Verifying Invariants of Lock-Free Data Structures with Rely-Guarantee and Refinement Types

COLIN S. GORDON, Drexel University
MICHAEL D. ERNST and DAN GROSSMAN, University of Washington
MATTHEW J. PARKINSON, Microsoft Research

Verifying invariants of fine-grained concurrent data structures is challenging, because interference from other threads may occur at any time. We propose a new way of proving invariants of fine-grained concurrent data structures: applying rely-guarantee reasoning to references in the concurrent setting. Rely-guarantee applied to references can verify bounds on thread interference without requiring a whole program to be verified.

This article provides three new results. First, it provides a new approach to preserving invariants and restricting usage of concurrent data structures. Our approach targets a space between simple type systems and modern concurrent program logics, offering an intermediate point between unverified code and full verification. Furthermore, it avoids sealing concurrent data structure implementations and can interact safely with unverified imperative code. Second, we demonstrate the approach's broad applicability through a series of case studies, using two implementations: an axiomatic Coq domain-specific language and a library for Liquid Haskell. Third, these two implementations allow us to compare and contrast verifications by interactive proof (Coq) and a weaker form that can be expressed using automatically-discharged dependent refinement types (Liquid Haskell).

CCS Concepts: • **Theory of computation → Type structures**; **Invariants**; **Program verification**; • **Computing methodologies → Concurrent programming languages;**

Additional Key Words and Phrases: Type systems, rely-guarantee, refinement types, concurrency, verification

ACM Reference Format:
Colin S. Gordon, Michael D. Ernst, Dan Grossman, and Matthew J. Parkinson. 2017. Verifying invariants of lock-free data structures with rely-guarantee and refinement types. ACM Trans. Program. Lang. Syst. 39, 3, Article 11 (May 2017), 54 pages.
DOI: http://dx.doi.org/10.1145/3064850

1. INTRODUCTION

Now that increasing core counts have replaced increasing clock frequencies in new CPUs, it is increasingly important to exploit parallelism in programs to improve application performance. In the general case, this requires introducing synchronization constructs to prevent threads from simultaneously interfering with each others' state.

This work was carried out while the first author was as the University of Washington, Samsung Research America, and Drexel University.

Authors' addresses: C. S. Gordon, Department of Computer Science, Drexel University, 3141 Chestnut St., University Crossings, Suite 100, Philadelphia, PA 19104 USA; email: csgordon@drexel.edu; M. D. Ernst and D. Grossman, Paul G. Allen School of Computer Science and Engineering, University of Washington, Box 352350, 185 E Stevens Way NE, Seattle, WA 98195 USA; emails: {mernst, djg}@cs.washington.edu; M. J. Parkinson, Microsoft Research Ltd., 21 Station Road, Cambridge, CB1 2FB United Kingdom; email: mattpark@microsoft.com.

DOI: http://dx.doi.org/10.1145/3064850

The simplest synchronization construct—the mutual exclusion lock—is effective, but it can slow down applications if it is used to protect too much data, because threads spend too much time waiting to acquire locks. Fine-grained locking—guarding different parts of a larger structure with separate locks—helps in many cases but not all. In the remaining cases, the only way to achieve acceptable scalability is to switch to *lock-free* data structures [Herlihy 1991; Herlihy and Shavit 2008]: concurrent data structures where instead of taking turns accessing data by waiting for locks, threads interact using hardware primitives that are less expensive and non-blocking but also significantly less powerful. Implementing these lock-free concurrent data structures is challenging on its own. In languages without enforced abstraction, ensuring that other parts of the program do not interfere incorrectly on these structures is an additional challenge.

The key challenge in proving properties of fine-grained concurrent data structures (FCDs [Turon et al. 2013])—whether lock-based or lock-free—is the treatment of interference from other threads: simultaneous side effects on shared state, whether a data race or not. An additional challenge is that frequently data structures are verified in isolation but then composed with larger, mostly unverified, programs, which may violate assumptions of the verification.

This article shows how to prove some safety properties—both traditional (e.g., $x > 0$) and two-state invariants [Liskov and Wing 1994] (e.g., $x_{\text{pre}} \leq x_{\text{post}}$)—of lock-free programs by building on recent work on *rely-guarantee references* [Gordon et al. 2013] (RGREFS). Ordinary reference types simply describe the type of value that may be read and written through a reference. Rely-guarantee references [Gordon et al. 2013] additionally restrict how the reference may be used: They summarize the capabilities granted to aliases, and they state a refinement [Freeman and Pfenning 1991] that is guaranteed to be preserved by actions through other aliases. For example, the refinement that a counter is positive is preserved by aliases that are restricted to incrementing the counter.

Rely-guarantee references target a middle ground between simple but weak basic-type systems and very powerful but correspondingly complex (for specification and automation) concurrent program logics. Standard-type systems in widespread use offer virtually no safety guarantees for shared-memory concurrency beyond basic memory- and type-safety. Full concurrent program logics can verify full functional correctness and more (e.g., linearizability [Vafeiadis et al. 2006; Liang and Feng 2013]) but require very high expertise to employ, and it is difficult to automate checking for the most sophisticated variants. Refinement types are amenable to effective inference and automatic checking [Rondon et al. 2008, 2010; Vazou et al. 2013, 2014b, 2015], and there is evidence that some working programmers may be willing to tolerate the specification burden of refinement types,[1] but their use with mutable state and concurrency has barely been explored. Rely-guarantee references [Gordon et al. 2013] were the first system to integrate refinement types with (sequential) aliased mutable state by combining reference types with a form of reference capability (for which there is also anecdotal evidence of developer support [Gordon et al. 2012]). This article focuses on the abilities of only the core elements of rely-guarantee references for verifying properties of concurrent data structures. Thus we explore a system that is less powerful than modern logics (such as Turon et al.'s Concurrent and Refined Separation Logic (CaReSL) [Turon et al. 2013], Iris [Jung et al. 2015], or Nanevski et al.'s Fine-grained Concurrent Separation Logic (FCSL) [Nanevski et al. 2014; Sergey et al. 2015a]) but, as this article shows, still very useful, with a lighter specification burden and the possibility of some inference and automated checking. Further extensions to improve

[1]Based on the increasing prevalence of presentations on refinement types at developer-focused venues [Jhala 2015, 2016; Vazou 2016].

flexibility are briefly discussed in Section 9, and extensions to derive full functional correctness proofs from RGREFS have been explored elsewhere [Gordon 2014].

The original RGREF design was unsound for concurrency due to assumptions about multiple reads being atomic but, more importantly, lacked a way to exploit dynamic observations of data structure properties in verifications: that is, a way to reflect a dynamic check of a property into the type system to locally strengthen *static* knowledge of data structure invariants. We fix the system for sound concurrent (and sequential) reasoning, extend its reasoning capabilities, and show via case studies that this is effective for specifying and verifying one- and two-state invariants of FCDs.

We implemented the type system and refinement approaches in two forms: a domain-specific language (DSL) implemented as an axiomatic shallow embedding in COQ and a library on top of Liquid Haskell [Vazou et al. 2013, 2014b]. We have used these to prove invariants for six lock-free data structures. Our implementations show feasibility of the approach from both theoretical and practical perspectives, exploring both expressivity (via COQ) and automation of a slightly restricted version for a real programming language (Haskell). Among these case studies are new results: We give the first mechanized proofs of invariants for a lock-free linearizable union-find implementation [Anderson and Woll 1991].

In summary, our contributions are as follows:

—The first refinement type system [Freeman and Pfenning 1991] for shared-memory concurrent heap structures.
—The first verification technique for FCDs that can verify invariants in the context of a mostly unverified program.
—Two implementations of concurrent RGREFS:
  —an axiomatic COQ DSL for verifying invariants by interactive proof, and
  —a Liquid Haskell library with slightly less power, but whose proofs are discharged automatically by a solver for satisfiability modulo theories (SMT).
—Evidence of concurrent RGREFS' utility in the form of mechanized or automatic proofs of one- and two-state invariants for classic FCDs [Treiber 1986; Michael and Scott 1996; Harris 2001] specified in terms of RGREFS.
—The first mechanized proof of invariants for a lock-free linearizable union-find [Anderson and Woll 1991].
—A soundness proof for extended sequential and concurrent RGREFS based on the Views Framework [Dinsdale-Young et al. 2013].

The implementations and example programs are available at https://github.com/csgordon/rgref-concurrent/ and https://github.com/csgordon/rghaskell/. A virtual machine image with all dependencies and compiled versions of both tools and examples is available at http://csgordon.github.io/rgref.

2. BACKGROUND: RELY-GUARANTEE AND RGREFS

Rely-guarantee reasoning is a well-established technique for specifying and verifying (bounds on) thread interference: how multiple threads modify state shared with other threads. This is an essential step for proving any properties of shared-memory concurrent programs, where without further care one thread may arbitrarily modify data in a way that violates the assumptions of another thread. Rely-guarantee reasoning originates in the concurrent program logic literature [Jones 1983]. It enables verification of a single thread modularly (in isolation) by characterizing possible interference between threads and asserting only properties robust to that interference. At points of parallel composition, the proofs of two threads can be checked for compatibility, ensuring the properties proven of threads in isolation hold when they are run concurrently.

There are four key ingredients in rely-guarantee reasoning, stated here for threads:

(1) A **rely**—a summary of possible behavior of other threads. Each thread's verification *relies on* this interference bound.
(2) A **guarantee**—a limit on the behavior of the current thread. This is a *guarantee* the current thread makes to other threads about the interference it may cause.
(3) **Stable** assertions—the only assertions that may be stated are those preserved by the rely: If an assertion is true in one state, and the state changes in a way allowed by the rely, then the assertion must also be true in the new state.
(4) A **compatibility** invariant—for any two threads that execute simultaneously, the rely of each thread includes at least the behavior in the other threads' guarantees.

The original exposition of these ideas [Jones 1983] takes place in the context of a concurrent Hoare logic for partial correctness. The core verification judgment is an extended Hoare triple $R, G \vdash \{P\}\ C\ \{Q\}$, which is a certification that command C, executed in a state satisfying global precondition P, that either diverges or terminates in a state satisfying global postcondition Q. This conclusion about a command's behavior is sound, assuming that interference from other threads is at most that described by the rely relation R and the command's own actions do not exceed those described by the guarantee G. All assertions P, Q are restricted to be stable with respect to the rely. Compatibility is ensured by the parallel composition rule:

$$\frac{R \vee G_2, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \quad R \vee G_1, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\}}{R, G_1 \vee G_2 \vdash \{P_1 \wedge P_2\}\ C_1 \,||\, C_2\ \{Q_1 \wedge Q_2\}}.$$

The above rule states that the parallel composition of two threads preserves their individual behaviors (up to termination) if each child's actions are included in the spawning thread's guarantee, and the child threads each tolerate at least the spawning thread's expected interference (rely) plus interference from the other forked thread (hence the disjunction $\vee$ of relations). The distinction between rely and guarantee enables verification of asymmetric protocols on state, such as producer-consumer relationships. In Section 4, we will see how this relationship between threads' rely and guarantee relations mirrors the relationship between rely and guarantee relations of RGref aliases.

Using global assertions and relations has the same modularity issues as the original Hoare logic [Hoare 1969], struggling with pointer-based programs and component reuse. But explicitly characterizing interference is valuable, so rely-guarantee reasoning has continued to be adapted by further work with better modularity properties [Vafeiadis and Parkinson 2007; Feng 2009; Wickerson et al. 2010; Dodds et al. 2009; Dinsdale-Young et al. 2010, 2013] or is used as a core principle in the soundness proofs for other logics [Turon et al. 2013; Nanevski et al. 2014; Sergey et al. 2015a; Jung et al. 2015], as discussed in Section 8.

2.1. Rely-Guarantee References

Gordon et al. adapted rely-guarantee reasoning to treat interference between aliases in a sequential setting similarly to interference between threads. The resulting type system, *rely-guarantee references* [Gordon et al. 2013], translates the four key ingredients to references, including nested references. The system is expressive enough to prove interesting refinements and to define a form of reference immutability [Gordon et al. 2012] (which, in previous work, we used to ensure data race freedom). However, the system is unsound for concurrent programs, and—more importantly—lacks constructs for using dynamic observations (such as comparing the value of a sorted list node to an element being inserted) to locally strengthen static knowledge (types) with new invariants (that some node's value is less than the value to insert). Flow of this information from dynamic checks into static information is critical to validating

```
type increasing (n : nat) (n2 : nat) (h : heap) (h2 : heap) : Prop = n <= n2;;
type pos (n : nat) (h : heap) : Prop = n >= 0;;
type monotonic_counter = ref{nat|pos}[increasing,increasing];;

let read_counter (c : monotonic_counter) : nat := !c;;
let inc_monotonic (c : monotonic_counter) : IO () = c := !c + 1;;
let mkCounter () : IO monotonic_counter = alloc 1;;

type ro (n : nat) (n2 : nat) (h : heap) (h2 : heap) : Prop = n == n2;;
type readonly_counter = ref{nat|pos}[increasing,ro];
type read_ro_counter (c : readonly_counter) : nat := !c;;

let test_counter () : IO nat = x <- mkCounter ();
                               inc_monotonic x;
                               val <- read_ro_counter (convert x);
                               return val;;
```

Fig. 1. A positive monotonically-increasing counter, adapted from Gordon et al. [2013].

invariant preservation (such as that the final insertion operation preserves list sortedness). This sort of static reflection of dynamic checks is critical for verifying concurrent programs, and such constructs would be useful for sequential programs as well. This section explains the original system, which we subsequently improve to verify invariants for concurrent programs.

The core concept is a RGREF: $\mathsf{ref}\{T \mid P\}[R, G]$. This is an extension of the standard ML-family reference type $\mathsf{ref}\ T$ to incorporate a rely (R), guarantee (G), and stable refinement (P as a mnemonic for "predicate"). Compatibility is checked whenever new aliases are created. The refinement is defined over the immediate referent and the heap reachable from it. The rely and guarantee are defined over the immediate referent in pre- and post-states of a heap access, as well as the heap reachable from it in both states; this is used to reason about the possible state transitions of the heap reachable from the immediate referent.[2]

A simple example is the monotonically increasing counter in Figure 1 [Pilkiewicz and Pottier 2011; Gordon et al. 2013]. A `monotonic_counter` is a reference to a number constrained (by both its rely and guarantee—`increasing`) to only ever increase. The `increasing` relation constrains the new value of the counter to be at least as large as the old value. The reference type is valid if the refinement pos is stable with respect to the rely $\mathsf{increasing}$. When type-checking the write in `inc_monotonic`, the type system verifies that the guarantee is preserved. This generates an obligation $\mathsf{increasing}\ !p\ (!p + 1)\ h\ h'$ for previous and new heaps h and h'. (Notice that predicates are defined over a T and heap, while relations are defined over two Ts and two heaps—pre- and post-heaps—though the counter does not use them.)

As in traditional rely-guarantee reasoning, the separate rely and guarantee relations enable asymmetric protocols, where two aliases may grant distinct permissions to modify memory. Figure 1 also defines a read-only counter `readonly_counter`, which permits aliases to increment but forbids updates through that alias. This permits defining a counter read operation that does not have permission to update the counter but can still read from it. Because this type is a weakening of the capabilities and assumptions of the `monotonic_counter` type, the `convert` coercion on the last line of the example may coerce the reference to the weaker type. Unlike a program logic, RGREFS cannot statically prove assertions in the traditional sense, although we will see later how they can often enforce sufficient conditions to ensure a dynamic assertion would succeed.

[2]This reachable-heap interpretation leads to subtleties with deep pointer structures, which we recall in Section 4.

Because references, unlike threads, are dynamically duplicated when aliases are created, a reference's own guarantee and rely interact directly. If a reference's guarantee does not imply its own rely, then duplicating the reference naïvely violates compatibility—since the original guarantee does not imply its rely, the guarantee of one new alias won't imply the rely of the original reference! Previous work [Gordon et al. 2013] gives the following example:

$$\mathsf{ref}\{\mathsf{nat} \mid \mathsf{any}\}[\mathsf{decreasing}, \mathsf{increasing}]$$

As a result, some references (and values that contain them) must be treated substructurally (linearly). This might initially appear inconvenient but in fact permits useful idioms: A freshly allocated location may have a very permissive guarantee, and a rely that requires immutability from (non-existent) aliases. This permits allocation to return a reference with a very precise refinement, exactly describing the contents of the new heap cell. Subsequent coercions from such linear reference types to types with more permissive rely relations (and, correspondingly, fewer precise predicates) can permit sharing, but the precise initial refinement is often useful in proof obligations when references are first stored into the heap. This was exploited in the original work [Gordon et al. 2013], and we exploit it here in both of our implementations.

2.2. Suitability of RGrefs for Concurrent Programming

This section explains the strengths and weaknesses of RGrefs and why we chose them as a basis for specifying and verifying concurrent programs. Rely-guarantee references are an appealing basis for concurrent programming, because they have features that allow natural integration with code unrelated to concurrency, and their specification style is a natural fit for specifying invariants and protocols for fine-grained concurrent data structures. Our work makes the system sound for concurrency and adds new primitives to refine verification goals based on dynamic observations. We also use a series of case studies to explore both the limits of the approach's expressiveness and effective integration with automation and real programming languages.

2.2.1. Strengths for Concurrent Programming. RGrefs offer a number of strengths for concurrent programming: They subsume and interact safely with well-typed but unverified code, they permit directly expressing protocols rather than embedding them in operations of a sealed module, and they are well-suited to specifying invariants of FCDs.

Subsuming Unverified Code. RGrefs subsume unverified code: an RGref whose rely, guarantee, and predicate impose no constraints is equivalent to a run-of-the-mill ML-style reference:

$$\mathsf{ref}_{\mathsf{ML}}\ T \overset{\mathrm{def}}{=} \mathsf{ref}\{T \mid \lambda x, h.\, \top\}[\lambda x, x', h, h'.\, \top, \lambda x, x', h, h'.\, \top].$$

We call these maximally permissive predicates and relations any and havoc, respectively.

Interacting with Unverified Code. Most verification systems cannot use unverified code without substantial conversion work. For example, most program logics can only assign the judgment $\vdash \{P\}\ C\ \{\mathsf{True}\}$ to unverified code, because there is no way to restrict how state is modified without also giving a precise postcondition, requiring strong verification to use results. It is possible to set P to the weakest precondition of C, making the command invocable: $\vdash \{\mathsf{WP}(C, \mathsf{True})\}\ C\ \{\mathsf{True}\}$. But composing C with verified code is challenging. Consider a program composed of verified code $\vdash \{P_1\}\ v_1\ \{Q_1\}$, C as above, and $\vdash \{P_2\}\ v_2\ \{Q_2\}$: $v_1; p; v_2$. It is quite possible that v_1's postcondition Q_1 implies $\mathsf{WP}(C)$, so having unverified code consume state and data produce by verified code is a non-issue. But C's known postcondition is True, which almost certainly does not imply precondition P_2 of v_2. In addition to this, computing C's weakest precondition requires reasoning about possible framing and access permissions, which accounts for a great deal of the work involved in soundly composing unverified code with fully verified components [Agten et al. 2015].

