Argosy: Verifying Layered Storage Systems with Recovery Refinement

Tej Chajed
MIT CSAIL, USA
tchajed@mit.edu

Joseph Tassarotti
MIT CSAIL, USA
jtassaro@andrew.cmu.edu

M. Frans Kaashoek
MIT CSAIL, USA
kaashoek@mit.edu

Nickolai Zeldovich
MIT CSAIL, USA
nickolai@csail.mit.edu

Abstract

Storage systems make persistence guarantees even if the system crashes at any time, which they achieve using recovery procedures that run after a crash. We present Argosy, a framework for machine-checked proofs of storage systems that supports layered recovery implementations with modular proofs. Reasoning about layered recovery procedures is especially challenging because the system can crash in the middle of a more abstract layer's recovery procedure and must start over with the lowest-level recovery procedure in the stack.

Argosy introduces *recovery refinement*, a set of conditions that ensure proper implementation of an interface with a recovery procedure. The metatheory of the framework proves that recovery refinements are composable and entail an end-to-end correctness theorem. The framework appeals to Kleene algebra to specify and reason about recovery execution, leading to concise definitions and metatheory. To prove recovery refinement, we implemented Crash Hoare Logic (CHL), the program logic used by FSCQ [8], and applied it to verify an example of layered recovery featuring a write-ahead log running on top of a disk replication system. The metatheory of the framework, the soundness of the program logic, and these examples are all verified in the Coq proof assistant.

CCS Concepts • **Theory of computation → Program verification**; • **Hardware →** *System-level fault tolerance*;

Keywords Kleene Algebra, Refinement

ACM Reference Format:
Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Argosy: Verifying Layered Storage Systems with Recovery Refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314585

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314585

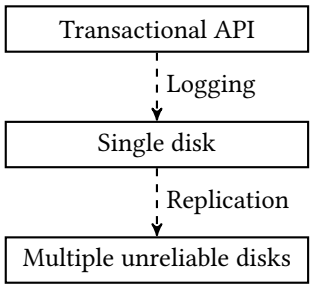


Figure 1. A simple storage system that uses recovery at multiple layers of abstraction.

1 Introduction

Storage systems, including file systems, databases, and persistent key-value stores, must protect data from loss even when the computer crashes (e.g., due to a power failure). These systems provide crash-safety guarantees about what data persists if such crashes occur. To achieve these guarantees, many systems perform some form of repair in a recovery procedure that runs after a reboot.

Storage systems are typically structured into several layered abstractions. For example, a storage system might use several physical disks for redundancy (see Figure 1). By replicating writes across these disks, the storage system can implement an interface presenting a single synchronous disk, and then use write-ahead logging to implement a transactional API for atomically writing multiple disk blocks.

If the computer crashes, write operations may have occurred on only some of the physical disks. To repair its state, the storage system runs a recovery procedure after reboot. First, it propagates missing writes to the remaining disks to restore replication. Then, it reads the transaction log to determine if transactions need to be aborted or applied, based on whether the system crashed before or after they were committed. The storage system may have to run the recovery

procedure several times, because the system can crash again during recovery.

Because a storage system needs to handle crashes at any time,[1] implementing and testing them is difficult. Storage systems in practice have had bugs that resulted in data loss and data leaks [16, 17, 23]. Since these bugs are costly, formal verification is attractive because it can rule out large classes of bugs. For verification to scale to modern, complex storage systems, the proofs for the implementations in each layer should be independent. This independence is hard to achieve because crashes during one layer of abstraction's recovery procedure requires re-running all of the recovery procedures of lower levels. For example, with the system in Figure 1, a crash in the middle of the write-ahead log's recovery procedure may leave disks out-of-sync, which requires starting over with the replicated disk's recovery.

This paper presents Argosy, a framework for verifying storage systems that supports layered recovery procedures with modular proofs. Argosy introduces the notion of *recovery refinement*, a set of proof obligations for an implementation and recovery procedure. These obligations are sufficient to guarantee clients observe the specification behavior, including with multiple crashes followed by recovery. Furthermore, recovery refinement *composes* between two implementations: this allows the developer to prove each implementation separately and then obtain a proof about the whole system with a general composition theorem. We describe the metatheory behind recovery refinement in section 4. The framework is encoded in the Coq proof assistant, with machine-checked proofs of soundness.

There are several existing systems that support reasoning about crashes and recovery, particularly in the context of file-system verification [7, 8, 11, 26, 28]. Most have no support for layered recovery, since they consider only a single recovery procedure at a time. The Flashix modular crash refinement work [11] does consider layered recovery, but to simplify proofs recovery procedures cannot rely on being able to write to disk. Argosy supports *active recovery* procedures which write to persistent storage; both the replicated disk and write-ahead log implementations rely on active recovery. Furthermore, the metatheory for a number of existing systems is based on pen & paper proofs, whereas Argosy has machine-checked proofs for both the metatheory and example programs.

To prove recovery refinement within a single layer, Argosy supports a variant of Crash Hoare Logic (CHL), the logic used in the FSCQ verified file system [7, 8]. Argosy generalizes FSCQ's CHL by supporting non-deterministic crash behavior, whereas FSCQ modeled only persistent state and assumed it was unaffected by a crash. The main benefit of using CHL is that as long as recovery's specification satisfies an *idempotence* condition, the developer can reason about recovery using only its specification and ignore crashes during recovery.

To simplify the definition of recovery execution as well as facilitate proofs of Argosy's metatheory for recovery refinement, we formulated the execution semantics and recovery refinement using the combinators of Kleene algebra [18]. Kleene algebra is well-suited for this purpose because it models sequencing, non-determinism, and unbounded iteration, which arise naturally when reasoning about crashes and recovery.

As a demonstration of Argosy, we implemented and verified the storage system of Figure 1. The disk replication and write-ahead log are separately verified using CHL, each with its own recovery procedure; section 6 details how this proof works in CHL within Argosy. We then compose them together to obtain a verified transactional disk API implemented on top of two unreliable disks. The composed implementation extracts and runs, using an interpreter in Haskell to implement the physical disk operations at the lowest level.

The paper's contributions are as follows:

1. Argosy, a framework for proving crash-safety properties of storage systems that introduces *recovery refinement* to support modular proofs with layered recovery procedures.
2. Machine-checked proofs in Coq of the metatheory behind recovery refinement that are simplified by appealing to properties of Kleene algebra.
3. An implementation of Crash Hoare Logic (CHL) for proving a single layer of recovery refinement, which we use to verify an example of a storage system with layered recovery.

2 Related Work

Verified storage systems Crash Hoare Logic [8] and Yggdrasil [28] are two frameworks for reasoning about storage systems, in which the authors built verified file systems (FSCQ and Yxv6, respectively). These frameworks address a number of complications raised by storage systems, especially reasoning about crashes at any time, recovery following a crash, and crashes during recovery. CHL introduces the notion of a *crash invariant*, an execution invariant that recovery relies on, as well as *idempotence*, a property where recovery's precondition is invariant under crashes during recovery. Yggdrasil takes a different perspective on specifications and uses refinement from the abstract specification to the code, proving the code's crash behaviors are a subset of the abstract crash behaviors. Both frameworks are also careful to specify crashing and non-crashing behavior separately; this is important since storage systems ought to provide stronger guarantees for non-crashing execution (for example, data buffered in memory might be lost on crash, but

[1] In this work we use "crash" to refer to the entire storage system halting and requiring restart, such as due to a power failure or kernel panic.

if the system does not crash reads should reflect all previous writes).

Argosy incorporates ideas from both of these previous lines of work, extending them to multiple layers with recovery, and with a fully machine-checked metatheory. In contrast, both CHL and Yggdrasil assume a single, global recovery procedure. Yxv6 consists of a number of verified layers, but the end-to-end refinement is an informal theorem, not a part of a proven metatheory. CHL does not use refinement, so for modularity FSCQ specifications are by convention structured into multiple abstractions. Refinement makes this modularity explicit. In Argosy, recovery refinement has the benefit that clients of an interface are also able to use recovery procedures, even if that interface is itself implemented using recovery; in FSCQ and Yxv6, clients of the file system have no mechanism to prove crash safety of their own recovery procedures.

Ernst et al. [11] also develop a theory of submachine refinement for crashing and recovering systems, which they used to verify their Flashix file system [10]. Their work has a similar notion of a refinement between an abstract specification and its implementation; their reduction from a white-box semantics to a black-box semantics for submachines is analogous to our notion of recovery refinement between interface L_A and L_C, as described in section 4. However, the metatheory of submachine refinement only holds when a strong property holds of the entire submachine: all operations must be "crash neutral", meaning there must be a way for every operation to run such that its effects are obliterated by a crash. This holds in the setting considered by the authors because they permit all writes to storage to fail, but does not support active recovery procedures that rely on writes, as well as complicating higher-level APIs by always including the possibility of failure. In contrast Argosy has a simpler and more general presentation of refinement that handles active recovery procedures.

