CAREER: Building Reliable Distributed Systems with Refinement Types

Lindsey Kuper (University of California, Santa Cruz)

1 Introduction

Today's most important computer systems are distributed systems: those that consist of multiple machines that communicate by sending messages over a network, and where individual machines or network connections may fail independently. Programming such systems is hard due to messages being reordered or delayed and the possibility of machines and network connections failing. Liveness guarantees (such as "messages are eventually received") may be impossible in such a setting — for instance, distributed consensus protocols such as Paxos [31, 32] are infamous for their lack of any termination guarantee [21] — and even safety properties (such as "received messages arrive in the order they were sent") can be challenging to prove. Distributed algorithms must be implemented in a defensive way to account for failures, and such designs are often expensive in terms of network bandwidth and storage space requirements; system designers may trade off fault tolerance along one or another dimension in exchange for efficiency. Furthermore, fault-tolerant systems hide their own bugs by nature [15], making them especially challenging to debug. Designers and implementers of distributed algorithms lack robust tools for reasoning about these trade-offs and challenges in real implementations, and for ensuring that those implementations satisfy the safety and liveness properties expected of them. The result is that, while protocols that ensure, say, a given message delivery order or a given data consistency policy are widely used in distributed systems, verification of the correctness of those protocols is less common, much less *machine-checked* proofs about *executable implementations*. **The goal of this project is to mechanically verify real, executable implementations of distributed systems, using rich verification capabilities integrated into a general-purpose, industrial-strength programming language.**

Why are existing approaches insufficient? For a verification approach to be practical and usable, we want to carry out verification directly in the same industrial-strength executable programming language used for implementation, with no subsequent transformation or extraction step necessary. Unfortunately, most existing approaches to *distributed systems* verification lack support for such a workflow:

- *Modeling languages* such as TLA+ [33] let designers and implementers write, test, and in some cases prove properties about models (i.e., executable specifications) of distributed systems. While the model can inform the development of the running implementation and is a useful design tool, there is no formal link between a model and a production implementation.
- *Verification-aware languages* such as Dafny [34] integrate a programming language with an Satisfiability Modulo Theories (SMT)-based program verifier. They let programmers specify pre- and postconditions of functions, as well as loop invariants, and can automatically check whether they hold. However, they lack the rich ecosystem of libraries and tools, and sometimes the attention to efficiency, that an industrial-strength general-purpose programming language has, making them impractical as an implementation language for real-world distributed systems.
- *Proof assistants* such as Coq [9], Agda [46], and Isabelle [72] let programmers implement a system within the proof assistant, prove properties, and then extract a verified executable implementation. A great advantage of these tools is that they (together with IDE support) provide excellent visibility into the state of an in-progress proof, helping the user understand what work remains to be done to complete the proof. However, the extraction process can be brittle and the extracted code can be difficult to integrate with libraries in the target language in a way that is both performant and not disconnected from the proofs. An additional issue is that a lack of SMT automation, like that offered by tools like Dafny, can make proofs more tedious and verbose than necessary.

For distributed systems in particular, there has also been much recent interest in SMT-solver-aided tool support (e.g., [6, 60, 24, 43]) for specifying and verifying particular correctness properties – for example, ensuring that no execution will violate a given policy specifying the consistency of replicated data. However, while these tools are useful, they do not entirely bridge the gap between specification and verified implementation. Designers of SMT-aided tools must walk a fine line between keeping the tool useful and usable for the programmer and keeping the analysis tractable for the solver. One way to ensure tractability is for the tools to operate on logical specifications [24, 43], designed to correspond to the first-order logic theories that SMT solvers support, rather than on executable code, resulting in a gap between the verified specification and the code that actually runs. If there is executable code involved [6, 60], there is no formal link between the specifications and the code. For example, a tool might report that it is safe to run program P with specification S_P against data store D with specification S_D, but the check only takes the specifications S_P and S_D into account; there is no checking that P indeed abides by S_P or D by S_D.

1.1 Research Plan

This project's approach to language-level distributed systems verification centers around *refinement types* [53, 75], data types that let programmers specify logical predicates that restrict, or refine, the set of values described by a type, and that can be checked by an off-the-shelf SMT solver. This mechanism is beginning to make its way into general-purpose, industrial-strength programming languages through tools such as *Liquid Haskell* [67], an extension to the Haskell programming language that adds support for refinement types. Conceptually, Liquid Haskell is a kind of hybrid of SMT-based program verifiers such as Dafny [34] and dependent-types-based interactive proof assistants such as Agda or Coq that leverage the Curry-Howard correspondence. Beyond giving more precise types to individual functions, Liquid Haskell's *reflection* [68, 69] facility lets programmers use refinement types to specify arbitrary "extrinsic" properties that can relate multiple functions, and then prove those properties by writing Haskell programs to inhabit the specified refinement types (see §2.1). Unlike Dafny (and its cousin F* [62, 41]), Liquid Haskell's integration with an existing programming language lets programmers work with pre-existing Haskell code and gradually port to Liquid Haskell, adding richer specifications to code as they go. Furthermore, verified Liquid Haskell libraries can be used directly in arbitrary Haskell programs, letting programmers seamlessly take advantage of formally-verified components from unverified code written in an industrial-strength, general-purpose language. The close integration with Haskell means that Liquid Haskell requires no extraction step (but also, no contortions to use Haskell's limited native support for dependent types [37]). In preliminary work, the PI and collaborators have used Liquid Haskell (§2.1) to verify the convergence of replicated data structures used for distributed programming (§2.2).

This project seeks to advance the state of the art of mechanized verification of distributed system implementations using refinement types, via two interrelated research thrusts:

- **Thrust 1: Verified Libraries for Distributed Systems (§3.1).** The PI and team will use Liquid Haskell to implement **the first mechanically verified executable library for *causal message delivery***, a foundational building block of distributed systems **(Task 1; §3.1.1)**. We will then use this verified causal message delivery layer to implement **verified libraries of *conflict-free replicated data types* (CRDTs) (Task 2; §3.1.2)** and **verified *causally-consistent distributed data stores* (Task 3; §3.1.3)**. Central to this research thrust is the concept of *modular* verification of each of these components. While executable verified causally-consistent data stores have been implemented before using a proof assistant extraction approach [36], and the PI's previous work [38] investigated CRDT convergence verification with Liquid Haskell, these previous verification efforts were *monolithic* in nature rather than building on an underlying verified messaging layer. The PI's proposed approach separates lower-level message delivery concerns from higher-level application semantics, and separately-verified components can be plugged together to get an end-to-end correctness guarantee. Our hypothesis is that higher-level consistency (Task 2) or convergence (Task 3) properties can

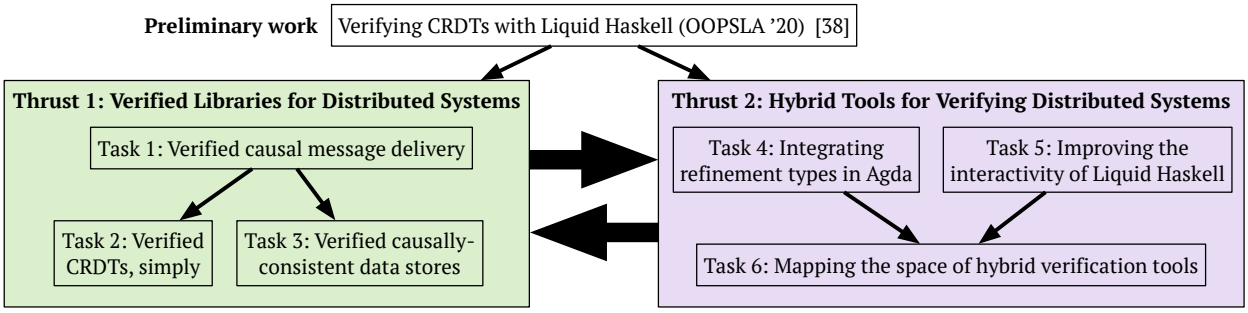

Figure 1: Overview of the research plan.

be more easily verified for applications that are built on top of a verified causal messaging layer (Task 1) than for monolithically implemented applications.