Meanwhile, unverified code already typechecks with our enriched reference types, using only the naïve translation above. Such code can be modified to refer to our richer types but simply "pass them along" without directly interacting with them. The `test_counter` routine in Figure 1 is an example of this: The routine itself is essentially unverified, and it simply manipulates the `monotonic_counter` as if it were an abstract data type. Any inappropriate attempts to write through a restricted reference simply fail to typecheck; passing a restricted reference to an unrestricted context—whose type assumes an unrestricted reference—produces a type error.

This more flexible interaction with unverified code is a consequence of giving types to memory locations rather than reasoning about precise details of the heap between every heap access (which is what separation logics are designed specifically to do). We deem this to be a productive tradeoff: We sacrifice some verification power but retain some benefits of full program logics while regaining some of the invariant-based flexibility of more traditional type systems.

To make the tradeoff more apparent, consider a function using an iterator that should increment a counter once for each natural number in an immutable list:

```
let rec forIO {T:⋆} (l:list T) (C:T->IO ()) : IO () =
  match l with
  | nil => return ()
  | cons t l' => C t; forIO l' C
  end;;
(* We can use the iterator to implement the spec *)
let doIncrements (c:monotonic_counter) (l:list nat) : IO nat =
  forIO l (fun n => incr c n); !c.
(* ... or increment too many times *)
let tooManyIncrements (c:monotonic_counter) (l:list nat) : IO nat =
  forIO l (fun n => incr c n; incr c n; incr c 1); !c.
(* This does not type-check in RGRefs:
let actuallyReset (c:monotonic_counter) (l:list nat) : IO nat =
  forIO l (fun n => [c]:=0); !c.
*)
```

A full specification for such an operation would require that in the final state, the counter has been incremented by the sum of the elements in the list (modulo interference from other threads, which would require use of subjective state [Ley-Wild and Nanevski 2013] to distinguish). Note that the code above contains three purported implementations of this specification: a satisfactory incrementor (`doIncrements`, which increments each counter in the list by `n`); one that is unsatisfactory (`tooManyIncrements` increments too much) but adheres to monotonicity; and a third (commented-out) implementation that flatly violates the expected two-state invariant of the counter (`actuallyReset`). The first two feed input to verified code and directly access the counter's representation (by reading its final value). Both type-check in the RGREF type system, because both respect the `increasing` predicate that enforced monotonicity; RGREFS cannot distinguish them because neither violates invariants. A full program logic could validate that `doIncrements` implements the specification correctly while rejecting `tooManyIncrements`, with some effort. This would require establishing an invariant relating the counter state to the sum of the un-visited portion of the list before and after each call to `forIO`'s higher-order parameter. The logic would require

```
let rec atom_inc (c:monotonic_counter) : IO () =
    x <- !c;
    done <- CAS(c, x, x+1);
    if done then return () else atom_inc c;;
```

Fig. 2. Atomic increment for the monotonic counter of Figure 1.

subjective state [Ley-Wild and Nanevski 2013] to express the specification precisely in a concurrent setting. The final, commented-out candidate would only be accepted by a regular type system (e.g., ML or Haskell), which would be unable to express or check the monotonicity requirements on the counter.

RGREFS never prevent any (ML-style) code from being written; a developer can always write her code with weaker refinements to make forward progress towards a running program. Thus a data structure with invariants proven using RGREFS can be safely used in the context of a larger program *without verifying the whole program*.

Protocols Independent of Abstraction. Another advantage of RGREFS for concurrency is that correctly enforcing state change protocols encoded in rely and guarantee relations does not *require* abstraction, even though state may be passed through unverified code. We exploit this in Section 6.1.

The monotonically increasing counter above was proposed by Pilkiewicz and Pottier [2011] as a verification challenge, because it requires proving temporal properties of how a piece of memory is used rather than characterizing the behavior of code on a given section of memory. Some solutions require creating modules that abstract over the type of the counter [Pilkiewicz and Pottier 2011; Jensen and Birkedal 2012] to ensure non-interference from other program components or are limited to a finite number of abstract states [Turon et al. 2013]. RGREFS permit exposing the counter's internal representation because the rely and guarantee ensure that all uses of that memory are consistent with a monotonically increasing counter. So, for example, a function that operates on read-only references to natural numbers can be passed an alias of our monotonic counter, with no mediation required. The solutions by Pilkiewicz and Pottier [2011] and Jensen and Birkedal [2012] additionally ensure functional correctness of increment, relying on a sealed module to constrain interference. A read-only alias could be mimicked by passing a closure rather than a reference at the cost of imposing a function call where a single memory access is sufficient.

The *requirement* to abstract the representation in these systems also hampers extensibility, as all operations must be verified within the sealed module. The RGREF counter in Figure 1 ensures that increment is the only permitted modification. But this is orthogonal to the abstraction required in the other solutions, since RGREFS can directly state and enforce limits on interference through a reference, so new operations can be added to data structures implemented using RGREFS by third parties. In the systems of Pilkiewicz and Pottier [2011] or Jensen and Birkedal [2012], monotonicity is ensured by verifying monotonicity for each of a *fixed*, *closed* set of operations and then sealing the module by abstracting the representation of the data outside the module. For example, consider adding an increment-by-n operation on monotonically increasing counters. Given implementations of the original increment-by-one operation in the various systems at hand, implementing this operation would require either modifying and *re-verifying* the module so the new operation has direct access to the representation or inefficiently calling the single increment operation n times. These are the only options because the protocol is enforced inside a module and then hidden rather than described in the module interface. With RGREFS, a new operation directly increments by n using a similar compare-and-swap loop to the implementation shown in Figure 2, without requiring any original code to be modified or re-proven. This is possible because RGREFS carry the restrictions on modification directly on the heap reference rather than implicitly coded into the pre- and post-conditions of a fixed set of operations. In fairness, the Pilkiewicz/Pottier and Jensen/Birkedal systems prove slightly stronger properties (e.g., that the counter was actually incremented, as opposed to proving it was not decremented), but this is a consequence of using a program logic (Jensen-Birkedal) or a linear capability accessible within a module via an anti-frame rule [Pottier 2008] (Pilkiewicz/Pottier), not a consequence of the abstraction. Other logics exist that do not *require* abstraction [Dinsdale-Young et al. 2010; Svendsen and Birkedal 2014; Sergey et al. 2015a] but instead specify protocols over a region of memory, and we discuss them in Section 8.

Suitability for FCD Specifications. Finally, the RGREF specification style is a natural fit for FCDs. Rely-guarantee reasoning itself has long established its utility for concurrent programs. RGREFS in particular allow encoding some forms of protocols similarly to recent work [Turon et al. 2013]. For example, O'Hearn et al. found it useful to describe many one- and two-state invariants of lock-free sets in terms of node-local properties and changes [O'Hearn et al. 2010] in a manner similar to RGREFS. Another key aspect of supporting (proofs of) FCD specifications is effective support for an idiom that is pervasive in verification of concurrent data structures: validating a heap write that publishes data (shares previously thread-local data via the heap) that will later be modified. Many algorithms store a previously thread-local node into a shared structure (e.g., inserting a list node) where validating the update requires knowing properties that are true *only* at the moment of update and not later. For example, validating list node insertion requires proving that the inserted node's successor pointer is the same as the predecessor's original successor pointer when the update occurs. This is true before the inserted node is shared, but because successor pointers are mutable, this can be falsified immediately after sharing the reference. RGREFS accommodate this idiom naturally, as we exploit throughout this article.

2.2.2. Disadvantages for Concurrent Programming. The chief limitation of RGREFS for concurrent programming is the original design's lack of a way to exploit dynamic observations (e.g., that a value was greater than 5) to induce new stable assertions. We add this ability to RGREFS in a way that works for sequential programs as well.

The other obvious disadvantage is that RGREFS are weaker than modern concurrent program logics such as FCSL [Nanevski et al. 2014; Sergey et al. 2015a] or Iris [Jung et al. 2015] in that RGREFS do not verify full functional correctness. This is a weakness but again by design: By targeting a weaker set of specifications and building on ideas known to be automatable and usable by developers (refinement types and reference capabilities), we aim to produce a system with intermediate expressiveness requiring intermediate user sophistication.

There are other limitations of RGREFS that we do not address here but discuss briefly in Section 8.

3. RGREFS FOR CONCURRENCY

In this section, we describe the key changes necessary to make RGREFS both sound and useful for concurrent programs (Section 3.1) and give a couple basic examples to develop intuition (Section 3.2) before proceeding with the formal development and extended case studies in subsequent sections.

3.1. Concurrency Changes to RGREFS

Granularity of Reasoning. In proving that $[x] := e$ (storing the result of e through reference x) obeys the guarantee for x's type, the original design [Gordon et al. 2013] considered e atomically, treating $!x$ (dereference of x) within e as a (locally) deterministic expression. This simplifies proofs: Verifying that $[x] :=!x + 1$ performs an increment requires proving the obligation $\forall h.\ \mathsf{inc}\ (!x)\ (!x + 1)\ h\ h[x \mapsto \ldots,]$ which is straightforward to prove in a dependent type theory treating dereference as any other expression. The left dereference in the obligation is produced because it is a write through x, allowing the proof to treat the update almost as a function from the old value in the heap ($!x$) to the new value ($!x + 1$). But this formulation implicitly assumes sequential semantics and is unsound in the presence of thread interleaving.

Recovering soundness is straightforward: We make heap reads fully monadic, explicitly sequencing every read and write. This makes the system *sound* but also *too weak* to prove many interesting properties; it removes the only mechanism the original work had to associate a read at a location to a subsequent write at that location.

Read/Write Atomicity. Sequential RGREFS also lack any notion of data size, which is required for reasoning about lock-free data structures, because some data types—anything larger than a machine word (e.g., pointer or machine-register-width integer)—cannot be read or written atomically without additional synchronization. We treat this with an Atomic predicate on types indicating those that are machine-atomic. Our COQ implementation includes support for fields, including compare-and-swap on a field. We use this in examples (Sections 3.2, 6) but omit fields from the formal treatment in Section 4.

Dynamic Refinement. Verifying concurrent programs often relies on reasoning about logical consequences of runtime checks on thread-shared data. For example, if code reads the value of a monotonic counter and observes that it is greater than 5, then a proof may correctly infer that the counter's value will remain greater than 5. The original RGREF design had no way to exploit this in proofs. This is critical in verifying lock-free data structure properties, as many data structures' correctness proofs rely on flow sensitive reasoning: an algorithm reads from a shared structure and takes different actions depending on the value observed. We add to RGREFS a *refiner* construct to induce new stable assertions based on values read out of shared data structures.

3.2. Basic Examples

We have used our COQ DSL and Liquid Haskell library to verify invariants for a number of lock-free data structures. This section presents simple examples to provide intuition for how RGREFS are used to specify and verify properties of FCDs. We defer more sophisticated examples to Section 6, after giving a formal account of concurrent RGREFS. The following examples are taken from our COQ DSL implementation but presented as a stylized dependent ML[3] for readability.

3.2.1. Atomic Counter. Figure 2 gives an atomic increment operation for a monotonic counter. In a loop, it reads the old counter value and uses the compare-and-swap (CAS) primitive to atomically store $x + 1$ only if it would overwrite x, looping again if the CAS fails. For this program, the type rule for CAS (Section 4.3) generates a proof obligation from the counter's guarantee to ensure that if the CAS does modify memory (i.e., if the value overwritten would be x), then the write will be permitted by the guarantee:

$$\forall (h : heap).\ h[\mathtt{c}] = \mathtt{x} \rightarrow \mathtt{increasing}\ h[\mathtt{c}]\ (\mathtt{x} + 1)\ h\ h[\mathtt{c} \mapsto \mathtt{x} + 1].$$

This obligation checks that if the value stored at c in the initial heap is x, then overwriting it with $x + 1$ is permitted by the guarantee (`increasing`). The CAS will only modify memory when c initially points to x (otherwise it fails and leaves the heap unmodified). So discharging this proof obligation ensures that if the CAS succeeds, it obeys the guarantee.

3.2.2. Treiber Stack. Figure 3 gives code for Treiber's lock-free stack [Treiber 1986] using concurrent RGREFS. The stack (`ts`) is a reference to an option of a reference to an

[3]Including inductive types for specifying the ways to construct evidence of a proposition.

```
type hpred (A : ⋆) = A -> heap -> Prop;;
type hrel (A : ⋆) = A -> A -> heap -> heap -> Prop;;
type Node : ⋆ =
    mkNode :: val:nat -> nxt:option (ref{Node|any}[local_imm,local_imm]) -> Node;;

type deltaTS : hrel (option (ref{Node|any}[local_imm,local_imm])) =
  | ts_nop : ∀ n h h', deltaTS n n h h'
  | ts_push : ∀ n hd hd' h h', h'[hd']=(mkNode n hd) -> deltaTS hd (Some hd') h h'
  | ts_pop : ∀ n hd hd' h h', h[hd]=(mkNode n hd') -> deltaTS (Some hd) hd' h h';;

type ts = ref{option (ref{Node|any}[local_imm,local_imm])|any}[deltaTS,deltaTS];;

let rec push_ts (s : ts) (n : nat) : IO () =
    tl <- !s;
    (* Γ = ..., tl : option(ref{Node | any}[local_imm, local_imm]) *)
    new_node <- Alloc (mkNode n tl);
    (* Γ = ..., new_node : ref{Node | λx, h. x = mkNode n tl}[local_imm, local_imm] *)
    success <- CAS(s,tl,Some (convert new_node));
    if success then return () else push_ts s n;;

let rec pop_ts (s : ts) : IO (option nat) =
    head <- !s;
    match head with
    | None -> return None
    | Some hd -> (* Γ = ..., hd : ref{Node | any}[local_imm, local_imm] *)
                 observe-field hd --> nxt as tl in (λ a h => (getF nxt a)=tl);
                 (* Γ = ..., tl : ..., hd : ref{Node | λx, h. getF nxt x = tl}[local_imm, local_imm] *)
                 observe-field hd --> val as n in (λ a h => (getF val a)=n);
                 (* Γ = ..., tl : ..., n : ..., hd : ref{Node | λx, h. x = mkNode n tl}[..., ...] *)
                 success <- CAS(s,Some hd,tl);
                 if success then return (Some n) else pop_ts s
    end;;
```

Fig. 3. A Treiber Stack [Treiber 1986] using RGREFS. The relation `local_imm` (not shown) constrains the immediate referent to be immutable; any is the always-true predicate.

immutable `Node`, updated according to relation `deltaTS`. `deltaTS` is used as the rely and guarantee for the Treiber stack's base reference (*delta* for change, *TS* for Treiber stack) and restricts writes through that reference to those that effect a single-node push or pop. This, with the relations for immutable interior nodes (`local_imm`), fully specifies one- and two-state invariants of the stack using reference types.

The push operation proceeds in a typical manner for this algorithm. It reads the current top of the stack (`!s`), allocates a new node (`Alloc`), and attempts to use compare-and-swap (`CAS`) to replace the old top of the stack with the newly allocated node. If the CAS fails, then the operation tries again.

In the case where the CAS succeeds, the mutation satisfies the `ts_push` case of `deltaTS`. Proving this relies on the strong (very specific) initial refinement when allocating the new head as an immutable node: $\lambda x, h.\, x = \mathsf{mkNode}\ n\ tl$. The `convert` operation is a type coercion on RGREFS that weakens the predicate (and/or rely and guarantee)—in this case weakening the predicate from $\lambda x, h.\, x = \mathsf{mkNode}\ n\ tl$ to any—while preserving identity. So the term stored in the heap will have the correct (less precise) type, but proofs can still exploit the stronger initial refinement. The CAS update satisfies the guarantee assuming the head of the stack is `tl` at the time of the write (an assumption the CAS rule introduces to characterize its conditional behavior; see Section 4.3). The strong refinement on `new_node` (that its next pointer is `tl`) proves that the new head's next pointer is the old head, validating the CAS. This sort of simultaneous sharing and weakening of a rely-guarantee reference appears in many lock free algorithms, including other case studies presented later.

The pop operation is slightly more involved. At a high level it is straightforward—read the top of the stack (`!s`), and if it is non-empty, read the current head's next-pointer (`observe-field`, explained momentarily) and attempt to CAS the current head to its successor. `observe-field` is a new construct we have added to refine reference predicates based on dynamic observation. Dynamically, it is a simple field read (here reading the `nxt` field). But the operation also takes a new refinement to apply to the reference accessed, stated in terms of the read result. Here the refinement is that the `nxt` projection of the stack top (`getF nxt a`) is equal to the read result `tl`. The type system verifies that the refinement is stable with respect to `local_imm` (next pointers are immutable) and rebinds the base reference (`hd`) with the new refinement.

The CAS in the pop operation satisfies the `ts_pop` case of the `deltaTS` relation. Proving this relies on the new refinement on `hd` introduced by `observe-field`. This refinement's information about the next pointer is sufficient to relate the fields of the old head to the new value stored by the CAS, proving the new top of the stack is the old second link.

The ABA Problem. In languages without garbage collection, the code from Figure 3 would permit the infamous ABA problem [Herlihy and Shavit 2008]. The ABA problem occurs when one thread reads the reference to the head node A at the start of a pop and finds its successor B, then other threads pop two or more nodes (removing A's original successor B), and then push a node with the same address as A (whose successor is no longer B). Then the first thread's CAS of the head from A to B (the *old* successor) succeeds, reinstalling previously removed node B. This occurs because the memory ascribed to A is reused after another thread succeeds in popping it, typically because A's memory is freed, but then handed out again by the memory allocator to a push operation. In a language with garbage collection (GC), this reuse can only occur if the code explicitly caches A for reuse (to reduce interaction with the GC and allocator).

Our specification prevents code from re-introducing the ABA problem, by prohibiting mutations required for the problematic reuse. As in other languages that assume GC, this reuse could occur only if the program explicitly reused the node. The `local_imm` relation on references to stack nodes prohibits the manual reuse except in the case that the node transitively points to the *same stack*. So even though our specification does not prove that `pop_ts` pops, we can enforce restrictions that prevent any code from causing the ABA problem.

From Treiber Stack to Producer-Consumer. The Treiber stack is a natural building block for more semantically meaningful primitives, such as a work queue for a producer-consumer relationship. In this case, we would like the producer to only be able to push and the consumer to only be able to pop. To accomplish this, we can coerce a reference to a Treiber stack as in Figure 3 into references with weaker guarantees. We can define relations `produce` and `consume` by omitting the undesirable case from `deltaTS` (`produce` omits the pop case, and `consume` omits the push case).

We can then define producer and consumer references as

```
type producer =
    ref{option (ref{Node|any}[local_imm,local_imm])|any}[deltaTS,produce];;
type consumer =
    ref{option (ref{Node|any}[local_imm,local_imm])|any}[deltaTS,consume];;
```

Each relation is reflexive and a subrelation of `deltaTS`, so these references may be freely duplicated and a `ts` as defined above may be coerced to either type. The body of `push_ts` (resp. `pop_ts`) type checks with the argument switched to a `producer` (respectively, `consumer`).