Ntzik et al. [26] developed an extension to concurrent separation logic to support reasoning about crashes in concurrent systems. Similar to CHL's idempotence principle, this logic has a rule for verifying a recovery procedure which involves showing that the precondition for recovery is an invariant during recovery's execution. However, as with CHL, this rule applies to verifying a single recovery procedure, as opposed to the multiple layers in Argosy.

The metatheory of Argosy is all accompanied by machine-checked proofs, unlike Yggdrasil, submachine refinement, and the logic of Ntzik et al. [26].

Concurrent notions of refinement There are systems like RGSim [22] and CCAL [14] which support verification of concurrent software using refinement between multiple implementation layers. It might seem that crashes are simply a special case of concurrency, since crashes interrupt threads in a similar way to interleaving threads. However, recovery requires new reasoning principles beyond what concurrency frameworks provide. Crashes interrupt a thread with no possibility of resuming it, an unbounded number of crashes can interrupt recovery itself before it completes, the definition of refinement should abstract away the behavior of recovery, and crash-free executions should have a stronger specification than post-crash behavior. No concurrent refinement framework has direct support for these special aspects of crashes and recovery. Indeed, these differences between crashes and standard concurrency are what required Ntzik et al. [26] to develop the extension to concurrent separation logic described above. However, as mentioned, that logic is for reasoning about a single layer of recovery, rather than multiple layers. Adding crash safety support to a layered concurrent refinement system like CCAL is an interesting direction for future work, where Argosy's ideas would be informative.

Distributed Systems Refinement reasoning is widely used for proving properties of distributed systems [15, 21, 25, 31]. Crashes occur in distributed systems, where nodes may suddenly fail. However, the existing work in this area does not involve reasoning about storage systems running on individual nodes. Instead, these systems' proofs assume correctness of storage at each node and show consistency properties of the aggregate system.

Kleene Algebra for Verification Kleene algebra, especially an extension called Kleene algebra with tests (KAT), has been used for a variety of program verification tasks. Applications range from proving the correctness of compiler optimizations [20], total correctness using refinement [30], and cache coherence protocols [9], to more recently specifying and analyzing software-defined networking systems [1].

Many extensions of Kleene algebra have been proposed, including variants with types [19], temporal modalities [4], probabilistic operators [12], monadic operators [13], and with equational axioms [3]. These extensions often enjoy *completeness* — any inequality that holds in the application domain can be proven using the axioms of the algebraic structure — and *decidability* — there is an algorithm to decide if an inequality is true or false. These two properties together make Kleene algebra attractive as a basis for automation, and indeed this automation has even been verified in Coq [5, 27]. We do not develop an axiomatic variant of Kleene algebra, and our variant embeds arbitrary Coq terms that certainly make it undecidable; instead, we focus on only one model and use Kleene algebra as a reference, manually proving any desired property within the model. It would be interesting to specify our variant axiomatically and try to identify a decidable fragment that is sufficient for our needs; this could automate our proofs considerably.

3 Combinators for Crash Semantics

This section describes how we specify systems and program in Argosy using combinators from Kleene algebra.

3.1 Overview

Argosy is centered around interfaces, which have a *signature* specifying what operations programs can use and an *operational semantics* that specifies how the operations behave. We'll use L to denote interfaces and Proc_L for the type of programs using operations from the interface L. An interface's signature determines what operations are valid, so the L stands for **l**anguage, though programs in any Proc_L share a common syntactic structure that gives control flow and additional operations to compose operations as we describe in more detail in subsection 3.2.

Interfaces include a semantics for each operation. Argosy defines the semantics of whole programs, composing together each operation's behavior appropriately for normal, crash, and recovery execution. On top of this common structure, Argosy includes a metatheory for reasoning about recovery execution. The semantics and metatheory leverage Kleene algebra to simplify both definitions and proofs; we describe our specific use of Kleene algebra in subsection 3.3.

3.2 Interfaces

Formally, an interface L in Argosy is a tuple $(O, S, \text{step}, \lightning, P)$, where

- O is the type of atomic operations the interface exposes. Each operation is indexed by the type of values it returns upon executing. O serves as the signature of the interface.
- S is the type of abstract state used to describe the interface's semantics.
- step is a transition relation specifying the semantics of each operation. If o is an operation of type $O(T)$, the relation $\text{step}(o)(s, s', v)$ holds if the operation o can transition from state s to state s' and return the value v of type T.
- $\lightning$ (pronounced "crash") is a transition relation on program states specifying the possible effects of a system crash. For example, if the system state contained volatile memory, then a $\lightning$ transition would erase this memory.
- P is a predicate on S which holds for any valid initial configuration of the system.

Example 1 (Transactional Disk API)**.** Figure 2 gives the definition of the transactional disk, the interface implemented by the write-ahead logging scheme mentioned in the introduction. The interface's abstract state consists of a tuple of disks of the form (d_{old}, d_{new}), where d_{old} represents the state of the disk before the current transaction began, and d_{new} represents what the disk will become after the transaction is committed. We use s.old for the first element of the state s and s.new for the second element. Each disk is modeled as a list of blocks. The initialization condition P guarantees both disks are the same size, and all operations preserve this invariant; the size operation returns this common length. The operation write($addr$, blk) sets the value of $addr$ to blk in d_{new} if it succeeds. (Out-of-bounds writes have no effect). However, it may also fail, in which case the disks are unchanged — this corresponds to the situation where the transaction log is full. On the other hand, the operation read($addr$) gets the value of $addr$ from the old disk, d_{old}, so that it does not see the effects of uncommitted writes. (Out-of-bounds reads return an arbitrary block). The commit operation atomically sets the old disk to be equal to the new disk. A system crash aborts the current transaction, reverting the new disk to the old disk.

For each interface L, Argosy defines a type of programs Proc_L, which have a common syntactic structure that turn the layer operations o into a monad. We further index the type Proc_L by the type of values a program can return. Programs are generated by the following grammar:

$$e ::= \text{call}(o) \mid \text{ret } v \mid \text{bind } e\ (\lambda x.\, e')$$

A call to an operation o from O returning values of type T is written call(o), which is a program of type $\text{Proc}_L(T)$. Programs also include the monad operations ret and bind. In our Coq implementation, bind e f *shallowly embeds* a Coq function f of type $B \to \text{Proc}_L(A)$; that is, programs can include arbitrary Coq functions to sequence operations together. Similarly, we can write any Coq expression v in ret v. Note that this includes Coq if expressions and recursive functions, which can be used to write loops. The only caveat is that Coq recursion is always terminating, so unbounded loops are not supported.

It is straightforward to lift the operation transition relation step(o) to give a semantics for *non-crashing* executions of programs expressed in the monad Proc_L. However, doing so for crashing executions, which run an associated recovery procedure (which may itself crash), is more involved. Although it is possible to do so directly by specifying an inductively defined big-step relation, as in FSCQ [8], the resulting definition can be difficult to understand and reason about. Instead, Argosy expresses this relation using a variety of relational combinators, which we explain next.

3.3 Kleene Algebra Combinators

We define a type for transition relations $\text{Rel}(A, B, T) \triangleq A \to B \to T \to \mathit{Prop}$. For some relation $r : \text{Rel}(A, B, T)$, the proposition $r(a, b, v)$ holds when r allows a transition from state $a : A$ to state $b : B$, returning value $v : T$. We allow transitions to change the type of state to support transitions across layers, but often both state types are the same, such as for step.

$$S \triangleq \text{List Blocks} \times \text{List Blocks} \qquad \lightning \triangleq \lambda s_1, s_2.\ s_2 = (s_1.\text{old}, s_1.\text{old})$$

$$O \triangleq \text{write}(addr, blk) \mid \text{read}(addr) \mid \text{size} \mid \text{commit} \qquad P \triangleq \lambda s.\ s.\text{old} = s.\text{new}$$

$$\text{step}(o) \triangleq \lambda s_1, s_2, r.\ \begin{cases} (s_2.\text{old} = s_1.\text{old} \wedge s_2.\text{new} = s_1.\text{new}[a := blk] \wedge r = \langle\rangle) \vee (s_2 = s_1 \wedge r = \text{failed}) & \text{if } o = \text{write}(a, blk) \\ s_2 = s_1 \wedge r = s_1.\text{old}[a] & \text{if } o = \text{read}(a) \\ s_2 = (s_1.\text{new}, s_1.\text{new}) \wedge r = \langle\rangle & \text{if } o = \text{commit} \\ s_2 = s_1 \wedge r = \text{length}(s_1.\text{old}) & \text{if } o = \text{size} \end{cases}$$