- **Thrust 2: Hybrid Tools for Verifying Distributed Systems (§3.2).** Dependent-type-based verification tools like Agda and refinement-type-based verification tools like Liquid Haskell each have usability advantages and disadvantages for distributed systems verification. In this thrust, the PI and team will explore ways to select the strengths of the two verification approaches by developing a hybrid approach. On the one hand, since some data types are easier to express and work with as SMT-decidable refinement types than as their dependently-typed, inductively defined counterparts, we will investigate **integrating refinement types into Agda (Task 4; §3.2.1)**. On the other hand, to provide better visibility into in-progress proof state, we will investigate **improving the interactivity of Liquid Haskell** by extending it with support for *typed holes* [22], a feature inspired by Agda **(Task 5; §3.2.2)**. More broadly, these efforts suggest that there is no hard boundary between interactive and automatic theorem proving; rather, there is a multidimensional spectrum of approaches, with different tools providing varying degrees of tightness of the human feedback loop and of granularity of feedback — as well as other possible axes of differentation. Therefore this research thrust will culminate with comprehensively surveying and **mapping the space of hybrid verification tools (Task 6; §3.2.3)**.

Figure 1 shows an overview of the proposed research plan. Within Thrust 1, Tasks 2 and 3 build on Task 1. Within Thrust 2, Task 6 builds on Tasks 4 and 5. Thrusts 1 and 2 are interrelated: the experience gained from the verification work in Thrust 1 will inform the design and implementation of the tools in Thrust 2, and the implementation of the tools in Thrust 2 will in turn aid the verification work in Thrust 1.

1.2 Education and Outreach Plan

As a field, distributed systems is both immediately practical and useful to students, and brimming with profound theoretical results. Yet distributed systems courses are often not offered at the undergraduate level, and the community has not converged on a standard undergraduate-level textbook, instead often opting to teach out of academic research papers [1]. Unsurprisingly, many undergraduate students (as well as industry practitioners) suffer from a lack of approachable distributed systems study materials. For would-be researchers (and would-be industrial consumers of distributed systems verification research), even less of an on-ramp exists.

The goal of the PI's education and outreach plan (described in detail in §4) is to create approachable entry points to distributed systems verification research, aimed in particular at undergraduate students and industry practitioners. The PI will create a new series of public-facing live online video broadcasts and recordings focusing on distributed systems verification, building on her existing experience creating publicly available distributed systems educational broadcasts and videos. Furthermore, the PI and team will create and distribute a series of *zines*

(short, self-published, and often hand-drawn and hand-illustrated pamphlets) about distributed systems research topics. There is a rich history of using zines for scientific communication (a recent example is the NSF-supported EPiQC project's zines about quantum computing [64]), and the PI has previously used zines as a teaching tool in her distributed systems courses. However, we do not only want students to *read* zines: as part of this project, the PI will integrate zine creation into the curriculum of her courses and subsequently into her team's research process by recruiting undergraduate students for summer research assistantships to create zines about the team's work. The student-created zines will be freely available online and will serve as instructional content in the PI's (and others') future courses as well as as an approachable introduction to the team's research. Moreover, the zine creation process itself will help students solidify their knowledge, develop their scientific and technical communication skills, and help integrate them into the broader research community.

1.3 Broader Impacts

People rely on complex distributed software systems in many aspects of their day-to-day professional and personal lives, from email and file sharing to videoconferencing, online gaming and more, and bugs in such systems have considerable economic consequences [65, 74]. In addition to fostering cross-pollination between the programming languages, software verification, and distributed systems communities, the proposed research aims to improve the overall trustworthiness and reliability of these systems. This project will provide verified, *immediately executable* implementations of widely used protocols and applications and release them as open source software artifacts. The educational component of our proposal will involve undergraduate students in cutting-edge research and give them an entry point to the broader research community, and the artifacts produced will be freely available online and will fill an growing need for distributed systems educational materials that are approachable both for students and for industry practitioners.

2 Background and Preliminary Work

This section gives a brief overview of refinement types and Liquid Haskell, and introduces the PI's preliminary work on distributed systems verification using Liquid Haskell.

2.1 Background: Refinement Types and Liquid Haskell

Refinement types [53, 75] let programmers specify data types augmented with logical predicates, called *refinement predicates*, that restrict the set of values that can inhabit the type. Depending on the expressivity of the language of refinement predicates, programmers can specify rich program properties using refinement types, sometimes at the expense of the decidability of type checking. Liquid Haskell avoids that problem by restricting refinement predicates to an SMT-decidable logic [52, 67]. For example, in Liquid Haskell we could define the type of even integers by refining the Haskell type `Int` using the refinement type `{ v:Int | v mod 2 == 0 }`, where `v mod 2 == 0` is the refinement predicate and `v:Int` binds the name `v` for values of type `Int` that appear in the refinement predicate. One could define an analogous refinement type for odd integers, and then write a Liquid Haskell function for adding them:

```
type EvenInt = { v:Int | v mod 2 == 0 }
type OddInt = { v:Int | v mod 2 == 1 }

oddAdd :: OddInt -> OddInt -> EvenInt
oddAdd x y = x + y
```

The type `OddInt` of the arguments to `oddAdd` expresses the *precondition* that `x` and `y` will be odd, and the return type `EvenInt` expresses the *postcondition* that `x + y` will evaluate to an even number. Liquid Haskell automatically

proves that such postconditions hold by generating verification conditions that are checked at compile time by the underlying SMT solver, Z3 [17]. If the solver finds a verification condition to be invalid, typechecking fails. If the return type of `oddAdd` had been `OddInt`, for instance, the above code would fail to typecheck.

Aside from preconditions and postconditions of individual functions, Liquid Haskell makes it possible to verify *extrinsic properties* that are not specific to any particular function's definition. For example, the type of `sumOdd` below expresses the extrinsic property that the sum of an odd and an even number is an odd number:

```
sumOdd :: x : OddInt -> y : EvenInt -> { _:Proof | (x + y) mod 2 == 1 }
sumOdd _ _ = ()
```

Here, `sumOdd` is a Haskell function that returns a *proof* that the sum of `x` and `y` is odd. (In Liquid Haskell, `Proof` is a type alias for Haskell's `()` (unit) type.) Because the proof of this particular property is easy for the SMT solver to carry out automatically, the body of the `sumOdd` function need not say anything but `()`. In general, however, programmers can specify arbitrary extrinsic properties in refinement types, including properties that refer to arbitrary Haskell functions via the notion of *reflection* [68]. The programmer can then prove those extrinsic properties by writing Haskell programs that inhabit those refinement types, using Liquid Haskell's provided *proof combinators* — with the help of the underlying SMT solver to simplify the construction of these proofs-as-programs [69, 68].

Liquid Haskell thus occupies a unique position at the intersection of SMT-based program verifiers such as Dafny [34], and proof assistants that leverage the Curry-Howard correspondence such as Coq [9] and Agda [46]. A Liquid Haskell program can consist of both application code like `oddAdd` (which runs at execution time, as usual) and verification code like `sumOdd` (which only "runs" at compile time), but, pleasantly, both are just Haskell programs, albeit annotated with refinement types. Being based on Haskell enables programmers to gradually port code from vanilla Haskell to Liquid Haskell, adding richer specifications to code as they go. Furthermore, verified Liquid Haskell libraries can be used directly in arbitrary Haskell programs, letting programmers take advantage of formally-verified components from unverified code written in an industrial-strength, general-purpose language.

Finally, unlike with a traditional proof assistant such as Coq or Agda, which requires an executable implementation to be *extracted* from the code written in the proof assistant's vernacular language, Liquid Haskell enables proving properties both *in* and *about* the code to be executed, resulting in *immediately executable* verified code with no need for a further extraction step.