Because code with access to a `consumer` stack reference can only pop elements, these weakened references can be used to impose strong correctness properties beyond what the RGREF type system actually proves. Consider a loop that repeatedly pops elements of the stack until empty. If all aliases of the stack outside the loop's scope are `consumer` references, then if the loop pushes no elements the loop is guaranteed to terminate—no part of the program outside the loop may add elements to make the loop run longer. Our implementation gives an example of this.

4. CONCURRENT RGREFS, FORMALLY

This section offers a formal account of concurrency-safe RGREFS. As in prior work [Gordon et al. 2013], the language is structured as a basic imperative language, which can call into a pure sublanguage (mutation-free but able to read from the heap) with dependent types. The dynamic semantics (omitted for brevity) are standard call-by-value reduction with interleaved thread execution.

4.1. The Pure Fragment

Figure 4 gives the core (runtime) typing rules for the language. The pure fragment is an extension to the Calculus of Constructions (CC [Coquand and Huet 1988]) with additional basic types and eliminations (natural numbers, Booleans, and propositional equality of the form present in COQ's standard library), plus heap access primitives for stating specifications.[4] The heap primitives include the reference type described earlier, with its requisite well-formedness restrictions. For brevity, we also assume non-dependent pairs with standard recursors and various arithmetic and Boolean operations. We also assume knowledge of which types' representations can be accessed atomically by an implementation (i.e., which types are suitable size for CAS).

Each pure term that occurs in a program is nested inside an imperative command (discussed in the next section).

Most of the rules for the pure fragment are simply inherited from CC, so we discuss only the extensions in Figure 4. T-VAR is the standard variable read, with the additional condition that the type does not behave linearly (the next section explains these behaviors and the $\Gamma \vdash \tau \prec \tau \circledast \tau$ judgment). T-LOC is an extension of standard location typing for the tagged locations in our system, which explicitly represent the refinement and relations of the reference. T-CONV types the conversion operation mentioned earlier (convert in Section 3.2.2 and Figure 3), which coerces data structures containing references according to $\Gamma \vdash \tau \leadsto \tau$ (Figures 4 and 5). Operationally, convert recursively re-tags each reference in the value with new predicates and relations (sound because conversion only permits *weakening* operations, thus preserving global aliasing invariants). Formal semantics are given in Appendix B. Since the tags are computationally irrelevant, convert corresponds to an identity transformation in actual implementations. It amounts to relaxing any references contained in the term with weaker predicates and relations according to C-REF, the only interesting case of the conversion relation, which we discuss more momentarily. Note that conversion on function types is an identity transformation. Finally, T-REF checks the validity conditions on reference types that we have discussed informally thus far: well-sortedness of the type components and semantic conditions on the predicates and relations (stability, containment, precision) are discussed below.

[4]Recall that CC contains only two universes, Prop and Type, not the richer system including Set and a cumulative hierarchy of universes Type_i present in the CIC underlying the current COQ implementation [Bertot and Castéran 2004]. Our COQ DSL places data types in Set and uses Prop for predicates and relations.

$\boxed{\Sigma; \Gamma \vdash M : N \text{ extending } \vdash_{\mathrm{CC}}}$

$$\textsc{T-Var}\ \frac{\Gamma \vdash \tau \prec \tau \divideontimes \tau}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \qquad \textsc{T-Loc}\ \frac{\Sigma(\ell) = A}{\Sigma; \Gamma \vdash \ell_{A,P,R,G} : \mathsf{ref}\{A \mid P\}[R,G]}$$

$$\textsc{T-Conv}\ \frac{\Sigma; \Gamma \vdash M : N \qquad \Gamma \vdash N \rightsquigarrow N'}{\Sigma; \Gamma \vdash \mathsf{convert}\ M : N'}$$

$$\textsc{T-Ref}\ \frac{\begin{array}{c} \Sigma; \Gamma \vdash A : \mathsf{Prop} \\ \Sigma; \Gamma \vdash P : A \to \mathsf{heap} \to \mathsf{Prop} \qquad \Sigma; \Gamma \vdash \{R,G\} : A \to A \to \mathsf{heap} \to \mathsf{heap} \to \mathsf{Prop} \\ \mathsf{precise}_p(P) \qquad \mathsf{precise}_r(R) \qquad \mathsf{precise}_r(G) \qquad \mathsf{stable}\ P\ R \qquad \mathsf{contains}_A\ R \end{array}}{\Sigma; \Gamma \vdash \mathsf{ref}\{A \mid P\}[R,G] : \mathsf{Prop}}$$

$\boxed{\Gamma \vdash M \rightsquigarrow N}$

$$\textsc{C-Ref}\ \frac{G' \Rightarrow G \qquad R \Rightarrow R' \qquad P \Rightarrow P' \qquad \Gamma \vdash \mathsf{ref}\{A \mid P\}[R,G] : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{ref}\{A \mid P'\}[R',G'] : \mathsf{Prop}}{\Gamma \vdash \mathsf{ref}\{A \mid P\}[R,G] \rightsquigarrow \mathsf{ref}\{A \mid P'\}[R',G']}$$

$\boxed{\Gamma; \Delta \vdash C \dashv \Gamma'; \Delta}$

$$\textsc{T-Alloc}\ \frac{\begin{array}{c} \Gamma \vdash A : \mathsf{Prop} \qquad \Gamma \vdash M : B \qquad \Gamma \vdash B \rightsquigarrow A \\ \Gamma \vdash \mathsf{ref}\{A \mid P\}[R,G] : \mathsf{Prop} \qquad \forall h.\ P\ M\ h \end{array}}{\Gamma; \Delta \vdash x := \mathsf{alloc}_{A,P,R,G}\ M \dashv \mathsf{PlaceSplittable}(\Gamma, \Delta, x : \mathsf{ref}\{A \mid P\}[R,G])}$$

$$\textsc{T-Read}\ \frac{\Gamma \vdash \divideontimes A \qquad \Gamma \vdash M : \mathsf{ref}\{A \mid P\}[R,G] \qquad \mathsf{atomic}\ A \qquad \mathsf{reflexive}\ G}{\Gamma; \Delta \vdash x := [M] \dashv \Gamma, x : (\mathsf{fold}\ G\ A); \Delta}$$

$$\textsc{T-LinStore}\ \frac{\begin{array}{c} \Gamma(x) = \mathsf{ref}\{A \mid P\}[R,G] \\ y \neq x \qquad \Delta(y) = B \qquad \Gamma \vdash B \rightsquigarrow A \qquad (\forall v, b : B, h.\ P\ v\ h \Rightarrow G\ v\ (\mathsf{convert}\ b)\ h\ h[x \mapsto (\mathsf{convert}\ b)]) \\ (\forall v, b : B, h.\ P\ v\ h \Rightarrow P\ (\mathsf{convert}\ b)\ h[x \mapsto (\mathsf{convert}\ b)]) \end{array}}{\Gamma; \Delta \vdash [x] := y \dashv \Gamma; \Delta/y}$$

$$\textsc{T-Store}\ \frac{\begin{array}{c} \Gamma, \Delta \vdash x : \mathsf{ref}\{A \mid P\}[R,G] \\ \Gamma \vdash N : B \qquad \Gamma \vdash B \rightsquigarrow A \qquad \forall v, h.\ P\ v\ h \Rightarrow G\ v\ (\mathsf{convert}\ N)\ h\ h[x \mapsto (\mathsf{convert}\ N)] \\ \forall v, h.\ P\ v\ h \Rightarrow P\ (\mathsf{convert}\ N)\ h[x \mapsto (\mathsf{convert}\ N)] \end{array}}{\Gamma; \Delta \vdash [x] := N \dashv \Gamma; \Delta}$$

$$\textsc{T-Shift}\ \frac{\Gamma(y) = \tau}{\Gamma; \Delta \vdash x := y \dashv \Gamma; \Delta, x : \tau} \qquad \textsc{T-LinPair}\ \frac{\Delta(y) = \tau \qquad \Delta/y(z) = \sigma}{\Gamma; \Delta \vdash x := (y,z) \dashv \mathsf{PlaceSplittable}(\Gamma, \Delta/y/z, x : (\tau, \sigma))}$$

$$\textsc{T-CAS}\ \frac{\begin{array}{c} \Gamma \vdash y : \mathsf{ref}\{A \mid P\}[R,G] \qquad \Gamma \vdash N_0 : A \\ \Gamma \vdash N' : B \qquad \Gamma \vdash B \rightsquigarrow A \qquad (\forall h.\ h[y] = N_0 \Rightarrow G\ (h[y])\ (\mathsf{convert}\ N')\ h\ h[y \mapsto (\mathsf{convert}\ N')]) \\ \forall h.\ h[y] = N_0 \Rightarrow P\ N_0\ h \Rightarrow P\ (\mathsf{convert}\ N')\ h[y \mapsto (\mathsf{convert}\ N')] \end{array}}{\Gamma; \Delta \vdash x := \mathsf{CAS}(y, N_0, N') \dashv \Gamma, x : \mathbb{B}; \Delta}$$

$$\textsc{T-While}\ \frac{\Gamma \vdash M : \mathbb{B} \qquad \Gamma; \Delta \vdash C \dashv \Gamma; \Delta}{\Gamma; \Delta \vdash \mathsf{while}\ (M)\ \{C\} \dashv \Gamma; \Delta} \qquad \textsc{T-Cond}\ \frac{\Gamma \vdash B : \mathbb{B} \qquad \Gamma; \Delta \vdash C_1 \dashv \Gamma'; \Delta' \qquad \Gamma; \Delta \vdash C_2 \dashv \Gamma'; \Delta'}{\Gamma; \Delta \vdash \mathsf{if}\ (B)\ \{C_1\}\ \mathsf{else}\ \{C_2\} \dashv \Gamma'; \Delta'}$$

$$\textsc{T-Seq}\ \frac{\Gamma; \Delta \vdash C_1 \dashv \Gamma'; \Delta' \qquad \Gamma'; \Delta' \vdash C_2 \dashv \Gamma''; \Delta''}{\Gamma; \Delta \vdash C_1; C_2 \dashv \Gamma''; \Delta''} \qquad \textsc{T-Par}\ \frac{\Gamma_1; \Delta_1 \vdash C_1 \dashv \Gamma_1'; \Delta_1' \qquad \Gamma_2; \Delta_2 \vdash C_2 \dashv \Gamma_2'; \Delta_2'}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash C_1 \parallel C_2 \dashv \Gamma_1', \Gamma_2'; \Delta_1', \Delta_2'}$$

$$\textsc{T-Refine}\mathbb{N}\ \frac{\begin{array}{c} \Gamma \vdash x : \mathsf{ref}\{\mathbb{N} \mid P\}[R,G] \qquad \Gamma \vdash \divideontimes\mathsf{ref}\{\mathbb{N} \mid P\}[R,G] \\ (\forall h, v.\ h[x] = v \to v = 0 \to P_0\ v\ h) \qquad \mathsf{stable}\ P_0\ R \qquad (\forall h, v, n.\ h[x] = v \to v = S\ n \to P_S\ v\ h) \\ \Gamma, n : \mathbb{N} \vdash \mathsf{stable}\ P_S\ R \qquad \Gamma, x' : \mathsf{ref}\{\mathbb{N} \mid P \cap P_0\}[R,G], pf : x \approx x'; \Delta \vdash C_1 \dashv \Gamma'; \Delta' \\ \Gamma, n : \mathbb{N}, x' : \mathsf{ref}\{\mathbb{N} \mid P \cap P_S\}[R,G], pf : x \approx x'; \Delta \vdash C_2 \dashv \Gamma'; \Delta' \end{array}}{\Gamma; \Delta \vdash !\mathcal{R}_{\mathbb{N}}(x, Z \Rightarrow C_1, S\ n \Rightarrow C_2) \dashv \Gamma'; \Delta'}$$

$$\textsc{T-RefineRef}\ \frac{\begin{array}{c} \Gamma \vdash x : \mathsf{ref}\{\mathsf{ref}\{A \mid P'\}[R',G'] \mid P\}[R,G] \\ \Gamma \vdash \divideontimes\mathsf{ref}\{\mathsf{ref}\{A \mid P'\}[R',G'] \mid P\}[R,G] \qquad \Gamma \vdash \divideontimes\mathsf{ref}\{A \mid P'\}[R',G'] \\ (\forall h, v, r.\ h[x] = v \to v = r \to P_0\ v\ h) \qquad \Gamma, r : \mathsf{ref}\{A \mid P'\}[R',G'] \vdash \mathsf{stable}\ P_0\ R \\ \Gamma, r : \mathsf{ref}\{A \mid P'\}[R',G'], x' : \mathsf{ref}\{\mathsf{ref}\{A \mid P'\}[R',G'] \mid P \cap P_0\}[R,G], pf : x \approx x'; \Delta \vdash C \dashv \Gamma'; \Delta' \end{array}}{\Gamma; \Delta \vdash !\mathcal{R}_{\mathsf{ref}}(x, r \Rightarrow C) \dashv \Gamma'; \Delta'}$$

Fig. 4. Core typing rules for concurrency-safe RGREFS. Auxiliary definitions are given in Figure 5.

$$\mathsf{contains}_A\ R \overset{\text{def}}{=} \forall l : \mathsf{ref}\{A \mid Pa\}[Ra, Ga].\, \forall y : \mathsf{ref}\{B \mid Pb\}[Rb, Gb] \in h[l].\, \forall h, h'.\, Rb\ h[y]\ h'[y]\ h\ h' \Rightarrow R\ h[l]\ h'[l]\ h\ h'$$

$$\mathsf{precise}_p(P) \overset{\text{def}}{=} \forall x, h, h'.\, (\forall l.\, \mathsf{ReachableFromIn}\ l\ x\ h \Rightarrow h[l] = h'[l]) \Rightarrow P\ x\ h \Rightarrow P\ x\ h'$$

$$\mathsf{precise}_r(R) \overset{\text{def}}{=} \begin{array}{l} \forall x, x2, h, h', h2, h2'.\ \ (\forall l.\, \mathsf{ReachableFromIn}\ l\ x\ h \Rightarrow h[l] = h'[l]) \Rightarrow \\ \qquad (\forall l.\, \mathsf{ReachableFromIn}\ l\ x2\ h2 \Rightarrow h2[l] = h2'[l]) \Rightarrow R\ x\ x2\ h\ h2 \Rightarrow R\ x\ x2\ h'\ h2' \end{array}$$

$$\mathsf{stable}\ P\ R \overset{\text{def}}{=} \forall x, x', h, h'.\, P\ x\ h \wedge R\ x\ x'\ h\ h' \Rightarrow P\ x'\ h'$$

$$\begin{array}{rl} \mathsf{fold}\ G\ (A * B) \overset{\text{def}}{=} (\mathsf{fold}\ G.1\ A * \mathsf{fold}\ G.2\ A) & R.1 \overset{\text{def}}{=} \lambda x, x' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall y : \sigma'.\, R\ (x, y)\ (x', y)\ h\ h' \\ \mathsf{fold}\ G\ (\mathsf{ref}\{A \mid P\}[R, G_0]) \overset{\text{def}}{=} R!G_0 & R.2 \overset{\text{def}}{=} \lambda y, y' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall x : \sigma'.\, R\ (x, y)\ (x, y')\ h\ h' \\ \mathsf{fold}\ G\ A \overset{\text{def}}{=} A \text{ otherwise} & R!G \overset{\text{def}}{=} \lambda a, a', h, h'.\, G\ a\ a'\ h\ h' \wedge (\forall l, h[l] = a \rightarrow h'[l] = a' \rightarrow R\ l\ l\ h\ h) \end{array}$$

$$\mathsf{atomic}\ A \overset{\text{def}}{=} A \in \{\top, \bot, \mathbb{N}, \mathbb{B}, \mathsf{unit}, (\Pi x : S.\, T[x]), \mathsf{ref}\{T \mid P\}[R, G]\}$$

$$\mathsf{PlaceSplittable}(\Gamma, \Delta, x : \mathsf{ref}\{A \mid P\}[R, G]) \overset{\text{def}}{=} \begin{cases} \Gamma, x : \mathsf{ref}\{A \mid P\}[R, G]; \Delta & \text{if } \Gamma \vdash \divideontimes\mathsf{ref}\{A \mid P\}[R, G] \\ \Gamma; \Delta, x : \mathsf{ref}\{A \mid P\}[R, G] & \text{otherwise} \end{cases}$$

$$\boxed{\Gamma \vdash \tau \prec \tau \divideontimes \tau} \qquad \frac{\tau \in \{\mathbb{N}, \mathbb{B}, \mathsf{unit}, \mathsf{Prop}, \mathsf{Type}, \mathsf{heap}, \_ = \_, \Pi x : \tau \rightarrow \tau'\}}{\Gamma \vdash \tau \prec \tau \divideontimes \tau} \qquad \frac{\Gamma \vdash \tau \prec \tau_a \divideontimes \tau_b \qquad \Gamma \vdash \sigma \prec \sigma_a \divideontimes \sigma_b}{\Gamma \vdash (\tau, \sigma) \prec (\tau_a, \sigma_a) \divideontimes (\tau_b, \sigma_b)}$$

$$\textsc{Ref-}\divideontimes\ \frac{\begin{array}{c} \Gamma \vdash \mathsf{ref}\{b \mid \phi'\}[R', G'] \qquad \Gamma \vdash \mathsf{ref}\{b \mid \phi''\}[R'', G''] \\ \emptyset \subset [\![G']\!] \subseteq [\![R'']\!] \qquad \emptyset \subset [\![G'']\!] \subseteq [\![R']\!] \qquad [\![G']\!] \cup [\![G'']\!] \subseteq [\![G]\!] \qquad [\![R]\!] \subseteq [\![R']\!] \qquad [\![R]\!] \subseteq [\![R'']\!] \end{array}}{\Gamma \vdash \mathsf{ref}\{b \mid \phi\}[R, G] \prec \mathsf{ref}\{b \mid \phi'\}[R', G'] \divideontimes \mathsf{ref}\{b \mid \phi''\}[R'', G'']}$$

$$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau \text{ cont.}} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \rightsquigarrow \tau} \qquad \frac{\Gamma \vdash \tau \rightsquigarrow \tau' \qquad \Gamma \vdash \tau' \rightsquigarrow \tau''}{\Gamma \vdash \tau \rightsquigarrow \tau''} \qquad \frac{\Gamma \vdash \tau' \rightsquigarrow \tau}{\Gamma \vdash (\tau \rightarrow \sigma) \rightsquigarrow (\tau' \rightarrow \sigma)}$$

$$\frac{\tau =_{\beta v} \tau'}{\Gamma \vdash \tau \rightsquigarrow \tau'} \qquad \frac{\Gamma \vdash \tau \rightsquigarrow \tau' \qquad \Gamma \vdash \sigma \rightsquigarrow \sigma'}{\Gamma \vdash (\tau, \sigma) \rightsquigarrow (\tau', \sigma')}$$

Fig. 5. Auxiliary definitions for type rules in Figure 4.