Figure 2. Semantics of Transactional Disk Layer

We define equality, ordering ($p \subseteq q$), and operations for non-deterministic choice ($p+q$), sequential composition ($p \cdot q$), and zero-or-more iterations ($p^\star$):

$$p = q \triangleq \forall a_1, a_2, t.\ p(a_1, a_2, t) \leftrightarrow q(a_1, a_2, t)$$
$$p \subseteq q \triangleq \forall a_1, a_2, t.\ p(a_1, a_2, t) \rightarrow q(a_1, a_2, t)$$
$$p + q \triangleq \lambda x, y, t.\ p(x, y, t) \vee q(x, y, t)$$
$$p \cdot q \triangleq \lambda x, y, t.\ \exists z, t'.\ p(x, z, t') \wedge q(z, y, t)$$
$$p^\star \triangleq \lambda x, y, t.\ \exists n : \mathbb{N}.\ p^n(x, y, t)$$
$$\text{where } p^0 \triangleq (\lambda x, y, t.\ x = y) \text{ and } p^{n+1} \triangleq p \cdot p^n$$

These three operations appear in algebraic structures known as Kleene algebras [18], which axiomatize the familiar properties of regular expressions. Our operations differ from Kleene algebra, however, because they are *typed*, so that certain operations are only defined when the types of the operands match appropriately, which is not the case in a Kleene algebra.[2] For example, our definition of $p \cdot q$ is well-typed only when the type of p is of the form $\text{Rel}(A, B, T_1)$ and the type of q is of the form $\text{Rel}(B, C, T_2)$, i.e. the types of states that p transitions *to* must match the type of states that q transitions *from*. The type of the composition $p \cdot q$ is then $\text{Rel}(A, C, T_2)$.

Despite this difference from Kleene algebra, in our Coq formalization we have proven that most of the axioms of Kleene algebra hold for these combinators, as well as many other derived rules, a selection of which we list in Figure 3. This means that by defining our semantics using these combinators, we are able to take advantage of these equational laws to simplify statements that we must prove.

Any relation of the form $A \rightarrow B \rightarrow \mathit{Prop}$ can be lifted to an output-producing relation of type $\text{Rel}(A, B, \text{Unit})$ which always returns the unit value $\langle\rangle$. We will use this implicit coercion throughout.

The sequential composition $p \cdot q$ above always runs the transition q regardless of what output value is returned by p. We therefore define an additional combinator that allows sequencing transitions based on intermediate output:

$$\text{andThen } p\ f \triangleq \lambda x, y, t.\ \exists z, t'.\ p(x, z, t') \wedge f(t')(z, y, t)$$

We also define an operator that returns a particular value, leaving state unchanged:

$$\text{ret } v \triangleq \lambda x, y, t.\ x = y \wedge t = v$$

For a fixed type A of state, $\text{Rel}(A, A, -)$ forms a monad with andThen and ret as the bind and unit. Note that we use ret to mean both a pure program and a pure relation, which are distinguishable from context (and closely related). Therefore, we will use the traditional bind notation for andThen, writing $x \leftarrow p;\ f(x)$ for $\text{andThen } p\ (\lambda x.\ f(x))$. Similarly, to improve clarity we will use $p;\ q$ instead of $p \cdot q$ when mixing bind notation and sequential composition. We use this notation even for sequencing relations with different input and output state types, since in this case relations form an instance of a more general structure called *parameterised monads* [2].

SEQ-MONOTONIC
$$\frac{p \subseteq p' \qquad q \subseteq q'}{p \cdot q \subseteq p' \cdot q'}$$

SLIDING
$$p \cdot (q \cdot p)^\star = (p \cdot q)^\star \cdot p$$

DENESTING
$$(p + q)^\star = p^\star \cdot (q \cdot p^\star)^\star$$

SIMULATION
$$r \cdot p \subseteq q \cdot r \implies r \cdot p^\star \subseteq q^\star \cdot r$$

Figure 3. Selected theorems from Kleene algebras that hold in our model, when the statement type checks.

3.4 Execution semantics

These relation operations provide a convenient way to specify the crash and recovery behavior of programs. We start by defining the crash-free execution of a program e, written $[\![e]\!]$. For a program e of type $\text{Proc}_L(A)$, its crash-free semantics is a relation $[\![e]\!]$ of type $\text{Rel}(S, S, A)$ (recall S is the state type

[2] Typed variants of Kleene algebras have been studied by Kozen [19].

for the interface L), defined inductively as:

$$[\![\mathsf{call}(o)]\!] \triangleq \mathsf{step}(o)$$
$$[\![\mathsf{ret}\ v]\!] \triangleq \mathsf{ret}\ v$$
$$[\![\mathsf{bind}\ e\ f]\!] \triangleq x \leftarrow [\![e]\!]\,;\ [\![f(x)]\!]$$

This definition simply maps the free monad generated by the operations of L into the relation monad: binds and returns of programs become the binds and returns of relations, while operations are interpreted according to $\mathsf{step}(o)$.

Next, we define a relation $[\![e]\!]_{\lightning}$ which describes a partial execution of e, interrupted by a crash that can occur non-deterministically at any step. The definition of $[\![e]\!]_{\lightning}$ captures the behavior of crashing programs after a crash but before recovery take place. We ignore any partial return value of such an interrupted program and just return $\langle\rangle$, but any state modifications are preserved:

$$[\![\mathsf{call}(o)]\!]_{\lightning} \triangleq \lightning + (\mathsf{step}(o) \cdot \lightning)$$
$$[\![\mathsf{ret}\ v]\!]_{\lightning} \triangleq \lightning$$
$$[\![\mathsf{bind}\ e\ f]\!]_{\lightning} \triangleq [\![e]\!]_{\lightning} + (x \leftarrow [\![e]\!]\,;\ [\![f(x)]\!]_{\lightning})$$

When executing an operation o, the system may crash before or after executing o; this means operations are atomic with respect to crashes (modulo the effects of the subsequent crash itself). When executing a pure program $\mathsf{ret}\ v$, a crash will erase the return value of v. For $\mathsf{bind}\ e\ f$, the system may either, (1) crash while executing e, or (2) finish executing e but then crash while executing f.

We also define recovery execution of a program e with a recovery procedure $r : \mathsf{Proc}_L(\mathsf{Unit})$, written as $[\![e \circlearrowleft r]\!]$. Note that $\circlearrowleft$ is just part of the notation, not an operation of the $\mathsf{Proc}_L(A)$ monad. Recovery execution involves a crashing execution of e, followed by potentially multiple crashing executions of r, and then a final complete execution of r:

$$[\![e \circlearrowleft r]\!] \triangleq [\![e]\!]_{\lightning} \cdot [\![r]\!]^{\star}_{\lightning} \cdot [\![r]\!]$$

Giving concise definitions of crash and recovery semantics like the above is important because these definitions are trusted: if we omit some possible behavior of crashing from our definitions, then the results we prove about crash safety may not apply to real executing programs.

4 Recovery refinement

This section discusses implementing interfaces, specifying correctness of implementations using refinement, and the metatheory Argosy provides for composing layers.

4.1 Implementations

An implementation of an interface is a program written against the target, lower-level, concrete interface for each primitive operation in the higher-level, abstract interface. A software stack is then a sequence of implementations composed together. In this section we'll discuss implementing one interface in terms of another, returning to the issue of composition in subsection 4.3. To improve readability, we will color-code the abstract interface in blue and concrete interface in red. More formally, an *implementation* of abstract interface L_A using concrete interface L_C is a tuple $M = (c, r, i)$, where

- c is a function mapping each operation o from L_A with return type T to a procedure $\mathsf{Proc}_{L_C}(T)$ implementing the operation.
- r is a designated recovery procedure of type $\mathsf{Proc}_{L_C}(\mathsf{Unit})$.
- i is an initialization procedure of type $\mathsf{Proc}_{L_C}(\mathsf{Bool})$, which returns a boolean indicating whether initialization succeeded.

We use the meta-variable M for implementations because we think of them as **m**odules implementing all of the operations of the signature L_A; see Figure 4 for an illustration of how an implementation relates two layers. Given such an implementation, we can "compile" programs written in L_A into programs in L_C:

$$C_M(\mathsf{call}(o)) \triangleq c(o)$$
$$C_M(\mathsf{ret}\ v) \triangleq \mathsf{ret}\ v$$
$$C_M(\mathsf{bind}\ e'\ f) \triangleq \mathsf{bind}\ C_M(e')\ (\lambda x.\, C_M(f(x)))$$

When compiling a recovery procedure r_A, we need to first ensure that the implementation recovery procedure r is run, and then translate the operations in r_A:

$$C^{\circlearrowleft}_M(r_A) \triangleq \mathsf{bind}\ r\ (\lambda\_.\, C_M(r_A))$$

Next, we want to show that these implementations are correct. Before giving our specific conditions for an implementation, we give a general definition for refinement between relations. A refinement involves first picking an abstraction relation $R : S_A \to S_C \to \mathsf{Unit} \to \mathit{Prop}$ that relates abstract and concrete states (the unit value makes the abstraction relation an output-producing relation of type $\mathsf{Rel}(S_A, S_C, \mathsf{Unit})$).