2.2 Preliminary Work: Verified Conflict-Free Replicated Data Types with Liquid Haskell

In prior work [38], the PI and collaborators have used Liquid Haskell to develop a framework for programming distributed applications based on *conflict-free replicated data types* (CRDTs). Data replication is ubiquitous in distributed systems to guard against machine failures and keep data physically close to clients who need it, but it introduces the problem of keeping replicas consistent with one another in the face of network partitions and unpredictable message latency. CRDTs [59, 58, 51] are data structures whose operations must satisfy certain mathematical properties that can be leveraged to ensure *strong convergence* [59], meaning that replicas are guaranteed to have equivalent states given that they have received and applied the same *unordered* set of update operations. For example, consider an CRDT representing the contents of a shopping cart, replicated across data centers in Seattle, Frankfurt, Mumbai, and Tokyo for fault tolerance and data locality. Due to network partitions and message latency, updates to the cart's contents may arrive in an arbitrary order at each data center, but strong convergence ensures that under reliable message delivery assumptions, the replicas will eventually agree.

We used Liquid Haskell to prove strong convergence for several Haskell CRDT implementations (e.g., multisets, two-phase maps [59], causal trees [25]). In our proof development, the strong convergence of a replicated data structure depends on a proof of the commutativity of the operations that the data structure provides. Given such a commutativity proof, the rest of the reasoning to get strong convergence can be independent of the specifics of the data structure. We were therefore able to state and prove the strong convergence property at

the level of a generic interface in Liquid Haskell (that is, a Haskell *type class*), and then plug in a data-structure-specific *commutativity* proof for each CRDT we wanted to verify. Carrying out the proof in this modular fashion required us to extend Liquid Haskell to add the ability to state and prove properties at the type class level, itself a nontrivial task. We defined a CRDT as the below Liquid Haskell type class `VRDT`, and the required mathematical properties of a CRDT are expressed extrinsically as methods `lawCommut` and `lawCompatCommut`:

```
class VRDT t where
  type Op t
  apply :: t -> Op t -> t
  compat :: Op t -> Op t -> Bool
  compatS :: t -> Op t -> Bool
  lawCommut :: x : t -> op1 : Op t -> op2 : Op t
      -> { _: Proof | (compat op1 op2 && compatS x op1 && compatS x op2)
                      => (apply (apply x op1) op2 = apply (apply x op2) op1
                          && compatS (apply x op1) op2) }
  lawCompatCommut :: op1 : Op t -> op2 : Op t -> {_:Proof | compat op1 op2 = compat op2 op1}
```

This type class-based approach let us state and prove in Liquid Haskell a strong convergence property that applies to *any* `VRDT` instance. Despite the modularity enabled by type classes, though, each CRDT we considered required monolithic proofs of `lawCommut` and `lawCompatCommut`, requiring on the order of thousands of lines of Liquid Haskell code (and hours of solver time) for the more sophisticated CRDTs [38, Table 3]. In fact, to our knowledge, this is the largest Liquid Haskell proof development to date — which is not necessarily something to be proud of! The strenuous verification effort suggests that for distributed applications of this nature, a modular approach that separates lower-level message delivery concerns from higher-level data structure semantics is called for. Moreover, our experience carrying out large-scale proofs in Liquid Haskell highlights the need for usable hybrid verification tools that integrate solver automation with visibility into the in-progress proof state. Thrust 1 and Thrust 2 of the proposed research plan, discussed in the following section, will address these respective needs.

3 Research Plan

3.1 Thrust 1: Verified Libraries for Distributed Systems

In this Thrust, the PI and team will tackle verification of three representative distributed applications: a causal message delivery protocol, conflict-free replicated data types, and causally consistent data stores.

3.1.1 Task 1: Verified Causal Message Delivery

Protocols to ensure that messages are delivered in *causal order* [10, 12, 54, 13, 11] are a ubiquitous building block of distributed systems. A causal delivery protocol ensures that when a message m is delivered to a process p, any message sent "before" m (in the sense of Lamport [30]'s "happens-before") will have already been delivered to p. When a mechanism for causal message delivery is available, it simplifies the implementation of many important distributed algorithms, such as replicated data stores that must maintain causal consistency [3, 39], replicated data types [59], distributed snapshot protocols [2, 4], and applications that "involve human interaction and consist of large numbers of communication endpoints" [66]. While causal message delivery protocols are widely used in distributed systems, verification of the correctness of those protocols is less common, much less machine-checked proofs about executable implementations.

What can go wrong in the absence of causal delivery? Suppose Alice, Bob, and Carol are exchanging group text messages. Alice sends the message "I lost my wallet..." to the group, then finds the missing wallet and follows up with a "Found it!" message to the group. In this situation, depicted in Figure 2 (left), Alice has a reasonable

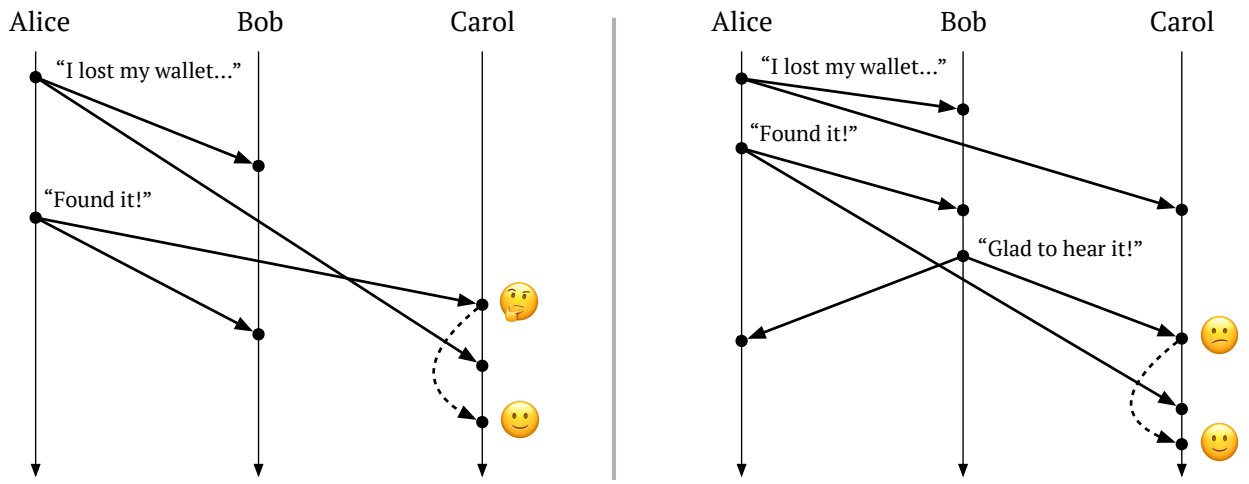

Figure 2: Two examples of executions that violate causal delivery.

expectation that Bob and Carol will see the messages in the order that she sent them, and such *first-in first-out (FIFO) delivery* is an aspect of causal message ordering. While FIFO delivery is already enforced[1] by standard networking protocols such as TCP [49], it is not enough to eliminate all violations of causality. In an execution such as that in Figure 2 (right), FIFO delivery is observed, and yet Carol sees Bob's message only after having seen Alice's initial "I lost my wallet..." message, so from Carol's perspective, Bob is being rude. The issue is that Bob's "Glad to hear it!" response *causally depends* on Alice's second message of "Found it!", yet Carol sees "Glad to hear it!" first. What is called for is a mechanism that will ensure that, for every message that is applied at a process, all of the messages on which it causally depends — comprising its *causal history* — are applied at that process first, regardless of who sent them.