C-Ref checks validity of weakening a reference type—a form of subtyping. Note that the predicate and rely are treated covariantly (relation R' must contain new rely relation R), while the guarantee is treated contravariantly (relation G must contain new guarantee relation G'). This corresponds to standard separations of read and write effects on references, going back as far as Reynolds' treatment in Forsythe [Reynolds 1988], separating variable reads as expressions with covariant subtyping from variable *acceptors* (writers without read capabilities) with contravariant subtyping. Today this is typically exploited only in the form of safe covariant subtyping in systems with deep reference immutability [Zibin et al. 2007; Gordon et al. 2012]. In the case of C-Ref, the coercion produces a reference that may perform no more modifications than the original, assumes at least as much interference from aliases as the original, and assumes a possibly weaker predicate on the values stored (stable with respect to the new rely R'). This preserves compatibility—any reference compatible with the original will be compatible with the new reference—but allows references that might have strong refinements to be weakened to an appropriate type when shared with other threads.

As mentioned in Section 2.1, because a reference's predicate, rely, and guarantee are interpreted as restrictions over the heap reachable from the immediate referent, additional checks are required for nested references (references to heap cells containing references). These are unchanged from the original formulation of RGrefs [Gordon et al. 2013]. First, the rely is required to admit any interference covered by the rely of any possibly-reachable reference. This ensures that if a pointer exists into the interior of a linked data structure, reasoning about "root" pointers suffices.[5] Second, because a guarantee may be more restrictive than the guarantees of reachable references (e.g., a read-only reference to a linked list whose interior pointers permit updates), the result type of a dereference is transformed to permit only actions permitted by both the

[5]Note that while this is phrased in terms of tree-shaped structures, it is still applicable to richer structures; the rely of any reference to a graph node would account for any possible interference on reachable nodes.

original reference and the reference stored in the heap. This is called *folding*, shown by the fold construct in Figure 4. This ensures that any writes through a reference read out of the heap satisfy the guarantee of the reference they were read through—a program may not acquire more permissions by reading a "stronger" reference out of the heap because they are weakened on the way out. Third, the refinement and relations are required to be *precise*—sensitive only to the reference's immediate referent and the heap reachable from that. This prevents nonsensical types, such as those asserting that the whole heap is immutable (making all predicates, even incorrect ones, stable).

4.2. The Imperative Fragment

The primary context is an imperative one, judged flow-sensitively via $\Gamma; \Delta \vdash C \dashv \Gamma'; \Delta'$. Γ and Δ are standard and linear contexts, respectively.

Linear and Reflexively Splittable Values. As prior work explains [Gordon et al. 2013], and we reviewed briefly in Section 2.1, an RGREF's guarantee must imply its rely to allow free duplication without violating the compatibility invariant. Other references must behave linearly (e.g., $\mathsf{ref}\{\mathbb{N} \mid \mathsf{any}\}[\mathsf{dec}, \mathsf{inc}]$ cannot be duplicated safely). Thus the judgment $\Gamma \vdash \tau \prec \tau' \circledast \tau''$ judges whether a type τ can be split into two possibly weaker values of type τ' and τ'' while preserving compatibility. The judgment checks that the types τ' and τ'' are mutually compatible and preserves assumptions about interference and limitations on modification from the original type τ. This check—particularly the rule REF-$\circledast$ in Figure 5—is akin to the "shuffling" of rely and guarantee in the classic rely-guarantee parallel composition rule—creating a new alias is the RGREF equivalent of forking new threads. In the frequent case where $\tau = \tau' = \tau''$, we say values of type τ are *reflexively splittable* and abbreviate the splitting judgment as $\Gamma \vdash \circledast\tau$. Every value (in particular, references) is either treated substructurally (linearly) or is reflexively splittable and therefore may be arbitrarily duplicated safely. The pure fragment operates only on reflexively splittable data, and thus Γ contains only reflexively splittable data.

Extensions for Refiners. We extend the formal model compared to the original RGREF model, making the core calculus more expressive, and allowing us to type the *refiner* primitive described below (the formal analogue of `observe-field` in Section 3.2.2). Γ may be a dependent context, for example, $\vdash x : \mathbb{N}, y : \mathsf{ref}\{nat \mid \lambda v, h.v = x\}[\ldots, \ldots]$. Δ is a non-dependent context that may contain substructural values (with non-reflexively splitting types), whose types are well-formed under Γ. Γ only grows flow-sensitively ($\Gamma \subseteq \Gamma'$ in every judgment), whereas Δ may drop variables. This is sufficient for allocating references with asymmetric rely and guarantee that support a very strong refinement, then weakening to a reflexively splittable type on sharing with T-LINSTORE (discussed near the end of this section).

Allocation. T-ALLOC allocates a new reference with the specified predicate, rely, and guarantee and may produce a linear reference (whose guarantee does not imply its rely and therefore cannot be freely duplicated). This is important because it can return a reference typed to assume no interference from aliases in its rely, which makes a predicate stating the exact value of the new heap cell stable. Later, when this reference is shared with other threads, the reference can be weakened using convert but obligations can be proven assuming the original very precise refinement (see the discussion of the heap update rules below).

PlaceSplittable adds the binding to Γ or Δ as appropriate. T-READ reads the value of a reference. fold (Figure 5) weakens the result type to ensure that no embedded references grant greater permissions to a structure than the base reference. For example, a reference whose guarantee restricts a heap structure to be immutable cannot be used to read a reference with mutation permissions out of a data structure, in the style of reference immutability [Gordon et al. 2012].

Heap Updates. T-STORE and T-LINSTORE write into the heap, the latter sharing a possibly linear value from Δ. Both rules permit simultaneous sharing and weakening—publication—as used in Section 3.2.2 to verify Treiber stack updates. T-CAS is similar, although it additionally introduces new information about the referent at the time of update, modeling the conditional success of the compare-and-swap operation.

Without the simultaneous weakening and sharing behavior in these rules (and the possibility for T-ALLOC to return a linear reference with a very precise initial refinement), it would be impossible to verify some operations. Notably, verifying the Treiber stack's push operation requires proving the updated top of the stack points directly to the old top of the stack—that only a single node was pushed. Similar proof obligations arise when inserting into the linked list representing the lock-free set in Section 6.3 and are discharged similarly.

Structural Rules. T-SHIFT and T-LINPAIR deal with moving reflexive values into Δ and constructing a pair of (possibly) linear values. T-WHILE, T-COND, T-SEQ, and T-PAR are mostly standard structural rules with proper treatment of the linear context.

Refiners. A new feature we add to the imperative fragment of RGREFS is shown by the $!\mathcal{R}_{\mathbb{N}}$ and $!\mathcal{R}_{\mathsf{ref}}$ *refiners*. Based on the constructors of a type, the construct introduces an alias refined additionally with a new stable predicate implied by the observed constructor. This is the core language equivalent of the `observe-field` from Figure 3's pop operation, where observing the (immutable-by-rely-guarantee) next pointer of the stack's head refines knowledge about the head pointer. We assume equivalent refiners for each atomic type (reference, Boolean, natural). In the case of natural numbers, behavior is refined based on whether the number is zero or some successor, while in the case of references—which have no pure eliminator—we simply bind reference identity. In each case, equality between the value stored in the heap and a relevant constructor (or for references, simply a particular reference) must imply a new stable predicate (over the value and heap, suitable for use as the predicate component of an RGREF whether the heap is used), which is assumed to hold on the appropriate branch of the refiner.

4.3. Treating Interleaving in Proofs

Because read operations are monadic, pure expressions cannot observe interference from other threads, so no special reasoning principles are needed to address interleaving.

When proving that a heap write satisfies the guarantee of the base reference, relationships to the heap may be derived only from the conditional behavior of the CAS (T-CAS in Figure 4): Proofs that the write respects the guarantee may assume the value overwritten ($h[y]$) is equal to the expected old value (N_0). This is clearly sufficient for reasoning about very local properties (e.g., the counter increment).

This may seem too weak for proofs about deeper portions of the heap, but values stored into the heap may initially carry a stronger refinement than their storage location ($\Gamma \vdash B \rightsquigarrow A$). This enables reasoning about sharing (publishing) strongly refined data, using `convert` and an axiom that weakening a reference type preserves pointer equality (informally, $\forall h, r. h[r] = h[\mathsf{convert}\ r]$). This axiom, in conjunction with the axiom that a reference's refinement is always true (a reflection of the type system preserving invariants—that a reference's predicate holds in the current heap), allows guarantee obligations to be proven by storing a precisely refined reference into the heap in Figure 3. This pattern of simultaneously publishing and weakening a rely-guarantee reference shows up repeatedly in lock-free algorithms. For example, see the CAS in the stack push operation in Figure 3 (Section 3.2.2): The initial refinement for the node being pushed onto the head of the stack states exactly the value of the next pointer, and this is weakened to a less precise type when shared via the CAS, but the knowledge of that exact next pointer is used to prove the push case of the guarantee relation. Enqueuing at the end of a Michael-Scott Queue (proving that the queue remains null-terminated while appending the previously-thread-local new node) is another example. In our lock-free union-find implementation (Section 6.1), we exploit this to carry specific information about a node's rank and parent into guarantee proofs. In these latter two cases, not only are the refinements but also the rely and guarantee are weakened.

4.4. Soundness Sketch

Here we sketch soundness for the core language above. Full details follow in Appendix A. Soundness for the type system follows from an embedding into the Views Framework of Dinsdale-Young et al. [2013]—an *abstract* concurrent program logic. When the framework's parameters are instantiated appropriately for choice of assertion language, state space, and primitive operations, soundness for the base system follows from a few lemmas about the parameters and an embedding theorem. The proof is essentially decomposed into a soundness proof for the pure fragment and a soundness proof for the impure fragment. The inner fragment is a fragment of Calculus of Inductive Constructions (CIC), Coq's core calculus: CC with a few standard data types, plus primitives for constructing propositions about heap contents and computing with references (which lack an eliminator in the pure fragment). Because it is a fragment of CIC, it is strongly normalizing.

The impure fragment is proven sound by embedding into an instantiation of the Views Framework. We instantiate the state space to an explicitly typed stack and heap storing terms of the pure fragment. As assertions, we choose a particular family of predicates on the syntactic typing of the pure terms stored in the stack and heap. We instantiate the primitives to the non-structural rules from our system (dereference, write, etc.) and give valid Hoare triples for those primitives. Finally, we give an embedding function from impure typing derivations to triples in the instantiated Views logic and prove that the embedding of any valid typing derivation is a valid derivation in Views. The embedding includes a desugaring $\downarrow - \downarrow$ of source statements to the Views core language, which has only non-deterministic conditionals and loops.

THEOREM 4.1 (RGREF. SOUNDNESS). *For all* Γ, Δ, C, Γ', *and* Δ'

$$\Gamma; \Delta \vdash C \dashv \Gamma'; \Delta' \Longrightarrow \{[\![\Gamma, \Delta]\!]\} \vdash \downarrow C \downarrow \dashv \{[\![\Gamma', \Delta']\!]\}.$$

5. IMPLEMENTATION, TWO WAYS

We have implemented RGREFS twice: once as an axiomatic Coq embedding and once as a Liquid Haskell library.

5.1. Axiomatic Coq DSL Implementation

We implemented concurrent RGREFS as a modification of the original RGREF implementation [Gordon et al. 2013], itself a shallow axiomatic DSL embedding in Coq. This means we have given axioms in Coq for various RGREF primitives, whose types ensure they are used in a manner consistent with the type rules in Figure 4. This axiomatization's correctness relies on our hand-proven metatheory, while our data structure verifications are certified by Coq to be correct with respect to our axiomatization. This is similar to the YNOT axiomatic shallow embedding of Hoare Type Theory [Nanevski et al. 2008; Chlipala et al. 2009]. Proof obligations arising from RGREF type checking are by default presented to the user for interactive proof discharge, although, as discussed in prior work [Gordon et al. 2013], we can use Coq's `Program` extension and proof search tactics to automatically discharge some obligations. Type-incorrect programs that fail to type check due to failure of conditions on the rely, guarantee, and so on, present unsolvable proof obligations to the user, as with any Coq proof. In our Coq formulation, we move the contents of the RGref reference type, and the type itself, into the universe Set rather than Prop, so the type of the reference constructor becomes:

```
forall (A:Set), hpred A -> hrel A -> hrel A -> Set
```

This better supports interaction with existing Coq data types in Set and avoids possible abuse of the proof irrelevance axiom.

The implementation also replaces Figure 5's total formulation of folding by a partial formulation. Rather than defining a total function that computes a result type restricting use of embedded references, we specify conditions under which it is safe for reads through a reference to a τ with guarantee G to produce a value in type σ. In our core calculus, this change is not useful (Figure 5 computes σ from τ and G), but in Coq we use inductive data types for convenience. There is no general way to compute a new inductive data type that incorporates the restrictions of a guarantee relation. For example, a reference to a queue `Node` whose guarantee only permitted addition of odd numbers to the queue would require a fold to propagate this information into the tail reference when reading the node value. This result would no longer be an element of the `Node` type. So, instead, we place trusted safety conditions on when a result is sound as type classes and provide a collection of general instances. Thus the type class instances validate specific result types for reads through references of certain rely/guarantee pairs. For instance, in the Treiber stack, the interior references use the relation `local_imm` for both rely and guarantee. Because `local_imm` only constrains the immediate referent and ignores the heap parameters to the relation, no transformation is needed (folding is a no-op) when reading through a reference with a `local_imm` guarantee. This is provided as a generic type class.

We worked around two limitations of Coq 8.4 using axioms. First, in one case, we axiomatized (propositional) eta equivalence for nodes of one structure (which would be unnecessary in the most recent Coq release). Second, we defined our Michael-Scott queue [Michael and Scott 1996] by axiomatizing an inductive-inductive [Forsberg and Setzer 2010] simultaneous definition of queue nodes and predicates on queue nodes. Inductive-inductive types are more general than mutual inductive types; they permit mutual definition of a type A alongside a type family B indexed by elements of A. For our purposes, A is the node of a Michael-Scott queue, and B is a (heap) relation used as the rely and guarantee on the next-node pointer in the queue (given as an inductively-defined predicate).

```
Inductive Node : Set :=
| mkNode : nat -> option (ref{Node|validNode}[deltaNode,deltaNode]) -> Node
with validNode : Node -> heap -> Prop :=
| ... (* ensure acyclicity *)
with deltaNode : Node -> Node -> heap -> heap -> Prop :=
| ... (* constrain queue to append-only *) .
```

This is the most natural way to specify the queue. Note that these definitions are not only mutual (the predicate and relation are used in the tail pointer's type), but also `Node` appears as an index in the types of `validNode` and `deltaNode`. Sequential RGrefs [Gordon et al. 2013] adapted an impredicative encoding of induction-recursion [Capretta 2004] to give a similar definition, by using Coq's support for impredicative set. Our embedding of concurrent RGrefs instead axiomatizes the above (idealized) inductive-inductive definition of the queue nodes and the rely/guarantee used for the tail pointer. Work on supporting induction-induction (and induction-recursion [Dybjer 2000]) in Coq is ongoing but remains experimental.[6]

5.2. Liquid Haskell Implementation

We have also implemented a restricted form of RGrefs as a library atop Liquid Haskell [Vazou et al. 2013, 2014b], an SMT-based refinement type system for Haskell. Our encoding is concise, complements Liquid Haskell's existing strengths, highlighting that RGrefs are amenable to automation, and integrates naturally with related verification techniques (namely, dependent refinement types).

Liquid Haskell is the latest in the line of work on Liquid Types [Rondon et al. 2008]. Liquid types use abstract interpretation to infer a class of dependent refinement types (for C, ML, or Haskell) that is efficiently decidable by an SMT solver.

This section gives a brief introduction to Liquid Haskell and then briefly describes our encoding of a restricted form of RGrefs.

5.2.1. Refinement Types in Liquid Haskell. Liquid Types [Rondon et al. 2008; Kawaguchi et al. 2009, 2010; Rondon et al. 2010; Jhala et al. 2011; Kawaguchi et al. 2012; Rondon et al. 2012; Rondon 2012; Vazou et al. 2013, 2014a, 2014b] is a design for dependent refinement types that support effective inference and automation. Boolean-valued predicates are mined from the Boolean test expressions in a program (plus a fixed set of basic predicates) to gather a set of candidate refinements. Abstract interpretation is then used to infer which predicates hold at each program location, and an SMT solver is invoked to resolve implications between refinements. The result is a family of type theories over OCaml, C, and Haskell that are useful for verifying safety properties with modest annotation burden and user expertise.

The latest incarnation of these ideas, Liquid Haskell [Vazou et al. 2013, 2014a, 2014b], implements Liquid Types for Haskell, extending the base theory to tackle issues with type classes, generating verification conditions for lazy evaluation [Vazou et al. 2014b], and polymorphism over refinements [Vazou et al. 2013], which were absent from previous Liquid Types systems. In short, Liquid Haskell permits writing refinement types over Haskell values, for example,

$$\{x : \texttt{Int} \mid x > 0\},$$

or taking advantage of binding argument values in subsequent refinements; one possible type for addition would be

$$x : \texttt{Int} \to y : \texttt{Int} \to \{v : \texttt{Int} \mid v = x + y\},$$

where the + in the result type corresponds to addition in the SMT solver's logic.

For our purposes, the most useful features are refinement polymorphism, and the ability to extend the SMT solver's logic with additional predicates.

Abstract Refinements. Abstract refinements permit generalizing refinements from the form

$$x : \{v : \tau \mid \phi[v]\} \to \ldots$$

to the form

$$\forall\langle p :: \tau \to \ldots \to \mathsf{Prop}, \ldots\rangle . x : \{v : \tau\langle p\rangle \mid \phi[v, p]\} \to \ldots .$$

So the dependent refinement types are extended to allow prenex quantification over n-ary predicates. In addition, data type definitions may be parameterized by such predicates and uses of such data types support explicit (full) application to parameters.

[6]https://github.com/mattam82/coq/tree/IR.

As a simple concrete example, consider the specification of `min` on integers, due to Vazou et al. [2013]:

```
{-@ min :: forall <p :: Int -> Prop>. Int<p> -> Int<p> -> Int<p> @-}
min :: Int -> Int -> Int
min x y = if x < y then x else y
```

The parametric refinement given above reflects the fact that whatever property holds of both inputs to `min` will also be true (trivially) of the outputs.

More recently, Liquid Haskell has gained *bounded refinements* [Vazou et al. 2015], which allow a bound to be stated on abstract refinements. The implicit bounds are roughly equivalent to a subtyping bound: Under the assumptions of the initial arguments, the last argument is a subtype of the "result type." As an example, given a unary abstract refinement p and binary abstract refinement r (acting as a predicate and rely), to ensure that predicate p is stable with respect to r, we can impose the refinement bound:

$$\{x : a\langle p\rangle \vdash a\langle r\ x\rangle <: a\langle p\rangle\}.$$

This requires that for any $x : a$ satisfying p, any a related to x by r ($a\langle r\ x\rangle$) also satisfies p (is a subtype of $a\langle p\rangle$). This encodes the stable predicate check from Figure 4. Other uses of bounds are similar. One limitation of these abstract refinement bounds is that they must come before concrete function parameters and therefore may not mention concrete parameters or use them to partially apply other abstract refinements. We required this for one primitive in our encoding, explained in Section 5.2.2.[7]

Measures, Axioms, and SMT. When verifying a program, it is generally necessary to give new logical definitions to write (and prove) rich specifications. In Liquid Haskell, these definitions arise in two ways. First, *measures*[8] may be defined that behave as partial computable predicates over some data.[9]

Second, axioms may expand the meaning of uninterpreted functions with select computation rules. Figure 6 demonstrates the use of axioms by simplifying an example due to Liquid Haskell's authors [Vazou et al. 2013] using an axiomatization of an SMT solution to producing the ith Fibonacci number to verify a memoized Fibonacci calculation. We simplify the example to verifying that a Haskell implementation of the Fibonacci function is equivalent to the SMT version.