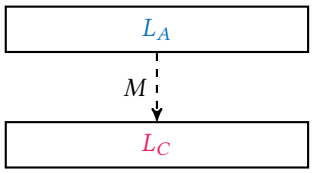


Figure 4. An implementation $M = (c, r, i)$ of L_A using L_C provides a way to run abstract programs Proc_{L_A} using a concrete interface by compiling them with C_M, running the recovery procedure r after each crash, and initializing the interface with the procedure i.

Definition 2. For an implementation transition relation *impl* and a specification transition *spec*, we say the implementation *refines* the specification under the abstraction relation R, written $\mathit{impl} \sqsubseteq_R \mathit{spec}$, if whenever $R(s_A, s)$ and

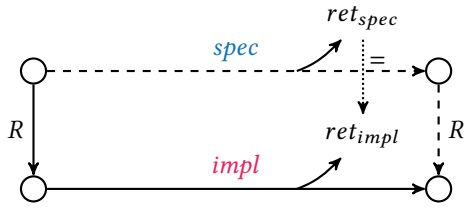


Figure 5. Definition of the refinement $impl \sqsubseteq_R spec$.

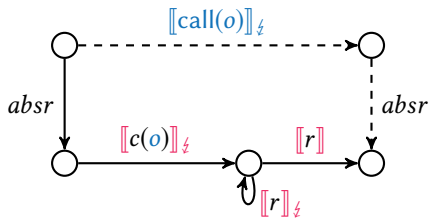


Figure 6. Abstraction for crash and recovery.

$impl(s, s', v)$ hold, there exists an s'_A such that $spec(s_A, s'_A, v)$ and $R(s'_A, s')$.

Figure 5 depicts this definition graphically. We can give an equivalent, compact definition of refinement by using relation composition with a varying state type, as follows:

$$impl \sqsubseteq_R spec \triangleq (R\,;\, impl) \subseteq (v \leftarrow spec\,;\, R\,;\, \text{ret } v)$$

Building on this abstract notion of refinement, we propose *recovery refinement* as a simple set of conditions developers can prove about an implementation to establish correctness.

Definition 3. We say that an implementation $M = (c, r, i)$ is a *recovery refinement* of abstract interface $L_A = (O_A, S_A, \text{step}_A, \lightning_A, P_A)$ into $L_C = (O_C, S_C, \text{step}_C, \lightning_C, P_C)$ if there exists an abstraction relation R with the following four properties:

COMPILE-EXEC: For all o, $[\![c(o)]\!] \sqsubseteq_R [\![call(o)]\!]$
RECOVER-OP: For all o, $[\![c(o) \circlearrowleft r]\!] \sqsubseteq_R [\![call(o)]\!]_{\lightning}$
RECOVER-RET: $[\![\text{ret } \langle\rangle \circlearrowleft r]\!] \sqsubseteq_R [\![\text{ret } \langle\rangle]\!]_{\lightning}$
INIT-ABS: For all s and s',
$P_C(s) \wedge [\![i]\!](s, s', \text{True}) \longrightarrow \exists s_A.\, P_A(s_A) \wedge R(s_A, s')$

Note that in this definition we folded some definitions to give some intuition as to why these conditions make sense; the conditions require that each operation is implemented correctly, using recovery on the implementation side to transparently achieve the abstract interface's crash behavior $\lightning_A$. We can unfold some definitions to re-write the first three conditions as follows:

COMPILE-EXEC: For all o, $[\![c(o)]\!] \sqsubseteq_R \text{step}_A(o)$
RECOVER-OP: For all o, $[\![c(o) \circlearrowleft r]\!] \sqsubseteq_R \lightning_A + (\text{step}_A(o) \cdot \lightning_A)$
RECOVER-RET: $[\![\text{ret } \langle\rangle \circlearrowleft r]\!] \sqsubseteq_R \lightning_A$

The COMPILE-EXEC obligation is straightforward and standard for forward simulation using an abstraction relation. Recovery execution is more interesting, and requires two obligations. RECOVER-RET is somewhat of a technicality for the case where the system crashes before any operations are run, in which case the only impact of recovery should be an abstract crash step.

RECOVER-OP is the main obligation of recovery refinement; we illustrate the obligation (with $[\![c(o) \circlearrowleft r]\!]$ unfolded) in Figure 6. Intuitively, the developer must show that recovery correctly handles a crash during any operation. Recall that the semantics of the abstract layer state that operations are atomic with respect to crashes, so this means a crash during $c(o)$ combined with recovery should simulate either just an abstract crash step $\lightning_A$ or the entire operation followed by a crash step $\text{step}(o) \cdot \lightning_A$. Proving this obligation is non-trivial since recovery execution internally loops for crashes during recovery. We later show how we use Crash Hoare Logic's idempotence principle for recovery, which gives an inductive invariant to prove RECOVER-OP.

The last obligation, INIT-ABS, the only one for initialization, is not stated in terms of refinement. The reason is that it serves as the base case for simulation between the compiled code and the specification: it constructs an abstract state satisfying the abstraction relation, as long as initialization succeeds. Once the abstraction relation is established, the rest of the obligations apply, and because they are all refinements, they maintain the abstraction relation with some abstract state.

4.2 Correctness

We now show that recovery refinement is strong enough to guarantee that the behavior of compiled programs is preserved, even with intermediate crashes and recoveries. Throughout this subsection, we'll assume we have an implementation $M = (c, r, i)$ that is a recovery refinement of interface L_A into L_C, where the associated abstraction relation is R, and will leave off the M subscript in $C_M(p)$.

The obligations in recovery refinement only talk about single operations from the abstract layer. We start by proving that we can extend refinement to whole abstract programs for crash-free execution (this is a standard *forward simulation* [24]):

Theorem 4. For any program $e : \text{Proc}_A(T)$, its compiled version refines the abstract program; that is, the following holds:

$$[\![C(e)]\!] \sqsubseteq_R [\![e]\!]$$

Proof Sketch. The proof is by induction over e. The base cases follow from the recovery refinement conditions. For the bind $e_1\ e_2$ case, unfolding the definition of $\sqsubseteq_R$, we have as inductive hypotheses

$$R\,;\, [\![C(e_1)]\!] \subseteq v \leftarrow [\![e_1]\!]\,;\, R\,;\, \text{ret } v$$
$$\forall v.\, R\,;\, [\![C(e_2(v))]\!] \subseteq v' \leftarrow [\![e_2(v)]\!]\,;\, R\,;\, \text{ret } v'$$

The proof proceeds by showing a sequence of inequalities:

$$\begin{aligned} & R\,;\ [\![C(\text{bind } e_1\ e_2)]\!] \\ & \subseteq R\,;\ (x \leftarrow [\![C(e_1)]\!]\,;\ [\![C(e_2(x))]\!]) \\ & \subseteq x \leftarrow [\![e_1]\!]\,;\ R\,;\ [\![C(e_2(x))]\!] \\ & \subseteq x \leftarrow [\![e_1]\!]\,;\ v \leftarrow [\![e_2(x)]\!]\,;\ R\,;\ \text{ret } v \\ & \subseteq v \leftarrow [\![\text{bind } e_1\ e_2]\!]\,;\ R\,;\ \text{ret } v \end{aligned}$$

□

There's nothing too interesting in Theorem 4, except perhaps that its proof follows from the algebraic structure of transition relations. However, the analogous result for recovery execution is the crux of the Argosy metatheory:

Theorem 5. For any program $p : \text{Proc}_A(T)$ and recovery procedure $r_A : \text{Proc}_A(\text{Unit})$, the following refinement holds:

$$[\![C(p) \circlearrowleft C^{\circlearrowleft}(r_A)]\!] \sqsubseteq_R [\![p \circlearrowleft r_A]\!]$$

What does this theorem accomplish? First, recall that $C^{\circlearrowleft}(r_A) = \text{bind } r\ C(r_A)$; it includes the layer recovery procedure r in addition to the abstract recovery procedure r_A. The idea is that running this r after a $\lightning_C$ is analogous to a $\lightning_A$ transition, at which point it makes sense to run $C(r_a)$. The theorem re-arranges the left-hand side to prove this end-to-end result, including a high-level recovery procedure, from the per-operation obligation RECOVER-RET.