A *causal broadcast* protocol [12, 11] addresses the problem by buffering messages at the receiving end until all causally preceding broadcast messages have been applied. Imagine a "mail clerk" at each process that intercepts incoming messages, and chooses whether, and when, to deliver each one (by handing it off to the above application layer and recording that it has been delivered), or buffer it for possible later delivery. The dashed arrows in Figure 2 represent the behavior of such a buffering mechanism. A typical implementation strategy is to have the sender of a message augment the message with metadata (for instance, a *vector clock* [42, 20, 56]) that summarizes that message's causal history in a way that can be efficiently checked on the receiver's end to determine whether the message needs to be buffered or can be applied immediately to the receiver's state. Although such mechanisms are well-known in the distributed systems literature [10, 12, 11], their implementation is "generally very delicate and error prone" [14], motivating the need for machine-verified implementations of causal delivery mechanisms that are usable in real, running code.

For this Task, the PI and team will implement a standard causal broadcast protocol (e.g., Birman et al.'s *CBCAST* protocol [11]) in Haskell, and use Liquid Haskell to express and mechanically prove a causal delivery property: messages can never be delivered to a process in an order that violates causality. In particular, an execution observes causal delivery if, for all messages m and m', for all processes p delivering both m and m',

$$m \to m' \implies \mathit{deliver}_p(m) \to_p \mathit{deliver}_p(m')$$

where $\to$ is Lamport's happens-before relation [30], $\mathit{deliver}_p$ is a message delivery event on p, and $\to_p$ is the total order of events on p. The behavior of our notional "mail clerk" is implemented by a predicate `deliverable` that

[1] TCP's FIFO ordering guarantee applies so long as the messages in question are sent in the same TCP session. For cross-session guarantees, additional mechanisms are necessary.

takes as arguments a message m and a process p, returning true if m can be delivered at p without causing a causality violation, and false otherwise. Our proof will establish the properties that must hold of any implementation of `deliverable` to ensure causal delivery of executions.

We will express this safety property using refinement types and prove that it holds using Liquid Haskell's theorem-proving facilities. For example, the following is a Liquid Haskell specification of a property that says that, if a message `m2` is deliverable at a given process `p`, then any causally preceding message `m1` is guaranteed to already have been delivered at `p`. This property is expressed as the refinement type of a function:

```
causalSafety
    :: p : Process -> m1 : Message -> m2 : Message
    -> { _:Proof | deliverable m2 p } -> { _:Proof | causallyBefore m1 m2 }
    -> { _:Proof | delivered m1 p }
```

In the `causalSafety` property above, the refinement type `{ _:Proof | deliverable m2 p }` is the type of a Liquid Haskell proof that `m2` is deliverable at `p`, where the occurrence of `deliverable` in the refinement refers not to a specification, but to the actual implementation of `deliverable` that is called at run time. It appears in the type of `causalSafety` via Liquid Haskell's *refinement reflection* mechanism that lets arbitrary (terminating) Haskell function calls appear in refinement predicates [68]. The types `{ _:Proof | causallyBefore m1 m2 }` and `{ _:Proof | delivered m1 p }` can likewise refer to running code. The programmer can then prove that `causalSafety` holds by inhabiting its type with a program, using Liquid Haskell's proof combinators [69].

The contribution of this Task will be, to our knowledge, the first formally verified executable causal broadcast library. Importantly, this verification approach is agnostic to the *content* of messages, enabling a variety of applications to be built on top of the underlying causal message delivery layer in a modular fashion. These applications can then leverage the causal delivery guarantee to establish higher-level, application-specific guarantees, such as convergence of CRDTs and causal consistency of distributed data stores, as we will see in the following two Tasks.

Related work. Much work on specification and verification of distributed systems has focused on specifying and verifying properties of models using tools such as TLA+ [33], rather than of executable implementations. The state of the art for machine-checked correctness proofs of executable distributed protocol implementations includes Verdi [73], IronFleet [26] and ShadowDB [55]. Verdi [73] is a Coq framework for implementing distributed systems; verified executable OCaml implementations can be extracted from Coq. IronFleet [26] uses the Dafny verification-aware language, which compiles both to verification conditions checked by an SMT solver and to executable code. Both Verdi and IronFleet have been used to verify safety properties (in particular, linearizability) of distributed consensus protocol implementations (Raft and Multi-Paxos, respectively) and of strongly-consistent key-value store implementations, and IronFleet additionally considers liveness properties. The ShadowDB project [55] uses a language called EventML that inverts the extraction workflow used in a proof assistant like Coq or Isabelle: instead of first carrying out a proof in a proof assistant and then extracting an executable implementation, the programmer writes code in EventML, which compiles both to a logical specification and to executable code that is automatically guaranteed to satisfy the specification, and correctness properties of the logical specification can then be proved using the Nuprl proof assistant. Schiper et al. [55] used this workflow to verify the correctness of a Paxos-based atomic broadcast protocol. None of these projects looked at causal broadcast or causal message ordering in particular.

3.1.2 Task 2: Verified CRDTs, Simply

As discussed earlier in §2.2, the PI and collaborators' preliminary work on CRDT verification [38] used Liquid Haskell to verify the convergence of several CRDT implementations, based on proving that their operations commute. CRDTs implemented in an *operation-based* style [59, 23] — as all those we considered were — typically require the existence of an underlying causal broadcast mechanism to deliver updates to replicas, following Shapiro

et al. [59, §2.4]. Further, Shapiro et al. [59] assume (reasonably!) that any preconditions that must be satisfied to enable an operation's execution (e.g., that a key must already be present in a dictionary if its value is to be updated) are already ensured by causal delivery.

This preliminary work, however, did *not* assume causal message delivery, which significantly complicated the implementation and verification effort. For example, our implementation of the *two-phase map* CRDT (a dictionary-like container type) had a "pending buffer" for updates that arrived out of order, and a collection of ad hoc, data-structure-specific rules to determine which updates should be buffered and which should be immediately applied. We observe that these mechanisms — which we were obliged to include to make our commutativity proof go through — resemble the delayed message buffer and the `deliverable` predicate used in standard implementations of causal broadcast, in e.g., Birman et al. [11]'s *CBCAST* protocol, but are specific to a particular application-level data structure and use an ad hoc delivery policy, rather than operating at the messaging layer and using the more general principle of causal delivery. We hypothesize that the separately-verified causal broadcast implementation of Task 1 would obviate the need for such ad hoc mechanisms and simplify the implementation and verification of CRDTs. For this Task, the PI and team will implement a library of verified-convergent CRDTs on the basis of the verified causal delivery library of Task 1. Our aim is to demonstrate that with this modular verification approach, the total amount of Liquid Haskell code necessary to verify CRDT convergence is an order of magnitude less than was required for our preliminary work.

Related work. Gomes et al. [23] use the Isabelle/HOL proof assistant [72] to implement and verify the strong convergence of several operation-based CRDTs. To carry out the proof, they bake in causal delivery as an underlying assumption, modeled by the "network axioms" in their proof development. Therefore, for strong convergence to hold for an actual deployed implementation of Gomes et al.'s CRDTs, the deployment environment would need to *provide* causal delivery. The work of Task 1 will implement just such an environment, with its safety verified by Liquid Haskell. Therefore, we plan to deploy the verified-convergent CRDTs of Task 2 atop the verified-safe causal broadcast protocol of Task 1 to get an "end-to-end" guarantee on top of a weaker network model that offers no causal delivery guarantee itself. Our verified CRDTs will also be *immediately executable* thanks to Liquid Haskell, unlike Gomes et al.'s, which would require an extraction step.

Zeller et al. [77] specify and prove convergence for a variety of state-based counter, register, and set CRDTs using Isabelle, while Nair et al. [44] present an automatic, SMT-based verification tool for specifying state-based CRDTs and verifying application-level properties of them. Neither Zeller et al. nor Nair et al. consider operation-based CRDTs, our focus here. Nagar and Jagannathan [43] address the question of automatically verifying strong convergence of various operation-based CRDTs (sets, lists, graphs) under different consistency assumptions provided by the underlying data store. Their CRDT specifications are not executable implementations, but written in an abstract specification language amenable to SMT solving, whereas our proposed verified CRDTs will be executable Haskell implementations and directly usable in real applications.