An axiom is simply stated as a function producing a Boolean asserting the truth of some refinement. In this case, `axiom_fib` asserts definition of the ith Fibonacci number. Its body is simply `undefined`, which in Haskell is a non-value (which causes an exception if evaluated). Since Liquid Haskell only proves refinements of *values* and `undefined` does not evaluate to a value, it has the refinement `false`, allowing the SMT solver to prove any obligations in the body of the axiom (in this case none) after its use.

To use an axiom, the proposition the axiom asserts must be injected into a refinement (conjoined with an existing refinement) where the axiom is necessary to prove an entailment. In this case, the return value of the `fibhs`[10] routine requires the axiom. Without knowledge of the meaning of the measure `fib`, Liquid Haskell cannot prove

[7]Our initial implementation predated bounded refinements and used exclusively these explicit proof terms.
[8]The name originates from the original use for proving termination (e.g., *measuring* the size of a data structure as a termination metric).
[9]Presently definition via pattern matching may only be given for measures of one parameter (which admits a clever "compilation" strategy of embedding the result of a measure application as a refinement on a constructor result type [Vazou et al. 2014b]).
[10]The `hs` suffix matches the `.hs` file suffix for Haskell programs to distinguish it from the SMT definition of the ith Fibonacci number.

```
module Fib where
import Language.Haskell.Liquid.Prelude

{-@ measure fib :: Int -> Int @-}

{-@ assume axiom_fib :: i:Int ->
    {v: Bool | (Prop(v) <=> (fib(i) = ((i <= 1) ? 1 : ((fib(i-1)) + (fib(i-2)))))) } @-}
axiom_fib :: Int -> Bool
axiom_fib i = undefined

{-@ fibhs :: i:Int -> {v:Int | (v = (fib i))} @-}
fibhs :: Int -> Int
fibhs i = if i <= 1
        then liquidAssume (axiom_fib i) 1
        else liquidAssume (axiom_fib i)
              ((fibhs (i - 1)) + (fibhs (i - 2)))
```

Fig. 6. Example usage of Liquid Haskell axioms.

that either branch of the conditional returns the correct value. Simply stating the axiom is insufficient: Liquid Haskell does not search through the context and try miscellaneous instantiations of universally quantified types, because doing so would be extremely expensive. Instead, `liquidAssume` is used to inject a refinement, in this case into the return values of `fibhs`. `liquidAssume` asserts the truth of the Boolean first argument and adds the consequences of that Boolean's refinement, assuming `true`, to the refinement of the second argument, which is then returned directly with an enriched type. This injection of `fib`'s definition into refinements of `fibhs`'s return values, along with the additional refinement of `i` in each branch of the conditional based on comparison to `1`, allows the SMT solver to fold the definition of `fib` in the return values' refinements, producing the correct type in each case.

Because `liquidAssume` can be used to axiomatize certain refinements, it could be accidentally abused to prove a falsehood, similarly to Coq's `Axiom`. We only use it for two safe cases. The first use is to inject refinements that are axioms of RGREFS (e.g., to implement refiners by using past observations to justify new stable refinements). The other use is to inject some predicate's definition or property in a key location, similarly to the use for `axiom_fib` above. This is sometimes necessary because the way measures are introduced to the SMT solver does not extend the background theories of the solver but instead is encoded into constructor refinements [Vazou et al. 2014b], so the type system requires occasional hints about properties such as that a measure only returns true for values with a certain constructor.

5.2.2. Embedding RGREFS *into Liquid Haskell.* To adapt rely-guarantee references to Haskell, we simplify the design slightly: We omit transitive heap access in predicates and relations. This sacrifices expressiveness (rely and guarantee relations will apply only to single heap cells) but comes with the additional benefit of eliminating containment, precision, and folding from the design (and, thus, from developers' minds).[11]

We also restrict the implementation to only reflexively splittable references–those whose guarantee relations imply their rely relations—and may therefore be freely duplicated (recall the discussion in Section 4.2). This sacrifices strong updates on thread-local data, but Haskell lacks support for linear values.

[11]This is also partly forced by Liquid Haskell's design. Liquid types in general are designed to infer *fully applied* refinements, which assumes every variable used is in scope during inference. Inference with heap parameters to predicates and rely/guarantee relations is complicated by the fact that heaps do not exist as explicit bindings in the program.

Despite these restrictions, our Liquid Haskell embedding is still very useful. As we demonstrate in Section 6.3, much of the lost expressivity can be recovered by combining RGREFS with other features of Liquid Haskell's dependent refinements, such as indexing a data type by a predicate.

Figure 7 gives slightly simplified type signatures, collapsing rely and guarantee for brevity. Our implementation tracks rely and guarantee separately and checks the required additional properties—as mentioned earlier, our Liquid Haskell implementation supports arbitrary reflexively splittable (Section 4.2) references, including asymmetric examples like read-only aliases. The remainder of this section describes the key components, but further details on some parts of the encoding (such as `downcast` or `rgCASpublish`) are explained only later in Sections 6.2 and 6.3 where they are used.

Figure 7 includes the `RGRef` primitive itself, a wrapper around `IORef` with a rely-guarantee protocol. The figure also gives types for the primitives for allocating, updating, and reading `RGRefs`, each wrapping the corresponding `IORef` operation and imposing stability and other checks using bounded refinements. In the case of `axiom_pastIsTerminal`, we encounter the limitation of refinement bounds mentioned in Section 5.2.1—that the bounds may not refer to concrete function parameters—and work around this by taking a function argument that acts as an explicit proof term. This serves a similar role to the implicit bounds, but because it is a proper parameter, its refinements can refer to earlier concrete arguments. This specific proof term acts as a proof that for a particular previously observed value `v`, any value related to `v` by the rely `r` must also be `v`—evidence that the predicate $\lambda x.x = v$ is stable with respect to r. This is a specialized refiner, for the case where a reference may not be updated after it holds a specific value. We call this the terminal value of the reference, and we will later use it in conjunction with `liquidAssume` to refine based on dynamic observations. It is always sound to use `liquidAssume` with `axiom_pastIsTerminal` because the latter's proof term checks validity of introducing the new refinement.

Figure 7 also includes three *measures*—uninterpreted functions in the SMT solver—for indicating that a value is a past, final, or initial value of a given `RGRef` and an axiom `axiom_pastIsTerminal` for coercing a past value to a terminal value when the rely would not permit further change. This is a Liquid Haskell specialization of a refiner (`observe-field`) from the metatheory and COQ embedding, simplified to only the family of predicates that identify an exact value.

The more general refiner equivalent that introduces any new stable predicate based on a previously observed value is `injectStable`. `injectStable` takes an RGREF and a new predicate q constrained to be stable with respect to the rely r, along with evidence that some previous value stored in the cell satisfied q (enforced by requiring that the past value is refined by q, not the reference's predicate p). Because q is true of some value previously stored in `ref`, and it is stable, any current (or future) value stored in `ref` must also satisfy q, so the implementation casts `ref` to an RGREF indexed by q rather than p, with the constraint that the two references point to the same cell in memory. `downcast` is a stronger related primitive discussed in Section 6.3.2.

For a familiar example, consider a lock free monotonic counter implemented in Liquid Haskell using RGREFS and an RGREF wrapper around Haskell's `atomicModifyIORef`:

```
{-@ alloc_counter :: () -> IO (RGRef<{\x -> x > 0}, {\x y -> x <= y}> Int) @-}
alloc_counter :: () -> IO (RGRef Int)
alloc_counter _ = newRGRef 1
{-@ atomic_inc :: RGRef<{\x -> x > 0}, {\x y -> x <= y}> Int -> IO () @-}
atomic_inc :: RGRef Int -> IO ()
atomic_inc r = atomicModifyRGRef r (\x -> x + 1)
```

```
{-@ data RGRef a <p :: a -> Prop, r :: a -> a -> Prop > =
    Wrap (rgref_ref :: IORef a<p>) @-}
data RGRef a = Wrap (IORef a) deriving Eq
{-@ newRGRef :: forall <p :: a -> Prop, r :: a -> a -> Prop >.
                  {x:a<p> |- a<r x> <: a<p>}
                  {x:a<p> |- {v:a | v = x} <: a<r x> }
                  e:a<p> -> IO (RGRef <p, r> a) @-}
{-@ modifyRGRef :: forall <p :: a -> Prop, r :: a -> a -> Prop >.
                  {x:a<p> |- a<r x> <: a<p>}
                  rf:(RGRef<p, r> a) -> f:(x:a<p> -> a<r x>) -> IO () @-}
{-@ measure pastValue :: RGRef a -> a -> Prop @-}
{-@ measure terminalValue :: RGRef a -> a @-}
{-@ measure shareValue :: RGRef a -> a @-}
{-@ assume axiom_pastIsTerminal ::
    forall <p :: a -> Prop, r :: a -> a -> Prop>.
      ref:RGRef<p,r> a ->
      v:{v:a | (pastValue ref v)} ->
      (x:{x:a | x = v} -> y:a<r x> -> {z:a | ((z = y) && (z = x))}) ->
      { b : Bool | (((terminalValue ref) = v) <=> (pastValue ref v))} @-}
{-@ assume readRGRef :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
                  x:RGRef<p, r> a -> IO ({res:a<p> | (pastValue x res)}) @-}
{-@ assume injectStable :: forall <p :: a -> Prop, q :: a -> Prop,
                                 r :: a -> a -> Prop>.
                  {x::a<q> |- a<r x> <: a<q>}
                  ref:RGRef<p,r> a ->
                  {v:a<q> | (pastValue ref v)} ->
                  {r : (RGRef<q,r> a) | (ref = r)} @-}
{-@ assume downcast :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
                { x::b |- b<r x> <: b<p> }
                ref:RGRef<p,r> a ->
                {v:b | pastValue ref v } ->
                {r : RGRef<p,r> b | ref = r } @-}
{-@ rgCAS :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
            {x::a<p> |- a<r x> <: a<p>}
            Eq a =>
            RGRef<p,r> a -> old:a<p> -> new:a<r old> ->
            IO Bool
@-}
{-@ rgCASpublish :: forall <p :: a -> Prop, r :: a -> a -> Prop
                           pb :: b -> Prop, rb :: b -> b -> Prop>.
            {x::a<p> |- a<r x> <: a<p>}
            {x::b<pb> |- b<rb x> <: b<pb>}
            {x::b<pb> |- {v:b | v = x} <: b<rb x>}
            Eq a =>
            e:b<pb> ->
            RGRef<p,r> a ->
            old:a<p> ->
            new:(({r:(RGRef<pb,rb> b) | shareValue r = e}) -> a<r old>) ->
            IO Bool
@-}
{-@ atomicModifyRGRef :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
                  {x::a<p> |- a<r x> <: a<p>}
                  rf:(RGRef<p, r> a) ->
                  f:(x:a<p> -> a<r x>) ->
                  IO () @-}
```

Fig. 7. Core RGREFS in Liquid Haskell.

Program	Lines of Code	Lines of Proof
Atomic Increment	10	2
Treiber Stack	42	63
Michael-Scott Queue	78	162
Union-Find	119	1433

Fig. 8. Lines of code and proof.

6. PUSHING EXPRESSIVITY AND AUTOMATION

This section gives an overview of the case studies we have performed and presents the details of two substantial verification case studies using concurrent RGREFS. We have used our implementations to verify invariants for

—an atomic counter (Section 3.2.1),
—a Treiber stack [Treiber 1986] (COQ; Section 3.2.2),
—a lock-free linearizable union-find implementation due to Anderson and Woll [1991] (COQ; Section 6.1),
—a tail-less Michael-Scott queue [Michael and Scott 1996] (COQ; see our implementation),
—a lock-free linked list with lazy deletion [Harris 2001] (Liquid Haskell; Section 6.2), and
—a lock-free set implemented as a sorted linked list with lazy deletion (Liquid Haskell; Section 6.3).

Figure 8 gives the lines of code[12] and proof[13] our examples proven via our COQ DSL, which gives a rough estimate of the proof burden relative to code size.[14] For smaller examples, the code and proof size are comparable, while the proofs for union-find, with significantly richer invariants, are more substantial. No special effort was made to minimize or aggressively automate proofs.

6.1. Lock-Free Union Find

Anderson and Woll give a lock-free linearizable union-find implementation [Anderson and Woll 1991] using ranks and path compression to improve performance [Cormen et al. 2009]. We have used RGREFS to verify the structural invariants for this data structure as well as that the only modifications are union, rank update, and path compression operations.

Recall that a union-find data structure supports unioning sets and looking up set membership, represented by a representative element of the set. The structure is a forest of inverted trees (children point to parents), where each tree represents one set, and the root element represents the set. Lookup proceeds by following parent links to and returning the root. Unioning two elements' sets occurs by looking up the respective sets' roots, and if they differ, reparenting one (which previously had

[12]Data structure, relation, and invariant definitions, as well as algorithms and type class instances for field access.
[13]Lemmas for stability, precision, folding, containment, reachability, and discharge of type checking obligations.
[14]COQ includes the `coqwc` tool to count lines of specification and proof, but it interprets new `Ltac` definitions as specification and the body of a `Program Definition` as we use for our algorithms as part of a proof, making it unsuitable for our needs. We derived these numbers by removing all blank, comment-only, or import lines from the working COQ files and partitioning the remaining lines.

no parent) to the other. To improve asymptotic complexity, two optimizations are typically applied [Cormen et al. 2009]. First, each node is equipped with a *rank*, which over-approximates the longest path length from a child to that node. Unions then reparent the lower-ranked root to the other to avoid extending long child-to-root paths. Second, *path compression* updates the parent of each node traversed during lookup to be closer to the root of the set, amortizing the cost of earlier lookups with faster future lookups.

Anderson and Woll use a fixed-size array with a cell for each element in the union-find instance, where each cell points to a two-field record with the rank and parent index for that element. To simulate a 2CAS in the original article, they make each record immutable and perform CAS operations on the pointer-sized cells of the array. A root is represented by an element that is its own parent. The key invariants are (1) each node has a rank no greater than its parent, (2) when a cell and its parent have equal ranks, the child has the lesser index in the array, and (3) all parent chains terminate.

We used concurrent RGREFS to verify that the key invariants hold. To our knowledge, this is the first machine-checked proof of invariants for this algorithm. This verification is a contribution by itself but also demonstrates the generality of rely-guarantee references and their natural applicability to concurrent data structures: We were unaware of this algorithm when designing concurrent RGREFS but found expressing the union-find structure in our system to be quite natural.

We briefly outline the verification and present verification of path compression in more detail. Our proofs are available with our DSL implementation (Section 5.1).

The key invariants 1–3 are embodied in the refinement on the reference to the array, ϕ in Figure 9. The rely/guarantee δ (for change) relation permit reparenting a root to a node with a greater rank (or equal rank and greater index) for unions, increases to root ranks (used occasionally in union), and the reparenting required for path compression (which has subtleties detailed below). The refinement ϕ is stable with respect to the relation δ, and each heap modification in the implementation respects the relation's restrictions. Proving the guarantee is respected in each case relies on the same principles used for the Treiber stack (Section 3.2.2): refining references based on observations and combining CAS operations with weakening strongly refined references (e.g., those exactly describing the contents of an immutable cell). So the same basic principles used to verify the relatively simple Treiber stack scale up to a substantially more complex structure.

RGREFS' decoupling of abstraction and interference (Section 2.2) supports modular verification, allowing Anderson and Woll's same-set operation (typically absent from union-find implementations) to be verified separately from other operations. In some other work [Pilkiewicz and Pottier 2011; Jensen and Birkedal 2012], adding a same-set operation to an existing implementation requires re-verifying all operations because the two-state invariants are tied to abstraction. In our case, the same-set operation is simply verified after the other operations as in modern concurrent program logics.

Verifying Path Compression. Figure 9 gives the code for set lookup, which performs path compressions as it looks up nodes. This is the most challenging union-find verification obligation. To support path compression, δ permits any reparenting among elements of the same set that preserves the invariants ϕ, because requiring a path from the node being updated to the new parent (that the path gets shorter) is too strong (false). At the exact moment a node's parent pointer is bumped, it is possible that other threads may have already advanced the current parent to be closer to the root than the soon-to-be-set parent. This not only means that there may be no path from the updated node to its new parent at the time of update, but the write may in fact make the path to the root *longer* momentarily.

```
(** * Lock-Free Linearizable Union-Find *)
type cell (n:nat) : ⋆ := mkCell :: rank:nat -> parent:(Fin.t n) -> cell n;;
type uf (n:nat) : ⋆ := Array n (ref{cell n|any}[local_imm,local_imm]);;
type chase (n:nat) (x:uf n) (h:heap) (i : Fin.t n) : Fin.t n -> Prop :=
  (* chase n x h i j is provable when there is a chain of parent links
     reaching from i to j in array x *) ... ;;
type ϕ (n:nat) : hpred (uf n) := ...
 (* require child ranks less than parent ranks (1) with ties
    broken by index order (2), all parent chains acyclic (3) *) ;;
type δ (n:nat) : hrel (uf n) := ...
  | path_compression : forall x f c h h', ϕ n x h ->
    (* install new cell c at index f, preserving rank, no path from new
       cell parent back to f (don't create cycle), f and f's new parent
       are in the same set, f rank-sorted below f's new parent *) ->
    δ n x (array_write x f c) h h'.
let rec Find {n:nat} (r:ref{uf (S n)|ϕ}[δ,δ]) (f:Fin.t (S n))
    : IO (Fin.t (S n)) :=
  observe-field r ⟶ f as c in (λ x h, sameset f ((h[c]).parent) x h /\
                        rankSorted (h[c]) (h[x<|((h[c]).parent)|>]));
  observe-field c ⟶ parent as p in (λ x h, x.parent = p);
  if (p == f) then (return f) (* found root --- f's parent = f *)
  else (observe-field c ⟶ rank as rnk in (λ x h, x.rank = rnk);
        observe-field r ⟶ p as p_ptr in
     (*A*) (λ x h, sameset p ((h[p_ptr]).parent) x h
     (*B*) /\ (rankSorted (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>]))
     (*C*) /\ (rankSorted (h[c]) (h[p_ptr]))
     (*D*) /\ ((h[p_ptr]).parent ≠ p ->
                nonroot_rank p ((h[p_ptr]).rank) x h)
     (*E*) /\ ((h[p_ptr]).parent ≠ p ->
                ( rankSorted_strict (h[p_ptr]) (h[x<|((h[p_ptr]).parent)|>])
                 \/ fin_lt p ((h[p_ptr]).parent))));
        observe-field p_ptr ⟶ parent as gp in (λ x h, x.parent = gp);
        gp_cell <- Alloc! (mkCell _ rnk gp ) ;
        _ <- fCAS( r ⟶ f, c, convert gp_cell);
        Find p);;
```

Fig. 9. A lock-free union find implementation [Anderson and Woll 1991] using RGREFS, omitting interactive proofs. `a<|i|>` accesses the `i`th entry of array `a`. The type `Fin.t n` is (isomorphic to) a natural number less than n—a safe index into the array.