Proof Sketch. The key part of the proof is to decompose running the recovery $C^{\circlearrowleft}(r_A)$ into first running r, and then running $C(r_A)$ with r as its "sub-recovery" procedure. That is, we first prove:

$$\begin{aligned} & [\![C^{\circlearrowleft}(r_A)]\!]_{\lightning}{}^{\star} \cdot [\![C^{\circlearrowleft}(r_A)]\!] \\ & \quad = [\![r]\!]_{\lightning}{}^{\star} \cdot [\![r]\!] \cdot [\![C(r_A) \circlearrowleft r]\!]^{\star} \cdot [\![C(r_A)]\!] \end{aligned}$$

This equivalence is shown entirely using Kleene algebra identities. Once we have this equivalence, the rest of the proof is intuitive. First, we have that r recovers the system (after crashing some number of times) to a state where we can execute other L_C programs. Then, we can reason about $[\![C(r_A) \circlearrowleft r]\!]$ as a crashing program in L_C, and show that it refines $[\![r_A]\!]_{\lightning}$; this follows by using the simulation theorem from Kleene algebra, which states that $p \sqsubseteq_R q$ implies $p^{\star} \sqsubseteq_R q^{\star}$. Finally, $[\![C(r_A)]\!]$ is a crash-free execution, so it refines $[\![r_A]\!]$ by Theorem 4. □

These results show that abstraction is preserved across a *single* program's complete execution or crash and recovery. What about an entire *interaction* with the system, with multiple programs, each of which may crash and recover? We extend the single-execution correctness result of Theorem 4 to interactions represented as a sequence of programs and a user-specified recovery procedure, all written at the abstract layer. As before, the intuitive correctness definition is that the behavior of this abstract sequence is preserved by compilation. Formally, we model such a sequence as an inductive type ProcSeq generated by:

- seqCons $e\ f$, where e has type $\text{Proc}_L(A)$ for some A and is the next program to run, while $f : \text{Option}(A) \to \text{ProcSeq}(R)$ determines what to run afterward.
- seqNil, the empty list.

To define execution of such a sequence, we first define a helper function which non-deterministically decides whether to execute or crash a program, and tags the result:

$$\begin{aligned} \text{exec\_or\_rexec}(e, r) \triangleq\ & (x \leftarrow [\![e]\!]\,;\ \text{ret some}(x)) \\ & + (x \leftarrow [\![e \circlearrowleft r]\!]\,;\ \text{ret none}) \end{aligned}$$

Execution of a sequence ps with recovery procedure r is defined by

$$\begin{aligned} & \text{execSeq}(\text{seqNil}, r) \triangleq \text{ret nil} \\ & \text{execSeq}(\text{seqCons } e\ f, r) \triangleq \\ & \quad x \leftarrow \text{exec\_or\_rexec}(e, r)\,; \\ & \quad l \leftarrow \text{execSeq}(f(x), r)\,;\ \text{ret } (x :: l) \end{aligned}$$

This definition recursively executes or crashes each program in the sequence, passing the resulting values to a function that decides which to execute next. All of the intermediate results are accumulated in a heterogeneous list, which is returned at the end. The final list represents what the user may observe as they execute the whole sequence.

We then have a function $\text{compileSeq}_M(ps)$ that compiles the programs in a sequence ps. Given ps and a user-specified recovery procedure r_A, we define a complete execution of the compiled program by the relation

$$\begin{aligned} & \text{execCompile}_M(ps, r_A) \triangleq \\ & \quad x \leftarrow [\![i]\!]\,; \\ & \quad \text{if } x \text{ then} \\ & \quad\quad v \leftarrow \text{execSeq}(\text{compileSeq}_M(ps), C_M^{\circlearrowleft}(r_A))\,; \\ & \quad\quad \text{ret some}(v) \\ & \quad \text{else (ret none)} \end{aligned}$$

which models running the initialization procedure, and if it succeeds, running the compiled sequence. For simplicity we assume here that the program does not crash during initialization.

Finally, we will want to consider all possible outputs of programs when run from initialized states satisfying P. Given $q : \text{Rel}(S, S, A)$, we define the predicate

$$\text{output}(q) \triangleq \lambda v.\ \exists s_1, s_2.\ P(s_1) \wedge q(s_1, s_2, v)$$

The following theorem shows that compilation using a recovery refinement M preserves an entire interaction consisting of a sequence of programs:

Theorem 6 (Correctness for sequences)**.**
If $\text{some}(v) \in \text{output}(\text{execCompile}_M(ps, r_A))$, then $v \in \text{output}(\text{execSeq}(ps, r_A))$.

Proof Sketch. If $\text{some}(v)$ is in the output of the compiled program, then initialization succeeded for that execution.

INIT-ABS shows that the abstraction holds after initialization. We then proceed by induction on ps. Based on whether the next program in the sequence executes normally or crashes, we use one of the two preceding theorems to show that abstraction is preserved. This then guarantees that the return values for each intermediate compiled program are equal to some possible return value for the original. □

Theorem 6 only guarantees that the return value of the executable code sequence matches some expected behavior from the abstract program sequence. The theorem is stronger than it may at first appear since these abstract programs can embed arbitrary Gallina functions to inspect intermediate results, and can maintain state between functions using the general bind $p\ f$ constructor common to all languages Proc_{L_A}. For example, ps could track all intermediate return values of L_A's operations and include them as part of the return value v.

Return values can capture all internal behavior but not any additional externally visible behavior emitted by operations; for example, we assume that clients of a file system written in Argosy have no way of directly observing low-level storage writes. Indeed, this low-level behavior would not even be part of the file system's specification.

4.3 Composition

The workflow for the developer is to construct a refinement of some intermediate abstraction L_I into L_C, and then to use the intermediate abstraction and prove a refinement of L_A into L_I, possibly repeating this several times with multiple intermediate interfaces — one verified file system, FSCQ, describes a stack with seven layers, and another, Yggdrasil, has five. In order to get a complete system, the developer needs to compose these recovery refinements and compile across the intermediate layer of abstraction.

The following theorem lets us combine two recovery refinements:

Theorem 7 (Composition)**.** If $M_1 = (c_1, r_1, i_1)$ is a recovery refinement of L_A into L_I, and $M_2 = (c_2, r_2, i_2)$ is a recovery refinement of L_I into L_C, then the following implementation is a recovery refinement of L_A into L_C:

$$
\begin{aligned}
c &= \lambda o.\, C_{M_2}(c_1(o)) \\
r &= (x \leftarrow r_2\,;\ C_{M_2}(r_1)) \\
i &= (x \leftarrow i_2\,;\ \text{if } x \text{ then } C_{M_2}(i_1) \text{ else false})
\end{aligned}
$$

We write $M_2 \circ M_1$ for this composed implementation.

The proof of this theorem has two main ideas: using Kleene algebra laws (specifically denesting) to re-arrange the execution of the nested recovery procedure, and applying Theorem 5 on the recovery procedure compiled to L_I. Of course recovery refinement also requires proving that normal execution is preserved, but this is a comparatively simple application of Theorem 4.

Note that because the composed implementation is a recovery refinement, Theorem 6 above applies to the composition $M_2 \circ M_1$ and the developer has a proof of correctness for the entire stack, built from modular proofs that only reason about adjacent interfaces.

5 Embedding Crash Hoare Logic

So far we've described recovery refinement, but how does a user prove that an implementation is a recovery refinement? We have designed Argosy to separate out the metatheory of recovery refinement from the reasoning about implementation behavior needed to prove refinement. To address this latter program-specific reasoning we implemented Crash Hoare Logic (CHL), the logic used to verify the FSCQ file system [8]. We prove all the rules of CHL as theorems in our Coq development. We review the basics of CHL and then describe what its implementation in Argosy looks like.

CHL is a variant of Hoare logic which features a judgment $\{P\}\ e\ \circlearrowleft\ r\ \{Q\}\{Q_R\}$. This *recovery specification* of procedure e recovering with r has three parts: P and Q are the familiar pre- and postcondition from Hoare logic, while Q_R is a new recovery postcondition. The interpretation of this judgment is that if e runs in a state s_1 satisfying P, then:

- If the system does not crash, e will terminate and return some value v in a final state s_2 such that $Q(s_1, s_2, v)$ holds.
- If the system crashes and runs r for recovery, then r returns some value v in a final state s_2 such that $Q_R(s_1, s_2, v)$ holds.

The postconditions are full relations over the input state, output state, and return value, instead of the traditional version of Hoare logic in which they do not depend on the initial state. While not strictly necessary, we use this formulation since it often makes writing the postcondition and recovery postcondition more convenient.

This judgment can be encoded straightforwardly using the Kleene algebra semantics.[3] We first recast the pre- and postconditions as relations over pairs of an initial state and a final state:

$$\text{specRel}(P, Q) \triangleq \lambda s_1, s_2, v.\, P(s_1) \longrightarrow Q(s_1, s_2, v)$$

Then the recovery quadruple is defined by:

$$
\begin{aligned}
\{P\}\ e\ \circlearrowleft\ r\ \{Q\}\{Q_R\} \triangleq\ & (\llbracket e \rrbracket \subseteq \text{specRel}(P, Q)) \\
& \wedge (\llbracket e\ \circlearrowleft\ r \rrbracket \subseteq \text{specRel}(P, Q_R))
\end{aligned}
$$

Once we have proven recovery specs for the implementations of each operation in an interface, proving recovery refinement is straightforward: we just need to show that the

[3] In their use of CHL to verify FSCQ, Chen et al. [8] used a shallow embedding of separation logic to be able to write assertions using the separating conjunction $*$. We have not needed to use these connectives in our examples, but a similar shallow embedding could be used.

abstraction relation implies the precondition of each operation's spec, and conversely, that the post/recovery conditions imply the abstraction relation for the updated states. Specifically, for every operation o in the abstract interface, we need to show its implementation $c(o)$ satisfies the following specification, using R for the abstraction relation:

$$\begin{aligned}&\forall s_A.\ \{\lambda s.\ R(s, s_A)\}\\ &\quad c(o)\circlearrowleft r\\ &\{\lambda s, s', v.\ \exists s'_A.\ R(s', s'_A) \wedge \mathsf{step}(o)(s_A, s'_A, v)\}\\ &\{\lambda s, s', \_.\ \exists s'_A.\ R(s', s'_A) \wedge\\ &\qquad \left(\lightning_A(s_A, s'_A) \vee (\mathsf{step}(o) \cdot \lightning_A)(s_A, s'_A)\right)\}\end{aligned}$$

This is simply a slight rephrasing of RECOVER-OP and RECOVER-RET from the definition of recovery refinement.