3.1.3 Task 3: Verified Causally Consistent Data Stores

The causal delivery library proposed for Task 1 is good for more than just CRDTs. A particularly compelling application is to use causal delivery to ensure *causal consistency* [3, 39] of data across a number of replicas. Out of dozens of distributed data consistency policies found in the literature [70], causal consistency represents an appealing "sweet spot" in the consistency/availability trade-off space, letting replica states diverge when necessary to preserve availability [40] while still ensuring that causal dependencies between operations are respected.

For this Task, the PI and team will build on the verified causal message delivery protocol of Task 1 to implement a verified, executable causally-consistent data store. Figure 3 shows an example application architecture for our proposed data store. A collection of (potentially geo-distributed) peer nodes, which we call the *cluster*, each run a causal broadcast protocol, along with application code that implements the data store. Clients of the data store communicate read and write requests to the nodes of the cluster; one or more clients may communicate

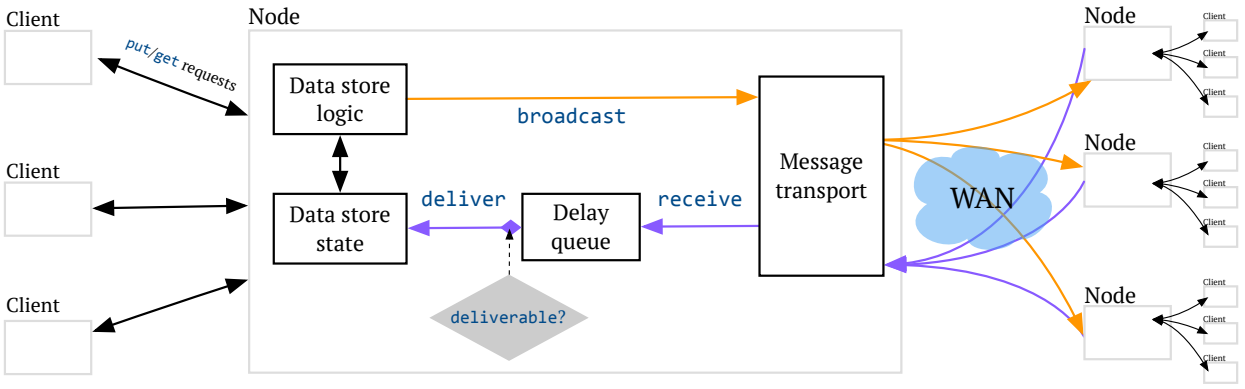

Figure 3: Architectural diagram of a causally-consistent data store implemented using a causal broadcast library. Each node is responsible for broadcasting messages to the other nodes in the cluster. Each of the nodes can support multiple clients.

with each node. The application instance on a node generates messages, broadcasts them to other nodes in the cluster, and delivers messages received from other nodes.

Verifying causal consistency of our data store will leverage the causal message delivery result of Task 1. However, causal consistency does not immediately follow from causal delivery at each of the nodes. For example, proving causal consistency will also require ensuring, via refinement types, that read and write operations on the data store are carried out in a total order *per node* (although, importantly, *not* necessarily a global total order). Furthermore, causal consistency requires the participation of clients to propagate causal metadata between each read and write operation, or else that clients be "sticky" to a particular cluster node [47]. We will use Liquid Haskell's refinement types to precisely express the behaviors required of both the data store application and of clients to preserve causal consistency.

Related work. Various SMT-powered verification tools [60, 24, 27] enable automatically verifying that a given application invariant or operation contract holds under a specified consistency policy, including causal consistency. However, rather than verifying that causal consistency itself is satisfied, these tools determine whether or not it is safe to execute a given operation under the *assumption* that that a given consistency policy is satisfied, or whether or not an application-level invariant will be satisfied given the consistency policies satisfied by individual operations. The goals of these lines of work are therefore complementary to ours: we *prove* a property that such tools could then leverage as an assumption to prove *application-level* properties, e.g., that a replicated bank account never has a negative balance. (Of course, it would also be possible to prove such application-level properties directly in Liquid Haskell as well.)

Chapar [36] presented a technique and Coq-based framework for mechanically verifying the causal consistency of distributed key-value store (KVS) implementations, with executable KVSes extracted from Coq. Lesani et al.'s approach effectively bakes a notion of causal message delivery into an operational semantics that specifies how a causally consistent KVS should behave. Lesani et al. then use the Chapar framework to check that a KVS implementation satisfies that specification. The executable KVSes extracted from Chapar can safely run on top of a messaging layer that does *not* provide causal delivery. Chapar is a monolithic implementation specific to the KVS use case, whereas our proposed approach factors out causal message delivery into a separate layer, agnostic to the content of messages. As with Task 2, we hypothesize that our modular approach will simplify the verification of the data store compared to the Chapar approach. Finally, another advantage of our proposed approach is that it is immediately executable code, with no need for an extraction step.

3.2 Thrust 2: Hybrid Tools for Verifying Distributed Systems

In this Thrust, the PI and team will turn their attention from applications to the verification tools themselves. Dependent-type-based verification tools like Agda and refinement-type-based verification tools like Liquid Haskell each have advantages and disadvantages. The PI and team propose to develop a hybrid approach that combines the strengths of the existing tools.

3.2.1 Task 4: Integrating Refinement Types in Agda

Liquid Haskell's SMT automation makes some oft-used data types easier to work with than they are in traditional proof assistants, such as Agda [46], that lack such automation. A simple example is `Fin n`, the type of finite sets with n elements. In the Agda standard library, `Fin n` is inductively defined [45]:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

Agda code that operates on expressions of type `Fin n` must handle both the `zero` case and the `suc` case, leading to verbose proofs. However (as noted in a comment in the same Agda standard library file), elements of `Fin n` can be seen as natural numbers in the set $\{m \mid m < n\}$, which is precisely what the corresponding refinement type looks like in Liquid Haskell:

```
type Fin N = { m : Nat | m < N }
```

Such types are useful for implementing verified distributed systems. For example, a *vector clock* [42, 20, 56], which is a type of logical clock that plays a central role in causal delivery protocols [11], is a vector of natural numbers of length N, where N is the number of processes participating in the system.[2] A vector clock index of type `Fin n` is therefore guaranteed to never index out of bounds — an extremely useful property. But the inductive style of the `Fin n` definition in Agda makes the data structure more tedious to work with, whereas in Liquid Haskell, the underlying solver's built-in theory of natural numbers makes such verbosity unnecessary.

Based on this observation, for this Task, the PI and team will explore adding support for Liquid-Haskell-style refinement types to Agda. Aside from being useful to the Agda community broadly, we hypothesize that adding refinement types to Agda could actually benefit the process of developing Liquid Haskell proofs. This is because, since Agda offers more visibility than Liquid Haskell into the in-progress proof state; a workflow that the PI and team have found useful in our preliminary work is to carry out a proof in Agda first, then port it to Liquid Haskell.[3] Refinement types in Agda could streamline this workflow by enabling more straightforward translation of proofs from Agda to Liquid Haskell.

Related work. Schmitty [71] is a project to provide bindings for SMT-LIB [8] (the common input language used by all mainstream SMT solvers) in Agda, and integrate those bindings via Agda's reflection mechanism, enabling automatic proving. Schmitty works by translating Agda propositions to SMT-LIB in accordance with the current theories loaded, and passing the resulting script to the SMT solver. Schmitty is currently under active development and depends on features in an as-yet-unreleased version of Agda. We hypothesize that, based on Schmitty, we could provide a basic notion of refinement types in Agda, and the PI and team are in active discussions with

[2] Like other logical clocks, vector clocks do not track physical time (which would be problematic in distributed computations that lack a global physical clock), but instead track only the order of events in an execution. Vector clocks provide a lightweight way for each process to keep track of how many messages it has seen and from whom they were sent, and for message senders to transmit information about the causal dependencies of each message along with the message.