Thus, to verify that the lookup operation's path compression operation (the `fCAS`[15] at the end of the procedure) respects the compression case of δ, we must accumulate enough stable predicates as we traverse the structure to prove that f and its new parent are in the same set and that their ranks and indices are appropriately sorted. To do so, we make heavy use of the `observe-field` construct. Note that rewriting uses of `observe-field` to simple field accesses yields just a few lines of straightforward code, almost the same as in Anderson and Woll's article. We take advantage of the fact that the cell for each element is immutable; reading a field of the array is effectively equivalent to reading both fields of the cell. Stepping through the `Find` routine, we first read the array field of the element being sought, observing that future values of the array field will preserve the current set membership and at most increase its rank. If

[15]`fCAS` is `CAS` on a field. Its typing resembles CAS, but the guarantee is proven assuming update to the specified field only.

the node is its own parent, then the search is complete. Otherwise, we find element f's grandparent and attempt to update f's parent to the grandparent.

Most of the interesting stable assertions arise when reading the parent out of the array (`observe-field r --> f...`). There we make the same observations made for f (markers `A`, `B`), as well as relating the current parent rank to f's recent rank (`C`); noting that if the parent is not the root, its rank is fixed permanently (`D`); and if the parent is not the root, its rank and identity order all of its future parents (f's grandparents) later than it (`E`).[16] With these array refinements relating the grandparent to f, plus the sharing idiom for the replacement node for f, the compression case of the δ relation is provable: preserving rank, set membership, and proper parent-chain ordering by rank and identity.

6.2. Lock-Free Linked List

One of the test cases for the Glasgow Haskell Compiler[17] is a lock free linked list along the lines of that originally proposed by Harris [2001] and Herlihy and Shavit [2008]. To ground our discussion, we first cover some background on lock-free linked list algorithms and how Haskell's design affects their implementation. Then we discuss the verification of two-state invariants for the linked list using Liquid Haskell with RGREFS.

Lock-Free Linked Lists. A lock-free linked list has a basic singly linked list structure as its basis—nodes with elements and tail pointers.

Manipulating the head or tail of the list is relatively straightforward. Adding a node to the head of the list is exactly the push operation on the Treiber stack [Treiber 1986] (Section 3.2.2), while appending a node is precisely the enqueue operation from a (tail-less) Michael-Scott queue [Michael and Scott 1996]. The algorithms get more sophisticated once modification of *interior* links of the list is required, as for deletion, or sorted insertion [Harris 2001; Heller et al. 2006].

Deletion occurs in two phases: logical deletion and physical deletion. Logical deletion logically removes a node from the data structure—lookups should not return that node—but leaves the node physically linked into the list. Physical deletion is the actual physical unlinking—setting the previous node's next pointer to the deleted node's next pointer—and is performed by subsequent operations. This separation of physical and logical deletion is typically referred to as a *lazy* deletion due to the delayed physical removal and forms the basis for a number of other lock free data structures, including sets [Heller et al. 2006; O'Hearn et al. 2010], which we revisit in Section 6.3.

This phase separation is required because without the logical deletion step marking a node for deletion, it is impossible to tell if a node has been physically removed from the data structure. A naïve CAS on the predecessor's next pointer to remove a node would succeed even if another thread had already removed the *predecessor* from the list. The logical deletion flag adds enough information to make the CAS on a predecessor fail if the predecessor has been deleted. This requires that the same atomic operation that would update the next pointer also checks the deletion flag.

Mechanisms for this vary, from using a 2CAS[18] to indicating deletion by setting the low-order bit in the next pointer (since the node should be at least word-aligned in memory) to the technique used here that relies on the standard memory layout of algebraic data types in functional programming languages.

[16]This helps establish a total rank+identity ordering on f and its ancestors.
[17]`testsuite/tests/concurrent/prog003/CASList.hs` in the compiler source tree.
[18]Compare-and-swap on two adjacent (typically aligned) machine words of memory. Most architectures, for example, x86/amd64, include a double-width CAS.

```
{-@ data List a = Null
    | DelNode (RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a))
    | Node a ((RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a)))
    | Head (RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a))
@-}
data List a = Node a (RGRef (List a)) | DelNode (RGRef (List a))
            | Null | Head (RGRef (List a)) deriving Eq
{-@ predicate ListRG X Y = (((isNull X) && (isNode Y)) ||
     ((isNode X) && (isDel Y) && ((nxt X) = (nxt Y))) ||
     ((isNode X) && (isNode Y) && (isDel (terminalValue (nxt X)))
      && ((val X) = (val Y)) && ((nxt (terminalValue (nxt X))) = (nxt Y))) ||
     ((isHead X) && (isHead Y) && (isDel (terminalValue (nxt X)))
      && ((nxt (terminalValue (nxt X))) = (nxt Y))) ||
     (X = Y)) @-}
{-@ assume isDelOnly :: x:List a ->
    {v:Bool | ((isDel x) <=> ((not (isHead x)) &&
                            (not (isNull x)) && (not (isNode x))))} @-}
isDelOnly :: List a -> Bool
isDelOnly x = undefined
```

Fig. 10. Core definition of linked list.

Compare-and-Swap in Haskell. In this GHC test case, CAS is performed not on a field of the node but also on the *whole node*, because the nodes themselves are actually immutable in Haskell; a CAS on the whole object amounts to a CAS on a pointer to an immutable record. This is due to a subtlety in Haskell's runtime system: The GC makes aggressive use of immutability assumptions and may occasionally duplicate immutable data.[19] This can cause uses of a raw hardware CAS to fail more often if the GC duplicates the expected-value reference. Our verification uses a slightly stronger primitive from the GHC test suite but could also trivially be changed to use GHC's more unstable `casMutVar#`, though the CAS would then fail more often at runtime.

The GHC test case we are interested in implements its own CAS on top of `atomicModifyIORef`:

```
atomCAS :: Eq a => IORef a -> a -> a -> IO Bool
atomCAS ptr old new = atomicModifyIORef ptr (\ cur -> if cur == old
                                                      then (new, True)
                                                      else (cur, False))
```

This CAS implementation matches the semantics for regular CAS, except it uses the equality comparison from the `Eq` type class (the == above) to compare the old and new values instead of a hardware pointer comparison. Technically this overcompensates for the unstable pointer equality in the naïve CAS case by making the CAS succeed for objects that are equal in the sense of referential transparency, which includes comparing values against freshly computed equivalents. The code for the linked list we verify would be correct under the proper CAS semantics (the CAS arguments it provides for the expected old values are the variables bound to the results of previous reads), so we proceed with our verification.

A List. This section walks through the types involved in the lock-free linked list with a focus on the rely relation used, as well as the verification of the delete procedure, which covers most cases of the rely.

Figure 10 gives the essential definitions from the lock-free linked list. First, the type `List a` is defined, the type of list nodes. Here an algebraic datatype is used to

[19]See http://www.haskell.org/haskellwiki/AtomicMemoryOps.

disambiguate different node roles—head (sentinel node), valid node, logically deleted node, and a terminal node. The Liquid Haskell declaration of the `List a` type simply instantiates the `p` and `r` parameters of the `RGRef` type. (Recall that this is a simplification for presentation, and the actual implementation separates rely and guarantee.) Not shown are a set of Liquid Haskell *measures* (predicates), `isNull`, `isNode`, `isDel`, and `isHead`—each true only for the corresponding constructor. `isDelOnly` is an axiom stating that satisfying `isDel` is mutually exclusive with the other node types—required due to the way Liquid Haskell encodes measures [Vazou et al. 2014b] (see discussion of `liquidAssume` in Section 5.2.1). Also omitted are `myNext`, a partial function returning the next-link reference from a node, and `nxt`, a measure to access the same information in refinements.

The `ListRG` predicate itself (Line 8) serves as the rely and guarantee relation for interior pointers inside the linked list. The predicate is a disjunction of five possible transitions, listed here in the order they appear in Figure 10:

(1) Appending a node: replacing a `Null` node with a `Node`.[20]
(2) Logical deletion: marking a node for deletion without removing it from the spine of the linked list.
(3) Physical deletion: physically removing a previously logically deleted node. In this case, the old and new node must carry the same value (preserve the value), and the new next pointer must point to the removed node's next node, thus removing only one node. The phrasing for this in terms of `terminalValue` requires updates in this case to also prove that the removed node has a terminal value, which in this case means it is a logically deleted node.
(4) Physical deletion at the head: as in the previous case, but when deleting the first node of the list.
(5) Reflexivity: permit no-ops.

Figure 11 gives the code for deleting a node with a given value from the list. Traversal to locate a node proceeds naturally. In the case the node is found and not yet deleted, the code attempts to CAS the `curPtr` from the node `curNode` just read[21] to a newly allocated `DelNode` preserving `curNode`'s next pointer (Line 24). The SMT solver naturally dispatches the guarantee proof using the logical deletion clause of `ListRG`.

The most interesting case is mid-traversal, when the code crosses a logically deleted, but physically present, node (Line 28). In this case, the code uses a CAS to swap a pointer to the logically deleted node—the previous-node pointer `prevPtr`—with a pointer to a copy of `curNode` with an updated next pointer. The code is duplicated slightly for performing a CAS on a `Head` (Line 31) or `Node` (Line 38) variant (if the previous node has been logically deleted, there is no point in updating its next pointer), but each requires proving a physical deletion case of `ListRG`. Recall from earlier that each of these cases requires preserving the node's main structure (node type and value if present) and proving that the new next pointer is the same as the next pointer in the old next-node's referent. This is phrased in terms of `terminalValue`. Recall from Figure 7 that reading from an `RGRef` produces a value refined with a `pastValue` predicate witnessing that the value was once stored in the cell it was read out of. To take advantage of this, we use Liquid Haskell's `liquidAssume` to inject properties into

[20]Because our LH encoding lacks transitive heap access, we cannot directly encode the acyclicity requirement as we do for our Michael-Scott Queue implementation in Coq—there is no predicate which explicitly states that the list is acyclic.
[21]Recall that although this appears semantically to be an atomic value comparison, and is due to the overcompensation for the CAS approximation used, a proper CAS would see this as a compare-and-swap of a pointer to an immutable record.

```
 1  {-@ terminal_listrg ::
 2      rf:RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a) ->
 3      v:{v:List a | (isDel v)} ->
 4      x:{x:List a | (x = v)} -> y:{y:List a | (ListRG x y)} ->
 5      {z:List a | ((x = z) && (z = y))} @-}
 6  terminal_listrg :: RGRef (List a) -> List a -> List a -> List a -> List a
 7  terminal_listrg rf v x y = liquidAssume (isDelOnly x) y
 8
 9  delete :: Eq a => ListHandle a -> a -> IO Bool
10  delete (ListHandle head _) x =
11    do startPtr <- readIORef head
12       go startPtr
13     where
14     {-@ go :: RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a) ->
15               IO Bool @-}
16     go prevPtr =
17       do prevNode <- (readRGRef prevPtr)
18          let curPtr = myNext prevNode -- head/node/delnode all have next
19          curNode <- (readRGRef curPtr)
20          case curNode of
21            Node y nextNode ->
22               if (x == y)
23               then -- node found and alive
24               do b <- rgCAS curPtr curNode (DelNode nextNode)
25                  if b then return True else {- spin -} go prevPtr
26               else go curPtr -- continue
27            Null -> return False -- reached end of list
28            DelNode nextNode -> {- try physical deletion; ignore failure -}
29              case prevNode of
30                Node v _ ->
31                  do b <- rgCAS prevPtr prevNode
32                                 (Node v (liquidAssume
33                                            (axiom_pastIsTerminal curPtr curNode
34                                              (terminal_listrg curPtr curNode))
35                                            nextNode))
36                     if b then go prevPtr else go curPtr
37                Head _ ->
38                  do b <- rgCAS prevPtr prevNode
39                                 (Head (liquidAssume
40                                          (axiom_pastIsTerminal curPtr curNode
41                                              (terminal_listrg curPtr curNode))
42                                          nextNode))
43                     if b then go prevPtr else go curPtr
44                DelNode {} -> go curPtr -- parent deleted; ignore
```

Fig. 11. Excerpt from a lock-free linked list implemented using Liquid RGREFS: definitions and deletion.

the refinement of a relevant value. `liquidAssume` causes the SMT solver to assume the truth of some refinement—an axiom—and inject it into some value's refinement. In this case, we use `liquidAssume` to inject an instantiation of the `axiom_pastIsTerminal` axiom into the refinement of `nextNode`. The use of this axiom bounds its instantiation to cases where the observed value, according to the rely and guarantee, must be the final value of the reference cell. Our proof of this uses the `terminal_listrg` "lemma" (Line 6) that proves that a deleted node remains deleted.[22] Note that we use `terminal_listrg` as a proof of an implication—`axiom_pastIsTerminal` takes a function argument as an explicit stability proof. This hint in `nextNode`'s refinement allows the SMT solver to conclude that the terminal value of the node being skipped had `nextNode` as its next field.

Because physical deletion is shared across all operations, these physical deletion CAS proofs arise in type-checking most of the operations.

6.3. Lock Free Set by Recovering Transitive Heap Access

At first sight, the loss of transitive heap access in predicates, rely, and guarantee relations may seem severe, even if examples like a lock-free linked list can be implemented this way. But in our experience with the Coq prototype, we rarely used the transitive heap access to reach more than one level into a recursive data structure, and in the presence of data types indexed by predicates, there are other ways to constrain deeper parts of the heap. Furthermore, lock-free data structures typically rely heavily on immutability and reasoning about very local changes, and even global invariants on recursive structures can be enforced using only local reasoning.

This section presents a case study showing that much of the lost expressivity can be recovered through careful use of other dependent typing features *already implemented in Liquid Haskell*.

The presence of a restricted form of dependent inductive data types in Liquid Haskell provides a path to recovering the fragment of transitive heap access expressivity that we have actually found useful. This section uses this feature to extend the lock-free linked list just discussed to a lock-free set implemented as a *sorted* linked list [Heller et al. 2006]. Note that throughout this section, `Set` refers to a Liquid Haskell implementation of a set abstraction rather than Coq's predicative universe of small types.

The key technique to imposing sorting is to index a data type by a predicate on the value and instantiate the predicate in the successor link with a different predicate. In particular, we will index nodes of the set representation by a lower bound on the value, and successor pointers will use the current element to impose a stronger lower bound on the rest of the list. Intuitively, this means roughly:

```
{-@ data Set a <p :: a -> Prop> =
     Node (val :: a<p>) (nxt :: RGRef<...> (Set <{\v -> v > val}> a) | ... @-}
data Set a = Node a (RGRef (Set a)) | ...
```

This is sufficient to ensure the list is sorted but insufficient to support insertion and deletion. The reason is that with the successor refinement above, insertion and deletion require taking a cell reference with a given lower bound on the value, and using it as if it had a different (still true) lower bound. For insertion, in a list with elements 3 and 5, consider inserting 4. The 3-node's successor pointer will have type `Set<{\v -> v > 3}> Int`, but the successor node for the newly allocated 4-node requires type `Set<{\v -> v > 4}> Int`. The dynamic checks required to find the insertion point ensure that the successor (the 5-node) semantically satisfies the latter type, but this must be reflected back into the type system. This is what refiners were meant for, but here we must refine the parameter to the base type referenced by the `RGRef` rather than the refinement carried by the reference itself. Subsequent deletion of 4 poses a similar problem, reversing the given and needed type when updating the 3-node's successor. In the pure functional case, deletion would be handled by covariant subtyping, but the base referent type for `RGRef`s, as for `IORef`s, is invariant for subtyping.

[22] `terminal_listrg` itself gives the SMT solver the hint that node types are mutually exclusive.

We require different solutions for insertion and deletion. Once we have solutions for these cases, the types for the set implementation ensure sortedness, and even if we hand a set to "unverified" code, that code may not violate sortedness, so the insertion, deletion, and lookup routines we define below will continue to operate correctly; see our implementation for an example client that exploits this.

6.3.1. Deletion via Slack Variables. Consider deleting the node holding the value 4 from the list containing 3, 4, and 5. This requires updating the 3-node's successor pointer to refer to the 5-node by copying the 4-node's successor pointer.

The trouble is that with the intuitive sketch given above, this requires replacing the 3-node's successor pointer of type `Set<{\v -> v > 3}> Int` with the 4-node successor pointer `next` of type `Set<{\v -> v > 4}> Int`. An equivalent of the refiner from the Coq DSL does not solve this problem, because this would require reading the value of the new *successor* (the node after the deleted node) and dynamically inspecting it to refine its properties. Neither of these are required for sorted insertion.

The key to permitting this replacement is recognizing that *there exists a value* x *such that* $3 \leq x$ *and* `next :: Set<{\v -> v > x}> Int`. That is, we can use a *slack variable*[23] to relax the relationship between a node's value and the type-based lower bound on the successor's value.

Because we cannot existentially quantify the slack variable,[24] we must modify the node structure to explicitly record the slack variable. Extending our earlier intuitive sketch:

```
{-@ data Set a <p :: a -> Prop> =
    Node (val :: a<p>)
         (slack :: {v:a | val <= v})
         (nxt :: RGRef<...> (Set <{\v -> v > slack}> a)) | ... @-}
data Set a = Node a a (RGRef (Set a)) | ...
```

When physically deleting a node, we update the predecessor of the removed node by updating both the successor pointer and the slack variable.[25]

Safe Covariant Reference Subtyping. Readers familiar with other systems containing references with restricted mutation may wonder if a similar form of safe covariant subtyping is available on references. For example, reference immutability type systems [Tschantz and Ernst 2005; Zibin et al. 2007, 2010; Gordon et al. 2012] admit covariant subtyping on read-only references. Such a construct in Liquid Haskell would permit us to address physical deletion without adding a slack variable, because a node with a given lower bound lb also has any value less than lb as a lower bound.

We can also write a similar construct here for safe covariant subtyping on the main reference type here (A in $\mathsf{ref}\{A \mid P\}[R, G]$) and restrict its use to the case where the rely and guarantee ensure updates through the upcast reference preserve membership in the original type:

```
{-@ assume safe_covar :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
                { x::a |- a <: b }
                { x::a<p> |- a<r x> <: b }
                ref:RGRef<p,r> a ->
                {r : RGRef<p,r> b | ref = r } @-}
safe_covar :: RGRef a -> RGRef b
safe_covar r = unsafeCoerce r
```

[23]The term *slack variable* comes from integer linear programming, where inequalities such as $Ax \leq B$ are converted to equality constraints $Ax + s = B$ by introducing an extra non-negative variable s—a slack variable—that quantifies the difference of the original inequality [Chvatal 1983].