Proving recovery specs directly is hard, since they mix reasoning about the behavior of the program e and the recovery procedure r. Instead, CHL relies on another judgment, a crash spec, of the form $\{P\}\ e\ \{Q\}\{Q_\lightning\}$. Crash specs have the same interpretation for the precondition P and postcondition Q. However, instead of the recovery condition, they have a crash condition $Q_\lightning$, where if e crashes in state s_2 at any point during its execution starting from s_1 (that is, if the program halts followed by the effects of a crash $\lightning$), then $Q_\lightning(s_1, s_2, \langle\rangle)$ must hold.[4] The crash specification is defined as

$$\begin{aligned}\{P\}\ e\ \{Q\}\{Q_\lightning\} \triangleq\ &(\llbracket e \rrbracket \subseteq \mathsf{specRel}(P, Q))\\ &\wedge (\llbracket e \rrbracket_\lightning \subseteq \mathsf{specRel}(P, Q_\lightning))\end{aligned}$$

Rules for proving crash specs are shown in Figure 7. The rule PRIM-OP lets us deduce specs for primitive operations. Its reading is straightforward: if the operation o executed normally, then the relation between the starting and ending states is precisely described by $\mathsf{step}(o)$, so this becomes the postcondition. Otherwise, if there was a crash, it either happened before or after $\mathsf{step}(o)$ finished, so the crash condition is the non-deterministic choice between $\lightning$ immediately occurring or taking place after o runs. We chain together crash specs with the rule SEQUENCING, a monadic variant of the usual Hoare sequencing rule. Finally, we have the rule CONSEQUENCE, which as usual lets us strengthen preconditions and weaken postconditions, while also weakening the crash condition.

Using these rules, we can first derive crash specs for programs and the recovery procedure itself. CHL then provides the following IDEMPOTENCE rule, which we use to derive a recovery spec from a crash spec:

IDEMPOTENCE

$$\frac{\{P\}\ e\ \{Q\}\{Q_\lightning\} \qquad \{P'\}\ r\ \{S\}\{Q_\lightning\} \qquad \forall s_1, s_2.\ Q_\lightning(s_1, s_2, \langle\rangle) \to P'(s_2)}{\{P\}\ e\circlearrowleft r\ \{Q\}\{S\}}$$

[4] The FSCQ paper [8] and corresponding thesis [6] also use the term "crash spec", but they use the term to refer to an invariant that holds if the program halts at any time, not incorporating the effect of a crash.

The premises of this rule ensure that the crash conditions for e and r followed by a crash must imply the precondition for r. This is necessary because r may itself crash. Then, in the derived recovery spec, the postcondition comes from the postcondition of e, and the recovery condition is r's postcondition. Note that the crash condition is this specification is an invariant over crashes during recovery. However, when applying CHL to recovery refinement, different operations can use different specifications for the recovery procedure, and thus can use different invariants (our examples do indeed exploit this property).

Although this rule captures the main principle behind going from crash to recovery specs, it is not convenient to use as stated. The reason is that often the crash spec one proves about the recovery procedure is of the form:

$$\forall a.\ \{P(a)\}\ r\ \{Q(a)\}\{Q_\lightning(a)\}$$

That is, we quantify at the meta-level over some variable a which all assertions in the Hoare quadruple depend upon. Often, this a represents state from an abstract interface, and then the assertions in the quadruple can state how the effects of r correspond to operations applied to this abstract state.

The problem that arises when we try to use the rule IDEMPOTENCE with such a specification is that we would need to pick a single fixed value to instantiate a with. However, this does not work: the abstract state we are simulating may change as a result of a crash, meaning that the crash condition of the recovery procedure does not imply the precondition with the same choice for a.

To resolve this issue, we prove the following stronger rule:

IDEMPOTENCE-GHOST

$$\frac{\begin{array}{c}\{P\}\ e\ \{Q\}\{Q_\lightning\} \qquad (\forall a.\ \{P'(a)\}\ r\ \{S(a)\}\{Q'_\lightning(a)\})\\ \forall s_1, s_2.\ Q_\lightning(s_1, s_2, \langle\rangle) \to \exists a.\ P'(a, s_2)\\ \forall s_1, s_2, a.\ Q'_\lightning(a)(s_1, s_2, \langle\rangle) \to \exists a'.\ P'(a', s_2)\end{array}}{\{P\}\ e\circlearrowleft r\ \{Q\}\{\lambda s_1, s_2.\ \exists a.\ S(a)(s_1, s_2)\}}$$

In this version, the a we quantify over at the meta (Coq) level is a kind of auxiliary "ghost-state", and it may change each time recovery crashes: from the crash condition followed by a crash step we only need to show that there exists *some* a' for which the precondition will hold. Below the line, the recovery condition then holds for some existentially quantified a'. As we'll see in the examples in section 6, we use this a to encode a kind of state transition system that recovery moves through.

CHL does not have a sequencing rule to extend a recovery spec $\{P\}\ e\circlearrowleft r\ \{Q\}\{Q_R\}$ with an additional recovery procedure r'. The only option in CHL is to re-prove the premises of the idempotence rule with the extended recovery procedure $\_ \leftarrow r;\ r'$. This is the limitation that makes it infeasible to reason about layered recovery with CHL alone. However, this is no longer a problem in the context of Argosy, since we only use CHL to prove recovery refinement for a single implementation at a time, and then use Argosy's general

PRIM-OP
$\{\text{True}\}\ o\ \{\text{step}(o)\}\{\lightning + \text{step}(o) \cdot \lightning\}$

RET
$\{\text{True}\}\ \text{ret}\ v\ \{\text{ret}\ v\}\{\text{ret}\ \langle\rangle\}$

SEQUENCING
$$\frac{\{P\}\ e\ \{S\}\{Q_\lightning\} \qquad (\forall s_1, s_2, v.\ S(s_1, s_2, v) \to S'(s_2)) \qquad (\forall v.\ \{S'\}\ f(v)\ \{Q\}\{Q_\lightning\})}{\{P\}\ x \leftarrow e;\ f(x)\ \{Q\}\{Q_\lightning\}}$$

CONSEQUENCE
$$\frac{P \to P' \qquad Q' \to Q \qquad Q'_\lightning \to Q_\lightning \qquad \{P'\}\ e\ \{Q'\}\{Q'_\lightning\}}{\{P\}\ e\ \{Q\}\{Q_\lightning\}}$$

Figure 7. Inference rules for crash specs.

transitivity theorems to reason about a whole stack of composed implementations.

6 Examples

We now return to the motivating example from the introduction, and describe recovery-refinement proofs for the disk replication and write-ahead logging implementations. These examples are intended to illustrate how CHL enables recovery reasoning within Argosy for individual refinements. The logging design we use is comparable to the log used in the original FSCQ paper [8] and in Yggdrasil [28], although it is simplified compared to the followup system DFSCQ [7] or the journaling in Linux's ext4 file system. We believe both implementations demonstrate the main issues that arise in crash and recovery reasoning and that Argosy could be used to verify designs that give better performance with more engineering work.

6.1 Disk Replication

Disk replication implements an interface exposing a robust one-disk single disk on top of two unreliable disks. We assume that at least one of the two disks in the system is functional, so the state in the two-disk interface consists of either two active disks, written $\text{TwoDisks}(d_0, d_1)$ or a single disk tagged with an identifier id, written $\text{OneDisk}(id, d)$. The interface provides operations to read, write, and get the size of a disk, all taking a disk id as a parameter. Just before each operation, if both disks are active, one of them may fail (this is independent from system crashes, which halt execution and trigger recovery). The system may thus transition from $\text{TwoDisks}(d_0, d_1)$ to $\text{OneDisk}(id, d_{id})$, at which point the system will continue to run with a single disk. If an operation attempts to access a failed disk, then it returns an error code.

The robust one-disk interface that replication implements is straightforward. The state consists of a single disk, again modeled as a list of blocks. Crashes halt system execution but leave the disk unchanged. This interface provides the same read, write, and size operations, without a disk id parameter.