[3] Thanks to reflection and SMT automation, we find that proofs typically get shorter when ported from Agda to Liquid Haskell, but Agda nevertheless aids in developing the proof in the first place.

the Schmitty developers about this possibility. A related project is SMTCoq [19], a Coq plug-in that enables communication with external SMT solvers.

Previous work by Atkey et al. [5] and Sekiyama et al. [57] has explored formalizing the relationship between data types such as the two alternate definitions of `Fin n` above. A promising approach is to leverage our understanding of these relationships to automatically translate types from standard Agda to Agda-extended-with-refinement-types in a semantics-preserving fashion, and then to Liquid Haskell.

3.2.2 Task 5: Improving the Interactivity of Liquid Haskell

Currently, a downside of Liquid Haskell for proof development is that it offers much less visibility into the in-progress proof state than tools such as Agda or Coq do. Currently, Liquid Haskell provides only coarse-grained feedback to the user: either it reports a type error, which means there is still more work to do to complete the proof, or it does not, which means the proof is done. An opportunity exists to extend Liquid Haskell to support an *interactive* proof development process, with finer-grained feedback to the user.

The interactive Agda proof development experience is enabled by a feature called *typed holes*. Typed holes allow Agda programmers to indicate parts of a proof that they need help with filling in. In an interactive environment (such as an IDE), the programmer can use holes to interact with the type checker by, for example, asking for the type expected by the hole (that is, the proposition that needs to be proven to fill in that hole). Additional IDE commands can assist the programmer in filling in holes [63]. For example, if the programmer has partially filled a hole with an expression whose *return* type matches the type expected by the hole, then an IDE command can generate new holes indicating the types of the arguments of the expression. For a hole on the right-hand-side of a definition, another IDE command can case-split on a pattern variable and generate new holes for each case.

For this Task, the PI and team will investigate improving the interactivity of Liquid Haskell by extending it with support for typed holes and interactive editing commands that take advantage of them. In doing so, we plan to leverage GHC Haskell's existing recently added support for typed holes [22], which was itself inspired by Agda. As a simple mock-up of how typed holes might work in Liquid Haskell, consider a `listLength` function that we want to prove has the same behavior as the built-in `len`.

```
listLength :: [a] -> Int
listLength [] = 0
listLength (x:xs) = 1 + listLength xs

listLengthProof :: xs:_ -> { _:Proof | listLength xs == len xs }
listLengthProof = _
```

The body of `listLengthProof` is a hole, written as `_`. Liquid Haskell might then generate the message:

```
Found `_' of type `xs:[a] -> { _:Proof | listLength xs == len xs }'.
       Consider a case split as in the body of `listLength'.
```

Performing the case split could be automated with a keystroke, resulting in code with two holes:

```
listLengthProof :: xs:_ -> { _:Proof | listLength xs == len xs }
listLengthProof [] = _
listLengthProof (y:ys) = _
```

For the first of these holes, Liquid Haskell could issue a message like

```
Found `_' of type `{ _:Proof | xs == [] && listLength xs == len xs }'.
       This can be completed with `()'.
```

The replacement of _ with () could again be automated with a keystroke, or manually completed. The PI and team have a history of collaboration with the Liquid Haskell developers, and we intend to work closely with them to design and develop this feature (see attached letter of collaboration).

3.2.3 Task 6: Mapping the Space of Hybrid Verification Tools

Tasks 4 and 5 suggest a blurring of the boundary between interactive theorem proving, exemplified by proof assistants such as Coq [9], Agda [46], and Isabelle [72], and automated theorem proving, exemplified by SMT solvers such as Z3 [17] and CVC4 [7] and tools that build on them. Instead, there is a multidimensional spectrum of hybrid automatic/interactive (sometimes called "auto-active" [35]) verification approaches, with different tools providing varying degrees of granularity of feedback provided to the human user, and different styles of specifying program properties, e.g., as types as opposed to as Floyd-Hoare-style assertions. Liquid Haskell represents one point in this multidimensional space, as do emerging proof assistants that integrate SMT automation such as Lean [18] and F* [62, 41], SMT-aided verification-aware languages such as Dafny [34] and Kaplan [29], and proof assistants like Agda if augmented with the aforementioned Schmitty [71]. Unfortunately, the research community's knowledge of such hybrid verification tools is fragmented: they are dots on a map, but we have no robust theory to tie them together. This to the tools being poorly understood and hence under-exploited.

For this Task, the PI and team will carry out a survey of the landscape of such hybrid language-based proof engineering tools. In addition to the above questions about granularity of feedback and specification styles, some of the questions that our survey will explore include: How does the tool address mismatches between the theories or data types of the solver and those of the host programming language? Does the tool isolate code that is only typechecked (because it is used only for verification purposes) from code that is executed at run time? How are solver-side proofs reified on the language side, if at all? What portions of the solver are used (for instance, what SMT theories are exposed), and what control does the user have over this choice, if any? By systematically surveying the landscape of these tools, this Task will contribute to a holistic scientific understanding of hybrid automatic/interactive verification approaches, and will aid future work on, for example, integrating refinement types and dependent types.

Related work. A recent survey of proof engineering tools by Ringer et al. deems proof assistants like Agda, Coq, and Isabelle to be in scope, and SMT-based verification languages like Dafny to be out of scope [50, §1.2].[4] Our survey will instead focus on the latter category, with particular attention paid to the tools' behavior at the solver/language boundary.

4 Education and Outreach Plan

As an educator, the PI works to engage and excite students, to ameliorate social obstacles to learning, and to build paths that invite students into research. Despite distributed systems being both theoretically profound *and* immediately practical and useful, distributed systems courses are often not offered at the undergraduate level, and many undergraduate students (as well as industry practitioners) suffer from a lack of approachable distributed systems study materials. For distributed systems *research* — and particularly for verification work, such as the proposed research agenda — even less of an on-ramp exists. Therefore the goal of the education and outreach plan is to build on the PI's experience teaching distributed systems to create approachable entry points to distributed systems research, especially for undergraduate students and industry practitioners.

[4] Ringer et al. do discuss *hammers*, which are collections of proof assistant tactics that leverage external solvers, including SMT solvers [50, §5.2.1]; our focus is on tools that use SMT more extensively.

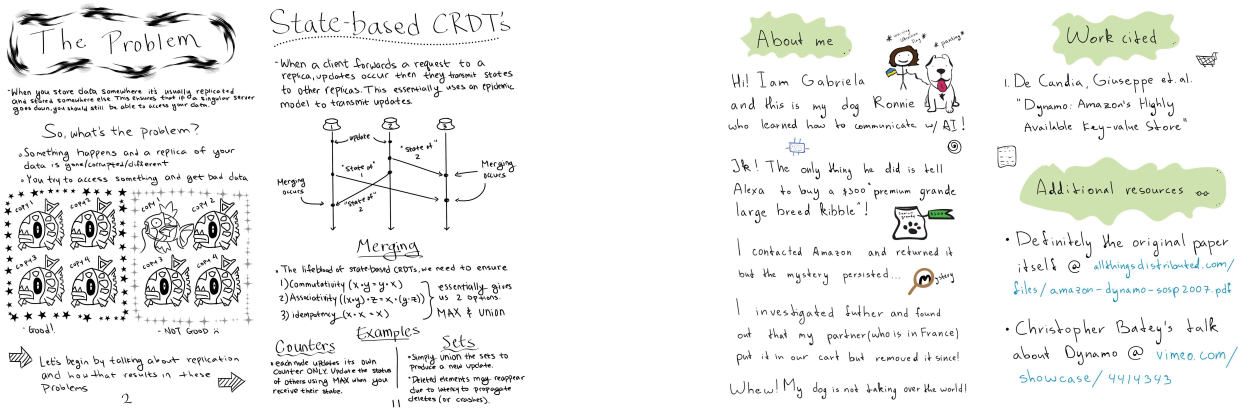

Figure 4: Excerpts of student-created zines from a previous offering of the PI's Distributed Systems course.