[24]Liquid Haskell allows existential quantification in predicates but not in types.

[25]In principle, this could be simplified by the addition of ghost state to Liquid Haskell.

Liquid Haskell accepts this trusted specification and can check its use. In our implementation, we use both this primitive (holding the slack variable constant in updates) for `find` and `delete` because of its convenience and the slack variable approach in `insert` because this technically lies outside the theory we have proven consistent (recall that the formal system in Section 4 does not include subtyping).

6.3.2. Insertion via Statically Checked Downcasts. Insertion faces the opposite problem of deletion. Insertion requires strengthening the lower bound on a node to prove that it belongs after the node being inserted. This is almost what refiners were designed for, but rather than changing the predicate portion of an RGREF , we are instead changing the index of the `Set` data type—from the perspective of RGREFS, this is a change of the base type component of the reference to a *subtype* of the original base type. Concretely, this is where we use the `downcast` primitive given in Figure 7:

```
{-@ assume downcast :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
                { x::b |- b <: a }
                { x::b |- b<r x> <: b<p> }
                ref:RGRef<p,r> a ->
                {v:b | pastValue ref v } ->
                {r : RGRef<p,r> b | ref = r } @-}
```

This axiom coerces an RGREF pointing to some `a` to an RGREF pointing to some subtype `b`. Concretely, this will permit us to coerce a reference to a `Set<{\v -> v > 3}> Int` into a reference to a `Set<\v -> v > 4> Int` (note that this latter type, by the covariant use of the index predicate in `Set`, is a subtype of the former). The key is the use of `pastValue` to observe that the value stored in the reference is also in the subtype and that the subtype itself is stable under the original rely (and guarantee). We give no formal account of this primitive's soundness, but, intuitively, this is justified by interpreting subtypes as predicates on the original types. `downcast` plays essentially the same role as refiners but for predicates that are tracked as indices of the reference's base type rather than through the predicate component of an RGREF. In both bases, observing that the reference has at some point held a value in a particular subset of the original base type, where that subset is closed under transitions according to the rely (i.e., the subset satisfying the predicate, or more precisely the new stable predicate introduced by refiners), justifies refining the reference to indicate the more precise subset.

The standard pattern for insertion of a value X is to traverse the list until a node with a larger element than X is found, tracking the predecessor of the "current" node, and then insert a node between the predecessor and current nodes. It is the predecessor's pointer to the current node (whose value is larger than X) that will be coerced with `downcast` to reflect that the program dynamically checks that the node's value is larger than the element being inserted. The result will carry X as the lower bound in the `Set` parameter, making it suitable for use in the new node.

6.3.3. On the Naturality of Our Encoding. This use of lower-bounding predicates in an inductive data type may initially appear to be a non-obvious contortion but is actually not far from idioms used in existing Liquid Haskell code. Our use of the indexing predicate could be seen as a natural extension of a technique espoused by the Liquid Haskell developers, who used a similar idiom to impose sortedness on pure lists.[26] In their case, the natural covariance of pure lists with lower bounds and the internal representation of index predicates applied to constructor arguments allow insertion and deletion to "just work." Our use of slack variables, safe reference covariance,

[26]http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/blog/2013/07/29/putting-things-in-order.lhs/.

```
{-@ predicate SetRG X Y =
    (((IsNull X) && (IsNode Y)) ||
     ((IsNode X) && (IsDel Y) && ((val X) = (val Y)) && ((nxt X) = (nxt Y))) ||
     ((IsNode X) && (IsNode Y) && (IsDel (terminalValue (nxt X))) && ((val X) = (val Y))
                && ((nxt (terminalValue (nxt X))) = (nxt Y))) ||
     ((IsHead X) && (IsHead Y) && (IsDel (terminalValue (nxt X)))
                && ((nxt (terminalValue (nxt X))) = (nxt Y))) ||
     ((IsNode X) && (IsNode Y) && ((val X) = (val Y))
                && (nxt X = nxt (shareValue (nxt Y)))) ||
     ((IsHead X) && (IsHead Y) && (nxt X = nxt (shareValue (nxt Y)))) ||
     (X = Y)
     )
@-}
```

Fig. 12. Relation used as rely and guarantee for Set references.

and (statically checked) downcasts are necessary, because our structure is built from mutable references instead of being a pure structure.

6.3.4. Rely-Guarantee for Set References. The rely-guarantee relation used here, `SetRG`, is an extension to `ListRG` from Section 6.2. It adds two cases for insertion: insertion after a node and insertion at the head of a list. In each case, we use the measure `shareValue` to refer to the value of a heap cell *at the time a node was shared*. Rather than moving to a full-blown Hoare triple as the Liquid Haskell authors have recently explored [Vazou et al. 2015], we instead rely on the `rgCASpublish` primitive from Figure 7. This primitive accepts a value to allocate in an `RGRef`, an old value (as in any CAS), and a (pure) function that computes a new value from the allocated reference. Crucially, the refinement type for the function may assume that the value at the time of updating the heap is exactly the value provided for allocation.

In the core language, this corresponds to a heap write where the value carried in is substructural (T-LinStore in Figure 4) and can therefore carry strong refinements into the heap write that shares the previously linear data. This primitive combines the separate allocation and write steps into one primitive that internally ensures the allocated reference is not duplicated (and modified) before sharing.

`SetRG` (Figure 12) uses `shareValue` to constrain the successor of an inserted node to be the node it is being inserted in front of. Naïvely, this might seem to allow allocating a reference with an appropriate initial value, mutating it, and then inserting a node. But `rgCASpublish` only exposes the `shareValue` in the scope of the allocation and sharing.

6.3.5. Set Insertion. Figure 13 gives the Liquid Haskell code for insertion into the set as a sorted linked list. Other operations—search, deletion—are similar to the lock-free linked list from Section 6.2, and we omit the case that performs physical deletion (which is also very similar to the linked list case).

At a high level, the code walks through the list, tracking a previous node (via `prevPtr`) and a current node (`curPtr`), using a refinement on the tail-recursive `go` to ensure the value of the previous node (if any) is smaller than the element being inserted. This is not only a good partial correctness property to have but is in fact necessary when type checking the CAS through `prevPtr`. To update the previous node's successor to a node with value `x`, the inserted node's refinement index—the lower bound on its value—must use the previous node's value as the lower bound.

Figure 13 depends on three cast operations: two for refining the lower bound of the `Set` type (as explained in Section 6.3.2) and one for incorporating the results of a comparison between the value being inserted and another node's value to satisfy the refinement of a recursive call to `go` that the previous pointer's referent has a smaller value than the value being inserted.

```
insert :: Ord a => SetHandle a -> a -> IO Bool
insert (SetHandle head _) x =
  do startPtr <- readIORef head
     go startPtr
  where
     -- Note that the RGRef predicate ensures that either
     -- (A) we're at the start of the list, or
     -- (B) inserting x just after prevPtr will preserve sortedness
     -- up to that insertion.
     {-@ go :: forall <p :: a -> Prop >.
              RGRef<{\nd -> IsHead nd || val nd < x},{\x y -> (SetRG x y)}> (Set <p> a)
              -> IO Bool @-}
     go !prevPtr =
        do prevNode <- readRGRef2 prevPtr
           -- note this next line skips over the head
           let curPtr = myNext prevNode
           curNode <- (readRGRef curPtr)
           case curNode of
             Node y lb nextNode ->
                if x == y
                then return False --- already present, not added
                else if y < x
                     then go (prove_lb curPtr x curNode)
                     else --- insertion! add between previous and current
                        case prevNode of
                          Node prevVal vlb q ->
                            do let newNode = (Node x x (downcast_set curPtr x curNode))
                               b <- rgCASpublish newNode prevPtr prevNode
                                                 (\ptr -> Node prevVal prevVal ptr)
                               if b then return True else go prevPtr
                          Head _ ->
                            do let newNode = (Node x x (downcast_set curPtr x curNode))
                               b <- rgCASpublish newNode prevPtr prevNode (\ptr -> Head ptr)
                               if b then return True else go prevPtr
                          DelNode _ _ _ ->
                            do newhd <- readIORef head
                               go newhd -- predecessor deleted, restart
             Null -> case prevNode of
                       Node prevVal vlb q ->
                         do let curPtrC = downcast_set_null curPtr x curNode
                            let newNode = (Node x x curPtrC)
                            b <- rgCASpublish newNode prevPtr prevNode
                                              (\ptr -> Node prevVal prevVal ptr)
                            if b then return True else go prevPtr
                       Head _ ->
                         do let newNode = (Node x x (downcast_set_null curPtr x curNode))
                            b <- rgCASpublish newNode prevPtr prevNode
                                              (\ptr -> Head ptr)
                            if b then return True else go prevPtr
                       DelNode _ _ _ ->
                         do newhd <- readIORef head
                            go newhd -- predecessor deleted, restart
             DelNode v lb nextNode -> ... {- physical deletion as in list -}
```

Fig. 13. Insertion into a lock-free set, represented as a sorted linked list.

```
{-@ downcast_set :: forall <p :: a -> Prop>.
    ref:RGRef<{\x -> (1 > 0)},{\x y -> (SetRG x y)}> (Set <p> a) ->
    x:a ->
    {v:(Set <p> a) | pastValue ref v && x < val v } ->
    {r:RGRef<{\x -> (1 > 0)},{\x y -> (SetRG x y)}> (Set <{\v -> x < v}> a) | r = ref } @-}
downcast_set :: RGRef (Set a) -> a -> Set a -> RGRef (Set a)
downcast_set r x v = downcast r v

{-@ downcast_set_null :: forall <p :: a -> Prop>.
    ref:RGRef<{\x -> (1 > 0)},{\x y -> (SetRG x y)}> (Set <p> a) ->
    x:a ->
    {v:(Set <p> a) | pastValue ref v && IsNull v } ->
    {r:RGRef<{\x -> (1 > 0)},{\x y -> (SetRG x y)}> (Set <{\v -> x < v}> a) | r = ref } @-}
downcast_set_null :: RGRef (Set a) -> a -> Set a -> RGRef (Set a)
downcast_set_null r x v = downcast r v

{-@ prove_lb :: forall <z :: a -> Prop>.
    ref:RGRef<{\x -> (1 > 0)},{\x y -> (SetRG x y)}> (Set <z> a) ->
    x:a ->
    {n:(Set <z> a)<{\s -> val s < x}> | (IsNode n || IsDel n) && (pastValue ref n) } ->
    {r:RGRef<{\n -> (IsNode n || IsDel n) && val n < x},{\x y -> (SetRG x y)}> (Set <z> a)
      | r = ref } @-}
prove_lb :: RGRef (Set a) -> a -> Set a -> RGRef (Set a)
prove_lb ref x v = (injectStable ref (v))
```

Liquid Haskell currently does not verify the bodies of these casts because they encounter an incompleteness of Liquid Haskell's inference: Liquid Haskell fails to instantiate the axioms `downcast` and `injectStable` correctly, but if we duplicate the axioms and manually instantiate the type parameter `a` with `Set a` in the duplicates, then the lemmas are verified. We take this as a necessary, but clearly safe trusted cast.

7. MANUAL VS. AUTOMATED RGREF PROOFS

Our two implementations of concurrent RGREFS allow us to compare the experience of verifying invariants with RGREFS in an interactive theorem prover against an automated setting. Here we describe some qualitative differences.

Proofs in the COQ *DSL*. Verifying these structures inside COQ exposes information that makes sophisticated proofs possible. In our union-find case study (Section 6.1), we found the ability to inspect intermediate states in a proof to be indispensable. The properties captured with `observe-field` in Figure 9 are complex and non-obvious. They would not have been discovered by Liquid Haskell's refinement inference without hints from the developer. We identified these important correctness properties because they helped prove intermediate goals of the COQ path compression proof, but they are important in any verification of this structure. Proving stability for some of these properties requires nontrivial inductions. We suspect that had we attempted this proof using an SMT-based implementation, we would have gotten stuck: SMT solvers give limited intermediate feedback that would have helped us realize these properties were necessary, and Liquid Haskell reports only that it failed to prove one refinement implies another. Without this, we would have had to resort to axiomatizing specific data structure properties. This is dangerous: It was not obvious initially that path compression sometimes extends a path (this is not mentioned by Anderson and Woll), and it was only in trying to manually prove that the compression operations always shortened paths that we realized they sometimes did not. However, with the benefit of the COQ proof, we believe we could probably port it to Liquid Haskell.

Proofs in Liquid Haskell. Verification with Liquid Haskell was generally more pleasant than using the COQ DSL. Our Liquid Haskell axiomatization allows programs that are typically only a few lines longer than totally unverified versions: After giving a small number of definitions for stating specifications, verifying the linked list in Section 6.2 required only one small (one-line) proof witness and a few calls to `liquidAssume` to give hints to apply definitions. Once we specified the lock-free linked list in Section 6.2 and gave the general axiom for terminal values, we required only a few runs of the verifier to determine the correct use of `liquidAssume` to guide the proof. We later strengthened the specification for two-state invariants on the list and simply re-ran Liquid Haskell to check the new specification; no further work was required. This trend continued in work on the lock-free set in Section 6.3. Once we recognized the need for slack variables and formulated a suitable `downcast` primitive, the verification was fairly straightforward.

It remains to be seen if this ease of use persists on larger examples—the *specification* of the rely/guarantee of the union find structure is longer than the `delete` *implementation* in Figure 11. Currently, when a Liquid Haskell verification fails, the resulting error message is a dump of the variables generated by inference as exported to the underlying SMT solver. Liquid Haskell also generates an HTML rendering of the source code with the results of type inference presented when hovering over source elements, but this exposes the type variables introduced by liquid type inference, and understanding an error often requires reconstructing the reasoning of the inference engine and manually relating liquid type variables with predicates. This may become much more difficult for specifications as complex as the union-find data structure.

The loss of transitive heap access in the Liquid Haskell formulation was not as severe as we originally feared. As we show in Section 6.3, the presence of a rich form of inductive data type makes recovery of some “deeper” specifications possible. Using idioms like `terminalValue`, we could give the specification for the lock-free union find implementation in Liquid Haskell. When the `Ord` type class used in the set implementation enforces a partial order (which is possible to enforce by extending the specification of the `Ord` type class in Liquid Haskell), the `Set` specification enforces acyclicity by inducing a total order on nodes of the data structure. However, in the absence of ghost/auxiliary state (logical state that is not represented at runtime), some transitive specifications are more difficult to enforce. For example, acyclicity for a Michael-Scott queue’s enqueue operation could be enforced by requiring links enqueued to the tail to initially have null (`None`) successor pointers. But that this implies an acyclic structure is less direct than the local ordering in our set implementation. But in exchange for a mostly automated discharge of proof obligations, this seems a reasonable tradeoff.

8. RELATED WORK

RGREFS target a very different part of the verification design space than most prior work, because we aim to prove rich logical invariants over data structures but cannot (without further extension [Gordon 2014]) prove full correctness. The majority of prior work falls into either simpler type systems for generic properties (such as data race freedom) or much more powerful concurrent program logics, with correspondingly greater complexity in exchange for proving stronger properties than RGREFS. Here we focus on two major streams of related work: concurrent program logics due to the similarities in reasoning principles (namely, rely-guarantee style reasoning) and the use of dependent types in imperative languages due to similarities in specific technical machinery. We also briefly discuss the relative proof burden of RGREFS compared to other approaches.

Verifying Concurrent Programs. A wide range of techniques for reasoning about concurrent programs exist, but we focus on logics: program logics and type systems. The fundamental issue in proving an assertion in a shared-memory concurrent program is tolerating or preventing possible interference from other threads. Since Owicki and Gries' original approach based on checking if actions of one thread invalidated any step in other threads' proofs [Owicki and Gries 1976], the major developments have been rely-guarantee reasoning [Jones 1983] and concurrent separation logic [Brookes 2004], the latter extending separation logic [O'Hearn et al. 2001] to verify threads with memory exchange using locks. Because these two approaches have complementary modularity benefits, the past few years have seen a flurry of work on combining them, starting with RGSep [Vafeiadis and Parkinson 2007], which uses concurrent separation logic with a shared region whose rely, and guarantee relations are given using separation logic assertions to describe pre- and post-states. Subsequent work extended the notion of separation to relations, allowing the rely and guarantee to be split and joined [Feng 2009] or used with unstructured parallelism [Dodds et al. 2009]. All rely-guarantee-based approaches permit some form of asymmetric protocol (including RGREFS).

Most recently state transition systems (STSs) have emerged as a popular means of verifying safety properties (single-state invariants and functional correctness) of FCDs [Dinsdale-Young et al. 2010; Svendsen et al. 2013; Svendsen and Birkedal 2014; Turon et al. 2013; Nanevski et al. 2014; Sergey et al. 2015a; Jung et al. 2015]. Concurrent abstract predicates (CAP) introduced the use of capabilities (sometimes referred to as tokens) in conjunction with abstract predicates to specify the protocols for updating concurrent data structures. Later work refined these ideas to support higher-order and impredicative protocols [Svendsen et al. 2013; Svendsen and Birkedal 2014], express a subset of protocols succinctly based on state transition systems [Turon et al. 2013], build and exploit rich notions of STS composition [Nanevski et al. 2014; Sergey et al. 2015a] (further enhanced by exploiting explicit notions of subjective auxiliary state [Ley-Wild and Nanevski 2013]), or build up to STSes from more primitive notions [Jung et al. 2015]. VeriFast [Smans et al. 2014; Jacobs and Piessens 2011] incorporates many of the ideas from CAP and related systems in an *automated* prover which we discuss below (Section 8.1). Each of systems enumerate a set of capabilities for moving nodes of a data structure through a finite state machine protocol, where each state specifies an invariant for the structure. The capabilities may be stored (in ghost state) and therefore transferred via shared-memory communication. In each of these systems, soundness is proven in part by using the set of capabilities held by a thread to induce a guarantee (to use the capabilities) and a rely (assuming other threads may use the complement of the thread's capabilities), which together induce some notion of stability.

All of the work above can verify full pre- and postconditions for concurrent data structures. We have previously demonstrated a way to prove triples for RGREF programs [Gordon 2014] by synthesizing an abstract trace and simplifying it through a series of rewrite rules reminiscent of Elmas et al.'s refinement for atomicity [Elmas et al. 2009, 2010]. So RGREFS form an adequate basis for proving functional correctness, but our focus in this article is on the expressivity of the stabilized reference refinements on their own—many extensions are possible (and discussed below) to improve expressive power.

In the STS-based program logics above, there is an interesting modularity limitation. While the logics are all equipped with means to compose STSes and/or frame away STSes that are not locally relevant, the granularity of composition or framing is limited to whole STSes. There is no way to extend a given STS with additional states and tokens, and no way to restrict an STS to a subset of its states or tokens. At most, they permit exchanging resources between STSes. Relation-based approaches (including concurrent RGREFS but also systems with explicit rely and guarantee relations [Jones 1983; Vafeiadis and Parkinson 2007; Feng 2009]) specify protocols and interference semantically rather than structurally and therefore do not suffer from this limitation. Understanding the precise relationship between relational (as in RGREFS or VCC [Cohen et al. 2009, 2010]) and STS approaches is a known open research problem [Jung et al. 2015].