We will refer to the implementations of these operations in the two-disk interface as Write, Read, and Size, respectively. These implementations are simple. $\text{Write}(a, blk)$ writes blk to address a in both of the disks. $\text{Read}(a)$ first tries to read a in disk 0. If this succeeds, it returns the corresponding value, and otherwise it reads and returns a from disk 1. Size similarly tries to query the size of disk 0, and then tries disk 1 if necessary. The initialization procedure checks that the two disks are the same size, and zeros out all blocks in both of them.

Setting aside crashes momentarily, why does this correctly implement the one-disk API? The relationship is captured by a simple abstraction relation maintained by replication: the two disks, if they are both active, are equal to some disk d representing the state of the one-disk interface. To state this more formally, we introduce some notation to give the status of the disk(s). Given a two-disk state s, we define $s \stackrel{?}{=} (d_0, d_1)$ as follows:

$$s \stackrel{?}{=} (d_0, d_1) \triangleq \begin{cases} d_0 = d'_0 \wedge d_1 = d'_1 & \text{if } s = \text{TwoDisks}(d'_0, d'_1) \\ d_0 = d & \text{if } s = \text{OneDisk}(0, d) \\ d_1 = d & \text{if } s = \text{OneDisk}(1, d) \end{cases}$$

Then the abstraction relation maintained by the implementation is $absr \triangleq \lambda s, d.\ s \stackrel{?}{=} (d, d)$.

However, this abstraction is broken if the system crashes in the middle of a write, when only the first disk has been updated. The situation is summarized by the halt spec we proved for Write, which is shown in Figure 8. The precondition assumes the abstraction relation holds between the initial state and some disk d. The halt condition has three cases, corresponding to a crash before the entire operation, after the entire operation, and in the middle of Write.

The recovery procedure Recv restores the abstraction relation in a situation like this. It works by iterating through all addresses, copying from disk 0 to disk 1. If it notices that one of the disks has failed (because an operation returns an error code), it stops. We have proven two specifications for this procedure, shown in Figure 9, because the effects of recovery depend on whether the disks were originally the same ("synchronized") or different ("out-of-sync"). In the former case, the disks stay the same in both the postcondition and

$$\{\lambda s_1.\ s_1 \stackrel{?}{=} (d, d)\}\ \mathsf{Write}(a, blk)$$
$$\{\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d[a := blk], d[a := blk])\}$$
$$\left\{\begin{array}{l}\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d[a := blk], d[a := blk]) \vee s_2 \stackrel{?}{=} (d, d) \\ \quad \vee\ s_2 \stackrel{?}{=} (d[a := blk], d)\end{array}\right\}$$

Figure 8. Halt spec for replicated disk writes. (d is universally quantified.)

Synchronized Spec:

$$\{\lambda s_1.\ s_1 \stackrel{?}{=} (d, d)\}\ \mathsf{Recv}$$
$$\{\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d, d)\}\{\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d, d)\}$$

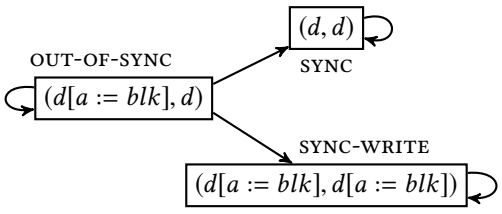
Out-of-sync Spec:

$$\{\lambda s_1.\ s_1 \stackrel{?}{=} (d[a := blk], d)\}\ \mathsf{Recv}$$
$$\{\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d[a := blk], d[a := blk]) \vee s_2 \stackrel{?}{=} (d, d)\}$$
$$\left\{\begin{array}{l}\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d[a := blk], d[a := blk]) \vee s_2 \stackrel{?}{=} (d, d) \\ \quad \vee\ s_2 \stackrel{?}{=} (d[a := blk], d)\end{array}\right\}$$

Figure 9. Halt specs for replicated disk recovery.

the halt condition. In the latter case, where disk 0 contains an additional write setting a to blk in d, the final state can vary. In the postcondition, either the write to a is copied to disk 1 or disk 1 fails, so that the abstraction relation is restored with $s_2 \stackrel{?}{=} (d[a := blk], d[a := blk])$; or disk 0 itself fails before the write is copied, in which case the abstraction relation holds with disk d instead. When this happens, it is as if the entire high-level write did not take place. The halt condition contains cases for these two scenarios, but also one where the disks remain out-of-sync.

It is helpful to visualize what can happen during recovery with the following state transition system:



Each node is labeled with what s is $\stackrel{?}{=}$ related to. If the disks are synchronized (SYNC and SYNC-WRITE states), then they stay so. However, if they are not (OUT-OF-SYNC) then the system can ultimately transition to either, or stay in the middle state while repeatedly crashing.

We use this state transition system to establish the premises of IDEMPOTENCE-GHOST, in order to prove a recovery spec for

Logical Layout

$$(\text{commit?}, (a_1, b_1) :: \ldots :: (a_n, b_n) :: \text{nil}, d_{old})$$

Physical Layout

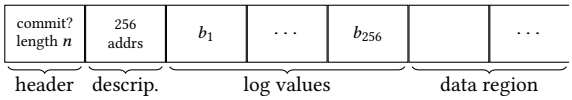


Figure 10. Physical and logical layout with write-ahead logging.

Write. The ghost state a we use in the rule is the state of a node in the above transition diagram, and on the basis of that state we use either the synchronized spec or out-of-sync spec for recovery. We obtain the following recovery spec for Write:

$$\{\lambda s_1.\ s_1 \stackrel{?}{=} (d, d)\}$$
$$\mathsf{Write}(a, blk)\ \circlearrowleft\ \mathsf{Recv}$$
$$\{\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d[a := blk], d[a := blk])\}$$
$$\{\lambda s_1, s_2, r.\ s_2 \stackrel{?}{=} (d[a := blk], d[a := blk]) \vee s_2 \stackrel{?}{=} (d, d)\}$$

The disjuncts of the recovery condition show that the abstraction relation holds for some appropriate disk, in which the write either did nothing or succeeded atomically, which is precisely what we need to establish the obligations of recovery refinement for this operation. We reason about the other operations similarly.

6.2 Write-Ahead Logging

Write-ahead logging implements the transactional disk interface described in Figure 2 on top of the one-disk interface. Recall that in the transactional interface, the state is a pair of disks (d_{old}, d_{new}) where d_{old} represents the persistent state of the disk before the current transaction, and d_{new} represents what the state will be if the current transaction is committed. Reads return values from d_{old}, while writes modify d_{new}. The commit operation replaces d_{old} with the current d_{new}, and crashes do the opposite.

To implement this interface on top of a single disk, we use write-ahead logging. The system uses a region of disk to keep track of the current transaction in the form of a log. The log holds the writes in d_{new} that are not yet committed to the data region d_{old}, which is represented by the rest of the disk. The transaction is stored separately so that upon crash the system can revert the disk by ignoring the pending writes and clearing the log. Committing these writes needs to be atomic even if the system crashes, so the commit operation first sets a *commit flag* to true on disk and then applies the writes in the log; if the system crashes in the middle, it checks the commit flag and if it is true, re-applies the log. Crucially,

re-applying parts of the log is equivalent to applying once, since these are overwrites of disk blocks.

The logical state tracked is a tuple (b, log, d) where b is a boolean commit flag, log is a list of address-value pairs representing the current transaction, and d is the data region. Given a physical state (a disk) s, we write $s \sim (b, log, d)$ to mean that s decodes into that tuple. The physical representation that encodes the logical state into disk blocks is depicted in Figure 10. The log header contains both the commit flag and the logical length of the log. The data in the log is stored in two fixed-size regions: a descriptor block encodes addresses and a log value region of 256 blocks holds the data for the current transaction's writes. The length of the log in the header determines how many of these address-value pairs are part of the log, while the rest are ignored. For simplicity we use a single descriptor block for addresses, and since we use 1KB blocks and 32-bit values for addresses, this limits transactions to 256 writes.

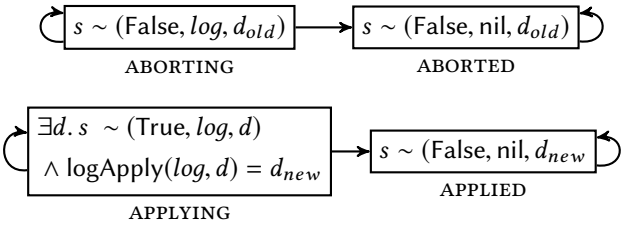
The implementation of read is straightforward, as it simply accesses the corresponding block in the data region, after first shifting the address by the size of the log. The implementation of write(a, blk) first accesses the header block to figure out how long the current transaction is. If there is no space left in the log, it returns an error code. Otherwise, it writes back the header block with the length incremented by 1, then updates the descriptor block to store a, and finally writes blk to the corresponding slot in the log value region. The commit implementation first reads the header block and writes it back with the committed bit set to true. It then calls a function doApply which actually applies all the operations: it loads the descriptor block to get the addresses for the writes that need to be applied, then iterates through each operation in the log and performs the corresponding write. Once all operations have been done, or if the commit flag was initially false, it writes back the header block setting the commit flag to false and the length to 0.

