate side conditions, rewriting $\#\$a$ to a.
4. Try to solve as many goals as possible using omega, congruence, and theorems like $x/y \leq x$.

There are some things the automation currently does not handle completely, such as subtraction. Since nat subtraction saturates on underflow but word subtraction wraps around, converting a subtraction requires showing that the first argument is greater than the second. However, this isn't necessary in all cases: Consider expressions of the form $(a - b) + c$. This will not underflow as long as $\#b \leq \#a + \#c$, but the current automation strategy would generate the unnecessarily strong obligation $\#b \leq \#a$. This didn't seem to be too much of an issue in practice, however. In fact, I'm not sure how much the existing attempts to reduce side goals helped either. I imagine it reduced the manual proof effort slightly in a few places, but I haven't actually made the comparison. Moreover, it is possible that the benefit is outweighed by the increased computation time in the tactic: Coq rewriting is slow, and the tactic often takes a surprisingly long time, slowing down exploration of possible proofs.

Array matching automation

As I gradually proved FscqLog correct, I attempted to use proof automation as much as possible. The ideal would have been for a single tactic invocation to take care of everything – once the tricky task of specifying the representation invariants is done, this should be theoretically possible. In pursuit of this goal, I created a number of Ltac tactics to take care of the kinds of goals that the Hoare logic automation creates for FscqLog. In a number of places, I needed to show that the separation conjunction of three or four array predicates (which specify the contents of a contiguous section of memory) implied another array predicate. The former were just a partition of the latter, so I wanted to convert the goal into one saying that the concatenation of the former's list equaled the latter's list. This would have been trivial to automate, except that the smaller predicates could be out of order, and I needed to make sure the automation only fired when it was actually correct rather than trying to concatenate non-contiguous pieces of memory. Thus, I implemented an $O(n^3)$ `array_sort` in Ltac to put the them in order. Of course, the tactic has to sort not integer addresses but expressions computing those addresses, such as LogBegin, LogBegin + (length m), and so forth. It repeatedly tries to prove that an array's start address is always less than or equal to the array ahead of it, and if so, swaps them. However, if both addresses are equal, it has to show that one array is empty and eliminate it, or else give up. To do this, it uses another tactic, `solve_lengths`, which performs some rewrites to simplify expressions involving the lengths of various lists and then applies my word automation. If, in the end, `array_sort` can't prove that all the arrays are in the right order, it fails.

After I applied `array_sort` to a number of cases, appending together all of the arrays that it sorted, I realized that the resulting list equalities were not really possible to prove automatically, because just appending the lists threw away too much information about their lengths. Sometimes, a goal would be generated that would be of the form `l = ?l0 ++ ?l1`, yet that goal did not have any clue how long `?l0` and `?l1` should actually be. Thus, rather than appending all the array contents after `array_sort`, I made a new tactic `array_match` that matches up corresponding parts of arrays and generates separate list equality goals for each part. Given a predicate implication with array predicates on both sides, it repeatedy chops off a piece off the end of the list: it compares the start addresses of the last array predicates on either side, splits the longer one, and matches the shorter one with the suffix of the longer one. The resulting goals are still not all easy to automate, but there is more information available.

The idea behind `array_sort` and `array_match` was quite simple, but developing them turned out to be rather tedious. I passed it into the Hoare Logic automation to have it be tried on the generated goals. The advantage of passing it in rather than just running it on all the resulting goals was that it enabled the Hoare Logic automation to try all cases of disjunctions on the left side of an implication, and if all the goals can be solved for one of the cases, solve the entire implication automatically. This is nice when it works, but it's one of the reasons that the Hoare Logic automation can be rather slow (especially since `solve_lengths` and certainly `array_sort` are quite slow themselves). As a result, each iteration could require up to twenty minutes of proof search, after which I would often discover that some unexpected corner case had been hit and my tactic generated an unprovable goal. Other times, the Hoare logic automation itself would generate an unprovable goal.

One interesting example was when existential variables started getting mysteriously filled in with empty lists, even though this was incorrect. I had to manually copy in the definitions of the automation and execute pieces of them to find where this happened. It turned out that in one place, `intuition` was applying `auto` to a goal of the form `Forall ... ?l`, which solved it by applying `Forall_nil`, a hint introduced by the standard library. Before this, I had thought that `auto` would not instantiate evars, but it turns out this is not precisely right – the difference to `eauto` is actually that it uses pattern matching rather than unification when finding lemmas to apply, but if a lemma is found using pattern matching, it can still instantiate evars. At other times, the automation had somewhat unpredictable behavior. When it cancels an implication, it tries to apply `finish_frame`. The intent was to solve implications where one side is just a single term with an evar. However, it also ended up canceling implications with multiple terms sometimes. This was not wrong – but it only happened if the terms randomly happen to be in the same order of both sides. Since inconsequential changes often cause terms to be reordered, this caused the behavior to vary unpredictably.

Coq's instability also made the automation development far more painful. Using Coq 8.5 beta, I triggered four separate crashing bugs (assertion failures) over the course of two days developing `array_match`. As a result, I had to write much longer proofs that tried to sidestep the minefield of random crashes.

These experiences have convinced me that it's extremely valuable to write conservative tactics that behave predictably and never turn true goals into false goals. However, Coq currently makes this fairly difficult to accomplish, especially when there are evars involved. Also, the benefit of tactics that can completely solve goals goes beyond the manual effort they might save versus leaving a small amount left to prove: they can be much more effectively composed with automation that tries a number of possibilities.

Conclusion / Future Work

Working on FSCQ has taught me a lot, and there are many interesting directions the project could be taken in. I thought of a potential alternative strategy for proving parts of FscqLog correct: Rather than making custom predicates `equal_unless_in` and `nil_unless_in` to separate out the parts of disk that are affected by a transaction from those that are not, it might be worth splitting the memory into two parts using separation logic, and then writing normal predicates on those parts. I'm not sure whether this would be easier or harder to prove (with automation or without), but I'd like to try it.

Taking a broader perspective, I'm interested in taking a more theoretical perspective of the Crash Hoare Logic, with namespaces, that we've introduced, and figure out how to present that for a future paper. Along with that, I'd like to think about creating general-purpose automation that solves a large, predictable class of Crash Hoare Logic theorems. It should be possible to use the experience I've gained to think through the automation more systematically, and create something that's more powerful and nicer to use.