Innovation in teaching distributed systems. Each spring, the PI teaches the undergraduate distributed systems course (CSE 138) at UC Santa Cruz with 80-100 students per offering of the course. The PI uses an interactive style of teaching at the chalkboard (or, for remote teaching, a document camera with markers on paper) rather than with pre-prepared slides. Students have responded extremely well to this teaching style, remaining engaged and responsive in a way that is unusual for a large lecture course. In course evaluations, students regularly comment on how the hand-written and hand-drawn style keeps the course at a reasonable pace: fast enough to be interesting, yet not so fast that students disengage. In spring 2020 and spring 2021, the PI used the online streaming service Twitch.tv to broadcast these lectures live to a worldwide public audience (using a separate, private discussion platform for enrolled students). Since March 2020, the YouTube recordings of the lectures have been collectively viewed over 60,000 times and are popular both with students and practitioners. As part of the education and outreach plan, the PI will build on her existing teaching experience and existing audience to create a new series of live broadcasts and video recordings focusing on distributed systems verification, and will integrate these materials into the existing course and disseminate them broadly on online platforms.

Creation and dissemination of educational zines. There is a rich history of using *zines* — short, self-published, often hand-lettered and hand-illustrated pamphlets — for scientific and technical communication; a recent example is the NSF-supported EPiQC project's zines about quantum computing [64]. In a previous offering of the CSE 138 course, the PI introduced an optional assignment in which students created zines about distributed systems topics. In this initial pilot, 16 students created 12-16 page zines (with titles such as "What Makes the Web Fast?" and "Let's Talk Replication in Distributed Systems"); Figure 4 shows excerpts of student-created zines. In addition to the zines serving a need for approachable learning materials in future iterations of the course, the zine creation process itself helps students solidify their understanding of the course material and improve their scientific communication skills. There are well-known benefits [48, 16] to involving undergraduate students in the teaching process as peer or "near-peer" teachers in STEM courses. Yang [76] discusses how the act of zine creation in particular can foster scientific literacy and learning gains, as students go beyond consumption of information to active participation in the creation of content.

Based on this successful pilot, we plan to deeply integrate zine creation into the proposed project's education and outreach plan. During each summer of the five-year project scope, the PI will recruit undergraduate students from the CSE 138 course into summer research assistantships on her team to create zines about the team's work. The students involved in the project will be compensated by department funds, academic credit, and/or REU funds. The zines that the students create will be freely available online and will serve as instructional content in the PI's (and others') future courses and as an approachable public-facing introduction to the team's research. To

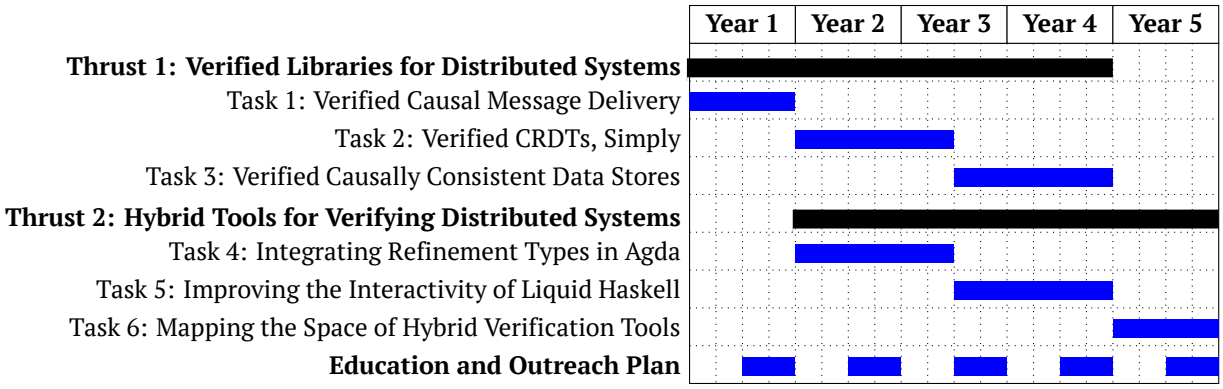

Figure 5: Project timeline.

ensure that we prioritize creating high-quality zines, we have also budgeted support for Julia Evans, a well-known professional computing zine creator [28] and a frequent collaborator with the PI on outreach work, to consult with the PI's team on the zine creation process (see attached letter of collaboration). For the undergraduate research assistants, the act of zine creation will help them build understanding of the team's work, develop their scientific communication skills, and integrate them as active and productive members of the PI's team, giving them an entry point to the broader research community.

Previous outreach. The PI has a strong track record of outreach. Since 2014, the PI has organized !!Con (pronounced "bang bang con"), a series of conferences that features ten-minute talks on the joy, excitement, and surprise of computing, and that fosters what one attendee describes as "a counterculture of curiosity" [61]. The PI led a team of volunteer organizers that brought !!Con to UC Santa Cruz as *!!Con West* for the first time in 2019 and 2020. The PI also serves on the board of the Exclamation Foundation, the nonprofit entity that oversees the ongoing organization of !!Con events. The PI plans to continue to organize !!Con and !!Con West over the five years of the proposed project. The PI also co-organized the Programming Languages Mentoring Workshop (PLMW) at ICFP '20 and '21, gave talks at PLMW at POPL '20 and PLDI '21, served on a panel of emerging researchers at PLMW at POPL '16, and is currently mentoring a student through the SIGPLAN-M Long-Term Mentoring program.

5 Project Timeline

The proposed project budget includes support for one full-time graduate student who will work with the PI on all aspects of the project. Figure 5 shows our proposed timeline for the five-year time frame of the proposal. Thrust 1 spans years 1-4, with year 1 focused on the development of a verified causal message delivery library (Task 1). In years 2-4, we will develop a library of verified CRDTs (Task 2) and verified causally consistent data stores (Task 3), building on Task 1. Thrust 2 overlaps with Thrust 1 and spans years 2-5. In years 2-4 we will explore integrating refinement types into Agda (Task 4) and adding interactive proof development to Liquid Haskell (Task 5). In year 5 we will focus on comprehensively surveying the space of hybrid verification tools (Task 6), integrating the knowledge gathered from Tasks 1-5. For the education and outreach plan, during spring of each year, the PI's annually offered distributed systems course will incorporate zine creation, and undergraduate students recruited from the course will join the PI and team each summer to create research zines.

6 Prior NSF Support

The PI has not previously been a PI or co-PI on an NSF-funded award.

References

[1] Cristina L. Abad, Eduardo Ortiz-Holguin, and Edwin F. Boza. *Have We Reached Consensus? An Analysis of Distributed Systems Syllabi*, page 1082–1088. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450380621. URL https://doi.org/10.1145/3408877.3432409.

[2] Arup Acharya and B.R. Badrinath. Recording distributed snapshots based on causal order of message delivery. *Information Processing Letters*, 44(6):317 – 321, 1992. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(92)90107-7. URL http://www.sciencedirect.com/science/article/pii/0020019092901077.

[3] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995. doi: 10.1007/BF01784241. URL https://doi.org/10.1007/BF01784241.

[4] Sridhar Alagar and S. Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. *Information Processing Letters*, 50(6):311 – 316, 1994. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(94)00055-7. URL http://www.sciencedirect.com/science/article/pii/0020019094000557.

[5] Robert Atkey, Patricia Johann, and Neil Ghani. When is a type refinement an inductive type? In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures*, pages 72–87, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19805-2.