The abstraction relation is:

$$absr \triangleq \lambda s, (d_{old}, d_{new}).\ \exists log.\ s \sim (\mathsf{False}, log, d_{old}) \\ \wedge\ \mathsf{logApply}(log, d_{old}) = d_{new}$$

where logApply is just a pure Coq function which represents the effects of applying each operation in the log to a disk. The abstraction relation states the current transaction is uncommitted, and that the new disk is exactly what the logical disk would contain if the log were applied. The halt specs for the read and write operations are straightforward. For instance, the spec for write says that it appends an entry to the log list.

The recovery procedure checks if the log is committed, and if so, calls the doApply subroutine used to finish a commit. Much of the time the transaction is uncommitted and in case of a crash the commit flag is false, so to recover doApply simply clears the log by setting its length to 0. This reverts d_{new} to d_{old}, which is the specified crash behavior in the transactional disk API. The interesting case is when a crash occurs during a commit or recovery itself when the transaction in the log is committed but not yet applied. To carry out the idempotence proof, it's again helpful to visualize things as a state transition system. Suppose that before the crash, the last time the abstraction relation held, the abstract state was (d_{old}, d_{new}), and the logical list representing the log was log. Then, we have the following states and transitions:



where each node is labeled with what physical states it corresponds to. When the system crashes without committing and a partial transaction in the log it is in the ABORTING state. Eventually the transaction is aborted by clearing the log (ignoring the partial transaction), transitioning to ABORTED. The APPLYING state corresponds to a crash after the commit flag has been set but before all the writes in the log have been applied to the data region. The invariant here is that whatever the current physical data region is, if doApply were to apply everything in the log to it, it would end up equal to d_{new}. This continues to hold even after a crash in the middle of doApply, because applying a prefix of the log a second time has no effect. Eventually recovery will finish applying all of these operations and move to the APPLIED state. From either ABORTED or APPLIED the desired abstraction relation holds.

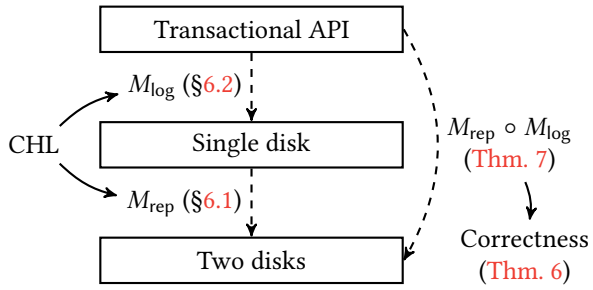


Figure 11. The verified composed stack. Each layer is independently verified using CHL as described in each implementation's subsection, then the two recovery refinements are composed to produce an overall correctness theorem.

6.3 Composing replication and write-ahead logging

We illustrate the overall result of our example development in Figure 11. Applying Theorem 7 to the two refinements we have established, we get a single implementation, which recovers the replicated disk followed by the log on crash.

Because the composition is a recovery refinement, Theorem 6 shows that for any sequence of interactions, including with crashes and recovery, the implementation behaves as the transactional API promises.

7 Implementation

We implemented Argosy in the Coq proof assistant [29]. The code is open source and available at https://github.com/mit-pdos/argosy. We give a breakdown of the code in the framework below, with non-comment, non-blank lines of code. The entire framework is 3,200 lines of code, around half of which is in re-usable libraries. The relation library includes many theorems that hold in Kleene algebras (adjusted to our typed, monadic setting) with some automation for equational reasoning.

Our two example refinements use a shared array library for reasoning about disks as arrays of blocks. Disk replication is around 1,300 lines, while logging, which has a more complicated recovery proof, is around 2,000. Much of the code in these examples comes from proving and especially stating many intermediate CHL specifications.

Component	Lines of code
Core framework	1,440
Relation library	1,020
Reusable libraries	740
Argosy total	3,200
Array library	530
Disk replication proof	1,350
Write-ahead logging proof	1,950
Examples total	3,830

In order to demonstrate that Argosy can be used to build working systems, we used Coq's extraction feature to run the composed logging and replication implementation. First we extract the Gallina implementation to Haskell. An interpreter written in Haskell runs the two-disk layer primitives, and a command-line interface exposes the logging API.

The resulting system has several trusted components. We trust that the semantics of the lowest layer (with two unreliable disks) is correctly implemented by our Haskell interpreter, and that the combination of extraction and the GHC compiler preserve the behavior of the Coq implementation. We trust that Coq checks the proofs correctly. Finally, we trust that the top-level specification reflects the intended behavior of the system.

8 Conclusion

Argosy is a framework for verifying storage system that supports layered recovery procedures and modular proofs. We introduce the notion of *recovery refinement*, a set of conditions for an implementation and its recovery procedure, that (1) guarantees correctness for clients of the specification, and (2) composes with other recovery refinements to prove a whole system correct. The semantics and refinement are modeled with combinators inspired by Kleene algebra, which informs our metatheory. To prove each implementation is a recovery refinement Argosy has an implementation of Crash Hoare Logic. We used Argosy to modularly verify an example of layered recovery, write-ahead logging implemented on top of a replicated disk.

Acknowledgments

Thanks to Anish Athalye, Jon Howell, Atalay İleri, and the anonymous reviewers, whose comments helped improve this paper. This research was supported by NSF awards CNS-1563763 and CCF-1836712, Google, and Oracle Labs.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: semantic foundations for networks. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 113–126.

[2] Robert Atkey. 2009. Parameterised notions of computation. *J. Funct. Program.* 19, 3-4 (2009), 335–376.

[3] Ryan Beckett, Eric Campbell, and Michael Greenberg. 2017. Kleene Algebra Modulo Theories. arXiv:cs.PL/1707.02894

[4] Ryan Beckett, Michael Greenberg, and David Walker. 2016. Temporal NetKAT. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 386–401.

[5] Thomas Braibant and Damien Pous. 2012. Deciding Kleene Algebras in Coq. *Logical Methods in Computer Science* 8, 1 (2012).

[6] Haogang Chen. 2016. *Certifying a Crash-safe File System.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[7] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 270–286.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, 18–37.

[9] Ernie Cohen. 1994. Lazy Caching in Kleene Algebra.

[10] Gidon Ernst. 2016. *A Verified POSIX-Compliant Flash File System - Modular Verification Technology & Crash Tolerance.* Ph.D. Dissertation. Augsburg University.

[11] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. 2016. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming* 131 (2016), 3–21.

[12] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 282–309.

[13] Sergey Goncharov, Lutz Schröder, and Till Mossakowski. 2009. Kleene Monads: Handling Iteration in a Framework of Generic Effects. In *Proceedings of the 3rd International Conference on Algebra and Coalgebra in Computer Science (CALCO'09)*. Springer-Verlag, Berlin, Heidelberg, 18–33. http://dl.acm.org/citation.cfm?id=1812941.1812945

[14] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Philadelphia, PA.

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, 1–17.

[16] Jan Kara. 2012. jbd2: issue cache flush after checkpointing even with internal journal. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=79feb521a44705262d15cc819a4117a447b11ea7.

[17] Jan Kara. 2014. [PATCH] ext4: Forbid journal_async_commit in data=ordered mode. http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977.

[18] Dexter Kozen. 1990. On Kleene Algebras and Closed Semirings. In *MFCS (Lecture Notes in Computer Science)*, Vol. 452. Springer, 26–47.

[19] Dexter Kozen. 1998. *Typed Kleene algebra*. Technical Report TR98-1669. Computer Science Department, Cornell University.

[20] Dexter Kozen and Maria-Christina Patron. 2000. Certification of Compiler Optimizations Using Kleene Algebra with Tests. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*. 568–582. https://doi.org/10.1007/3-540-44957-4_38

[21] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

[22] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A Rely-guarantee-based Simulation for Verifying Concurrent Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 455–468. https://doi.org/10.1145/2103656.2103711

[23] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, CA, 31–44.

[24] Nancy Lynch and Frits Vaandrager. 1995. Forward and Backward Simulations – Part I: Untimed Systems. *Information and Computation* 121, 2 (Sept. 1995), 214–233.

[25] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann.

[26] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-tolerant Resource Reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*. Pohang, South Korea.

[27] Damien Pous. 2013. Kleene Algebra with Tests and Coq Tools for while Programs. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. 180–196.

[28] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA.

[29] The Coq Development Team. 2019. The Coq Proof Assistant, version 8.9.0. https://doi.org/10.5281/zenodo.2554024

[30] J. von Wright. 2004. Towards a refinement algebra. *Science of Computer Programming* 51, 1 (2004), 23 – 45. https://doi.org/10.1016/j.scico.2003.09.002 Mathematics of Program Construction (MPC 2002).

[31] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, 357–368.