[6] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741972. URL https://doi.org/10.1145/2741948.2741972.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://smt-lib.org, 2021.

[9] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642058809.

[10] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987. ISSN 0163-5980. doi: 10.1145/37499.37515. URL https://doi.org/10.1145/37499.37515.

[11] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991. ISSN 0734-2071. doi: 10.1145/128738.128742. URL https://doi.org/10.1145/128738.128742.

[12] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987. ISSN 0734-2071. doi: 10.1145/7351.7478. URL https://doi.org/10.1145/7351.7478.

[13] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987. ISSN 0734-2071. doi: 10.1145/7351.7478. URL https://doi.org/10.1145/7351.7478.

[14] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 626–638, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009888. URL https://doi.org/10.1145/3009837.3009888.

[15] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, 2007. URL http://dx.doi.org/10.1145/1281100.1281103.

[16] H. E. Chrispeels, M. L. Klosterman, J. B. Martin, S. R. Lundy, J. M. Watkins, C. L. Gibson, and G. K. Muday. Undergraduates achieve learning gains in plant genetics through peer teaching of secondary students. *CBE Life Sciences Education*, 13(4):641–652, 2014. doi: 10.1187/cbe.14-01-0007. URL https://www.lifescied.org/doi/10.1187/cbe.14-01-0007.

[17] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

[18] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[19] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak, editors, *Proceedings of the 29th International Conference on Computer Aided Verification (CAV '17)*, volume 10426 of *Lecture Notes in Computer Science*, pages 126–136. Springer, July 2017. URL http://www.cs.stanford.edu/~barrett/pubs/EMT+17.pdf. Heidelberg, Germany.

[20] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[21] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411. doi: 10.1145/3149.214121. URL https://doi.org/10.1145/3149.214121.

[22] GHC Team. Typed holes. https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/exts/typed_holes.html, 2021.

[23] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133933. URL https://doi.org/10.1145/3133933.

[24] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 371–384, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837625. URL https://doi.org/10.1145/2837614.2837625.

[25] Victor Grishchenko. Deep hypertext with embedded revision control implemented in regular expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, WikiSym '10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300568. doi: 10.1145/1832772.1832777. URL https://doi.org/10.1145/1832772.1832777.

[26] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. *IronFleet: Proving Practical Distributed Systems Correct*, page 1–17. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450338349. URL https://doi.org/10.1145/2815400.2815428.

[27] Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290387. URL https://doi.org/10.1145/3290387.

[28] Julia Evans. Wizard zines. https://wizardzines.com/, 2021.

[29] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 151–164, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310833. doi: 10.1145/2103656.2103675. URL https://doi.org/10.1145/2103656.2103675.

[30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7): 558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563.

[31] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

[32] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001. URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[33] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002. ISBN 032114306X.

[34] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642175104.

[35] K Rustan M Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, 2010.

[36] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 357–370, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837622. URL https://doi.org/10.1145/2837614.2837622.

[37] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, page 81–92, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323833. doi: 10.1145/2503778.2503786. URL https://doi.org/10.1145/2503778.2503786.

[38] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428284. URL https://doi.org/10.1145/3428284.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 401–416, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043593. URL https://doi.org/10.1145/2043556.2043593.

[40] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical report, The University of Texas at Austin, 2011.

[41] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, pages 30–59. Springer, 2019. doi: 10.1007/978-3-030-17184-1_2. URL https://fstar-lang.org/papers/metafstar.

[42] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[43] Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of crdts. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 459–477. Springer International Publishing, 2019. ISBN 978-3-030-25543-5.

[44] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In Peter Müller, editor, *Programming Languages and Systems*, pages 544–571, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44914-8.

[45] Nils Anders Danielsson, Matthew Daggitt, Guillaume Allais. The agda standard library: Finite sets. https://agda.github.io/agda-stdlib/Data.Fin.Base.html, 2021.

[46] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, page 230–266, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3642046517.

[47] Peter Bailis. Stickiness and client-server session guarantees. http://www.bailis.org/blog/stickiness-and-client-server-session-guarantees/, 2014.

[48] Heather Pon-Barry, Becky Wai-Ling Packard, and Audrey St. John. Expanding capacity and promoting inclusion in introductory computer science: a focus on near-peer mentor preparation and code review. *Computer Science Education*, 27(1):54–77, 2017. doi: 10.1080/08993408.2017.1333270. URL https://doi.org/10.1080/08993408.2017.1333270.

[49] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. URL http://www.rfc-editor.org/rfc/rfc793.txt. http://www.rfc-editor.org/rfc/rfc793.txt.

[50] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019. ISSN 2325-1107. doi: 10.1561/2500000045. URL http://dx.doi.org/10.1561/2500000045.

[51] Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, 2011. doi: 10.1016/j.jpdc.2010.12.006. URL https://doi.org/10.1016/j.jpdc.2010.12.006.

[52] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375602. URL https://doi.org/10.1145/1375581.1375602.

[53] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998. doi: 10.1109/32.713327.

[54] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, page 219–232, Berlin, Heidelberg, 1989. Springer-Verlag. ISBN 3540516875.

[55] N. Schiper, V. Rahli, R. Van Renesse, M. Bickford, and R. L. Constable. Developing correctly replicated databases using formal tools. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 395–406, 2014. doi: 10.1109/DSN.2014.45.

[56] Frank B Schmuck. *The use of efficient broadcast protocols in asynchronous distributed systems*. PhD thesis, 1988.

[57] Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 195–207, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676996. URL https://doi.org/10.1145/2676726.2676996.

[58] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 2011.

[59] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24550-3.

[60] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 413–424, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737981. URL https://doi.org/10.1145/2737924.2737981.

[61] Sumana Harihareswara. Toward a !!con aesthetic. https://recompilermag.com/issues/extras/toward-a-bangbangcon-aesthetic/, 2016.

[62] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 266–278, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi: 10.1145/2034773.2034811. URL https://doi.org/10.1145/2034773.2034811.

[63] The Agda Team. Holes and case splitting. https://agda.readthedocs.io/en/v2.6.2/getting-started/a-taste-of-agda.html#holes-and-case-splitting, 2021.

[64] The EPiQC Team. Zines. https://www.epiqc.cs.uchicago.edu/zines, 2021.

[65] Joseph Tsidulko. The 10 biggest cloud outages of 2020. https://www.crn.com/slide-shows/cloud/the-10-biggest-cloud-outages-of-2020, 2020.

[66] Robbert van Renesse. Causal controversy at le mont st.-michel. *SIGOPS Oper. Syst. Rev.*, 27(2):44–53, April 1993. ISSN 0163-5980. doi: 10.1145/155848.155857. URL https://doi.org/10.1145/155848.155857.

[67] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 269–282, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328739. doi: 10.1145/2628136.2628161. URL https://doi.org/10.1145/2628136.2628161.

[68] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158141. URL https://doi.org/10.1145/3158141.

[69] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in liquid haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 132–144, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358354. doi: 10.1145/3242744.3242756. URL https://doi.org/10.1145/3242744.3242756.

[70] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1), June 2016. ISSN 0360-0300. doi: 10.1145/2926965. URL https://doi.org/10.1145/2926965.

[71] Wen Kokke. Schmitty the solver. https://github.com/wenkokke/schmitty, 2021.

[72] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 33–38, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-71067-7.

[73] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737958. URL https://doi.org/10.1145/2737924.2737958.

[74] Sean Wolfe. Amazon's one hour of downtime on prime day may have cost it up to $100 million in lost sales. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7, 2018.

[75] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 249–257, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919874. doi: 10.1145/277650.277732. URL https://doi.org/10.1145/277650.277732.

[76] Andrew Yang. Engaging Participatory Literacy through Science Zines. *The American Biology Teacher*, 72(9): 573 – 577, 2010. doi: 10.1525/abt.2010.72.9.10. URL https://doi.org/10.1525/abt.2010.72.9.10.

[77] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43613-4.