Another challenge for modern separation logic approaches is allowing flexibility in overlapping views of shared state. Most modern logics for concurrency interpret (STS) protocols over named regions, and each heap fragment belongs to exactly one region (which may be strictly nested within another region). Logics in this space permit transfer among regions, but overlapping region specifications (without nesting) are not permitted. The sole exception is CoLoSL [Raad et al. 2015], which allows a thread to reason about a *subregion* of a shared region in isolation soundly, and different subregions may overlap arbitrarily. This can simplify the specification of algorithms such as graph algorithms. FCSL [Sergey et al. 2015a] also permits a form of this, with a structured mechanism to construct larger transition systems out of smaller transition systems and a framing rule on those transition systems without explicitly naming regions. RGREFS by contrast, permit arbitrary overlap of specification regions, ensured sound by the combination of the global aliasing agreement invariant (a reference's guarantee implies all aliases' relies) and containment (a reference's rely includes the rely of any reachable reference)—so one reference's guarantee permits only operations that are assumed possible by any aliases and references into the heap reachable from that reference. With the exception of Raad et al.'s work on CoLoSL, there is no focused exploration of the improvements this approach offers; further exploration seems worthwhile.

Some recent work has also focused on proving, within a program logic, that one program contextually refines another (e.g., a FCD is an observational refinement of a coarse-grained one) [Turon et al. 2013] is linearizable [Herlihy 1991] (roughly, that the operation's effects appear instantaneously to other clients of a data structure, though the definition is actually a more subtle condition on reorganizing interleaved traces) or satisfies related forms abstraction over interleaved thread histories [da Rocha Pinto et al. 2014; Sergey et al. 2015b]. We expect that future work can adapt ideas from proving linearizability in a traditional rely-guarantee system [Vafeiadis et al. 2006; Liang and Feng 2013] and work proving functional correctness atop RGREFS [Gordon 2014] to prove linearizability using RGREFS.

RGREFS currently cannot relate reference-centric specifications to other granularities of reasoning (e.g., region-based rely-guarantee [Feng 2009]) or represent asymmetry of knowledge (e.g., that a lock is held by one thread but not by others). This makes it difficult to relate structures that have no references between them, such as connecting the auxiliary structures used in lock-free algorithms with helping (e.g., an elimination stack [Hendler et al. 2004], where one thread may *help* complete the operation of some other thread's concurrent operation). These are not fundamental limitations but are out of scope for our focused exploration of the capabilities of minimal concurrent RGREF systems. Support for helping should be possible through the addition of linear capabilities and regions to RGREFS, as exist in some concurrent separation logics—particularly the STS logics—for exactly this purpose [Dinsdale-Young et al. 2010; Svendsen et al. 2013; Svendsen and Birkedal 2014]. The rely and guarantee specifications would need to be extended to interact with the capabilities in a style similar to CAP [Dinsdale-Young et al. 2010] or its many descendants. Structuring a full program logic with explicit regions over a base language with RGREFS would then permit expressing heap properties of other granularities, where the presence of RGREFS may specify some region transition specifications. RGREFS cannot reason about the combination of *multiple* writes satisfying a guarantee; this could be addressed by using results from a program logic over RGREFS to prove a non-atomic guarantee satisfaction (e.g., on thread-local state guarded by a lock). Also important, as references are duplicated, the guarantees possessed by the whole program for a given heap cell become weaker (permit fewer actions); thus it is not possible to temporarily share a unique reference to parallelize some work and later recover a reference with the original, stronger guarantee. Some form of fractional permissions [Boyland 2003] and/or the *recovery* [Gordon et al. 2012] variant of borrowing could enable this. We plan to explore these extensions in future work.

Another piece of recent work done in parallel with ours is Kloos et al.'s asynchronous liquid separation (ALS) types [Kloos et al. 2015]. They extend the OCaml implementation of Liquid Types [Rondon et al. 2008] to reason about concurrency in a cooperative task-based threading model. Specifically, they add linear analogues of points-to facts from separation logic, where the assertions give a subset—a refinement—of the type a reference points to and permit strong updates to the choice of refinement. They use this for a series of case studies, culminating in finding a bug in the MirageOS [Madhavapeddy et al. 2013] filesystem implementation (written in OCaml). Their design is well suited to the form of cooperative concurrency in MirageOS but is restricted to linear ownership transfer through task creation. RGrefs permit flexible aliasing, but our Liquid Haskell embedding lacks proper substructural reasoning, making the two incomparable. ALS also permits linear transfer of refinements *without* data flow, which RGrefs do not support. Both systems build on Liquid Types, which is both strong evidence of the generality of Liquid Types and that both ALS and RGrefs support effective automation.

In parallel with our earlier work on RGrefs, Militão et al. developed rely-guarantee *protocols* [Militão et al. 2014]—transition systems built on linear capabilities, where a protocol may be split into two compatible protocols. Each transition of the protocol is associated with an update to a single heap cell, but the protocol may exchange capabilities to other heap cells, permitting protocols over flexible portions of the heap. More recently, in parallel with our concurrent extension of RGrefs, they also extended rely-guarantee protocols for lock-based concurrency [Militão et al. 2016] and abstract protocols. In both variants, compatibility between protocols over the same state is checked by exhaustive search of possible interleavings (essentially exhaustive model checking) rather than by logical implication. The protocols only express the identity of a tag on tagged values in the heap, so the expressive power is limited to reasoning about a (statically) fixed set of tags.

Dependent Types for Imperative Programming. Integrating dependent types and state is a long-standing challenge for program verification. One approach is allowing types to depend on immutable data [Freeman and Pfenning 1991], often using a decidable theory of refinements [Rondon et al. 2008]. More recent and powerful approaches include Hoare Type Theory (HTT) [Nanevski et al. 2008, 2010] as implemented in several Coq embeddings (both axiomatic like ours [Nanevski et al. 2008; Chlipala et al. 2009] and foundational [Nanevski et al. 2010; Chlipala 2011]) and the Dijkstra Monad, as implemented in F⋆ [Swamy et al. 2013].

HTT is essentially a monadic embedding of Hoare logic and separation logic in CIC (though this understates its elegance). FCSL [Nanevski et al. 2014; Sergey et al. 2015a], discussed earlier, is a program logic similar in flavor to HTT, also implemented foundationally in Coq, giving the specification language access to all of Coq's dependently typed term language for specification. Unlike HTT, it lacks the ability to store higher-order procedures in the heap. For its foundational soundness proof, FCSL terms are given a denotation in the style of Brookes' trace semantics [Brookes 2004], represented as an inductive type of partial traces in Coq.

The Dijkstra Monad has a similar flavor but focuses on Dijkstra's predicate transformers [Dijkstra 1975]. HTT has considered concurrency in the form of transactional memory [Nanevski et al. 2009] but does not handle fine-grained concurrency and restricts itself to single-state invariants over transactionally accessed data. F⋆ has not yet explored concurrency. Older variants of F⋆ (which was recently rebuilt from scratch [Swamy et al. 2016]) included a variant of the Dijkstra Monad that enforces that all heap updates must satisfy a two-state invariant on heaps (as a binary relation) and included axioms for relating the current heap via the heap relation to a previous heap. This mechanism is similar in flavor to (sequential) rely-guarantee references but coarser in both relation granularity and the inability to separate multiple roles in a protocol.

RGREFS integrate dependent refinement types into an imperative language by allowing refinement types to apply to heap fragments, subject to a semantic check (stability) that the type is sensible and no operations will arbitrarily change the meaning of the type. This notion of stability is based on traditional rely-guarantee reasoning [Jones 1983], so support for concurrency carries over naturally. RGREFS can specify fine-grained asymmetric protocols (as can FCSL but not F⋆) and (by weakening references) support enforcing progressively stronger invariants over time, particularly when sharing previously thread-local data for the first time. As with the program logics discussed above, HTT, F⋆, and FCSL prove full Hoare triples—functional correctness—while RGREFS require some extension to prove full correctness [Gordon 2014].

8.1. Verification Burden

Throughout we have claimed RGREFS target an intermediate point in the verification space between generic concurrency type systems for properties like race freedom [Flanagan and Freund 2000; Gordon et al. 2012] or deadlock freedom [Flanagan and Abadi 1999] and the concurrent program logics discussed above. It should be clear that the expressiveness is clearly an intermediate between the two groups. But we have thus far left implicit that the specification and verification burden is similarly intermediate, beyond implying that basing RGREFS on refinement types (which have been shown to be effectively inferable and automatable [Rondon et al. 2008, 2010; Vazou et al. 2013, 2014b, 2015]) and reference capabilities for mutability control [Tschantz and Ernst 2005; Zibin et al. 2007, 2010] (which have shown evidence of developer preference and usability at scale [Gordon et al. 2012] as well as suitability for inference [Huang et al. 2012]) should produce an intermediate burden on developers. Strong evidence for this requires more experience, but we offer preliminary evidence here based on comparing specification and (manual) proof burden for implementation of analogous structures.

8.1.1. FCSL. Sergey et al. [2015a] give specification burden for FCSL versions of several lock-free data structures we also verified here. FCSL is a full program logic for functional correctness and also used to verify a form of abstract atomicity [Sergey et al. 2015b] of these structures, so we would expect a larger specification and verification burden for FCSL than RGREFS.

The FCSL version of the Treiber stack requires 980 lines of COQ to specify on top of the FCSL implementation [Sergey et al. 2015a], while we require only 105 lines to specify an RGREF version and discharge the resulting proof obligations. Their proofs ensure abstract atomicity and actual completion of operations (i.e., if push returns, the element was actually pushed onto the stack). As described in Section 3.2.2, the RGREF implementation proves only that the stack's invariants are respected (both one- and two-state invariants ensuring only pops and pushes occur). A significant portion of the FCSL proof and specification work is devoted to setting up concurroids [Nanevski et al. 2014], the core specification primitive of FCSL. These primitives are very expressive but also significantly more structured than simple binary relations.

They also build a producer-consumer example on top of their Treiber stack, similar to the extension we give in Section 3.2.2 to highlight asymmetric protocols. This required an additional 608 lines of Coq to specify the extension in FCSL [Sergey et al. 2015a], 243 of which (according to Sergey et al. [2015a]) are client code and client-side proofs. The remainder of their code—365 lines—specifies auxiliary state and proves assertion stability for that state (mostly lemmas relating stack states and histories).

We required only 38 additional lines of code (with no client code) and 66 additional lines of proof, though this includes *duplicating* the push and pop operations to verify that they satisfy more restricted types (Section 3.2.2) and proving that these more restricted forms could be used anywhere the originals were. We could have instead used the restricted versions (e.g., with push prohibited from popping) for the base Treiber stack, in which case the overall Treiber stack would have required 38 + 18 = 56 (adding the original `Node` and `deltaTS` definitions to the producer-consumer implementation) lines of code and 66 + 20 = 86 lines of proof (adding stability for `deltaTS`, etc.).

The FCSL client code sets up exactly two threads (one producer and one consumer), with each thread keeping track of the set of items it has successfully pushed (or popped) using a thread-local array, ensuring these assertions are stable and that after termination each thread has pushed (popped) all elements of a predetermined set. This relies on the use of the explicit history monoid used by the stack specification, which permits ensuring that any element popped from the stack was previously pushed. We could write RGref client code to perform the same actions as the FCSL client code and would be able to prove the producer at most removed elements from its local queue and pushed elements on the stack (similarly for the consumer). RGrefs could not prove that either thread *actually* performed its work or that the final set of elements in the consumer's local structure matched the original set of elements in the producer's local structure. This is partly due to lack of ghost state (notably, histories) in RGrefs and partly due to the fact that we lack a way to share any refinements to argument predicates that occur inside an operation (such as push) with the operation's caller. We describe this issue in more detail in relation to VeriFast below.

Both FCSL and RGrefs are aimed at shallow embedding into Coq, so our Coq implementation seems to satisfy our goals of an intermediate proof burden for proving properties of intermediate strength, at least with respect to FCSL.

8.1.2. VeriFast. VeriFast is a mature system for automated (SMT-based) verification of concurrent programs using separation logic, applicable to C and Java [Jacobs and Piessens 2011]. VeriFast includes a *shared boxes* extension [Smans et al. 2014] to separation logic, which are roughly named regions with a fixed two-state invariant over the region contents, with permissible actions on the region explicitly enumerated (in a way, similar to the role individual constructors of rely and guarantee relations provide, though reflexivity is implicit). Shared boxes have been used to implement a lock-free monotonic counter and a sequential stack. As with FCSL, these verifications prove full functional correctness (i.e., that each operation actually *does* work, the correct number of times), but, unlike FCSL, no form of atomicity (abstract or otherwise) is proven (indeed, the stack is sequential, and would not satisfy such a specification).

The C implementation of the monotonic counter[27] using shared boxes requires (ignoring the `main` function) 27 lines of specification to declare the shared box governing the counter, annotate the increment operation, and coerce an integer pointer into the appropriate shared box. For roughly the same functionality, our Coq implementation

[27]Here we manually count the lines in the economically typeset version [Smans et al. 2014], but a more readily accessible version exists: https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/incr_box/incr_box.c.html.

requires 12 lines total (for specification, proof hints, and implementation), and our Liquid Haskell implementation requires only the 2 lines of specification given in Section 5.2.2: the types for the allocation and increment functions.

VeriFast's numbers include the use of *handle predicates*, which play a somewhat similar role to refiners in RGREFS and F*'s older primitive for witnessing relations between successive heaps—the read code makes two reads, and handle predicates are used to statically prove the result of the second read is greater than or equal to the result of the first. This is not explicitly proven in the RGREF code of Figure 2. The analogous result in our system would be to read an atomic counter twice using refiners to embed the relationship in the counter's type:

```
x <- mkCounter ();
observe x as v1 in (λ n h => n >= v1);
observe x as v2 in (λ n h => v1 <= v2 <= n);
(* Γ = ..., x : ref{ℕ | λn h. v1 <= v2 <= n}[increasing, increasing] *)
assert (v1 <= v2)
```

RGREFS cannot statically verify the final assertion, only accumulate equivalent invariants in the counter's type. Because the concurrent RGREFS presented here are a minimalist system, the lower bound on the counter's value does not escape the reference itself—the current system lacks a supporting program logic, and the only form of refinement is attached to references.

The sequential stack implementation[28] requires 174 lines of annotation (it does not use shared boxes, as it is sequential). This is single-threaded C, and some portion of the overhead is due to handling memory deallocation as well as size and empty-check operations that we do not implement. The implementation also contains a (memory leaking) Treiber stack,[29] which requires 308 lines of code.[30] The VeriFast distribution also includes a Treiber stack using hazard pointers [Michael 2004] to reclaim memory, with a correspondingly higher proof burden for the more complex structure.

Our lock-free Treiber stack implemented in the COQ DSL requires only 63 lines of proof, for proofs that are predominantly manual, about 42 lines of specification and code. The producer-consumer extension to use more precise specifications for pushing and popping adds another 104 lines of code and proof, but this includes proving that complete reimplementation of the push and pop operations on more restricted rely- and guarantee-relations are type-correct for more precise specifications—so much of that is actually performing more precise versions of proofs from the first definition, making the actual specification and proof burden for the producer-consumer lower: one hundred forty-two lines of code and proof for a self-contained version.

The VeriFast implementations prove that the push and pop operations actually perform the appropriate action, as opposed to only ensuring the operations preserve certain one- and two-state invariants. The VeriFast implementations are parameterized by ghost code to update auxiliary state on behalf of clients [Jacobs and Piessens 2011], which permits clients to choose and manage their own auxiliary state, such as partial views of data structure contents and moving permissions between the client and data structure. RGREFS lack auxiliary state but have refiners, which can be used for some (not all) of the same use cases (notably, to relate previous and current values).

[28]Taken from https://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/stack.c.html, but our counts omit the usage examples and cover only the definitions and proofs for the structure and operations themselves.

[29]`examples/shared_boxes/stack_hp/stack_leaking.c`, which despite the directory name is a hazard-pointer-free implementation in the same directory as the hazard pointer version described in a article [Smans et al. 2014].

[30]We manually extracted annotation lines for the structure's specification, and the annotations/lemmas for allocation, concurrent push, and concurrent pop.

However, RGREFS do not have a way to reflect use of a refiner inside a routine back to callers. In our COQ implementation, we could leverage the ambient logic to parameterize over any rely strong enough to prove local operations (i.e., a rely that implies `deltaTS`) and in principle could modify the language to indicate how refinements on the stack may have been updated inside the operations. Then using a form of explicit stabilization [Wickerson et al. 2010] (i.e., refine the references by adding predicates that are the strongest property implied by actions internal to the operation, that are also preserved by the rely), the choice of a stronger rely and the ways this strengthens post-operation knowledge of the stack state could be "exported" to callers.

$$\begin{gathered}\forall R.\ R \subseteq \mathsf{deltaTS}.\ \forall G.\ \mathsf{deltaTS} \subseteq G.\{s : \mathsf{ref}\{\mathsf{nodeopt} \mid P\}[R, \mathsf{deltaTS}], n : \mathsf{nat}\}\\ \mathit{push\_ts}\ s\ n\\ \{s : \mathsf{ref}\{\mathsf{nodeopt} \mid P \cap \lceil \mathsf{headIs}\ n \rceil_R\}[R, \mathsf{deltaTS}], n : \mathsf{nat}\}.\end{gathered}$$

The theory to permit such a specification remains future work, but the expressiveness gap does not appear insurmountable. The relative tradeoffs in expressiveness compared to VeriFast's approach are not immediately obvious, though VeriFast's technique appears more general.

The difference in language abstraction level, operations provided, and strength of properties proven makes this comparison a bit unfair (in RGREFS' favor), but this still suggests RGREFS are imposing specification and proof burden in line with the desired intermediate goal.

9. CONCLUSIONS AND FUTURE WORK

We have shown how to soundly enable refinement types constraining mutable data in a shared-memory concurrent setting and shown the efficacy of the approach with case studies verifying state-based and change-based (two-state) invariants for a number of lock-free data structures, using two implementations. RGREFS' expressiveness lies at an intermediate point between classic type systems for safe concurrency and modern concurrent program logics, proving useful invariants with correspondingly intermediate specification (and proof) burden. This is also the first approach to verifying invariants of concurrent data structures that can prove these invariants in the context of an otherwise-unverified program. Our COQ implementation demonstrates the technique's expressiveness, while our Liquid Haskell implementation shows that very slight restrictions allow effective integration with existing automated refinement type implementations; the technique is usable now. In addition to these, we have performed the first formal machine-checked proof of invariants for a lock free union-find implementation.

Looking forward, there are some natural extensions to this work worth exploring. One example is the addition of linear capabilities to the system to reason about certain types of interference being enabled or disabled over time; related systems have used this idiom to great effect [Dinsdale-Young et al. 2010; Svendsen et al. 2013; Turon et al. 2013; Jung et al. 2015], and it could be further combined with logical transfer of refinements separate from dataflow [Kloos et al. 2015] for additional flexibility. Another example is the integration with explicit stabilization [Wickerson et al. 2010] outlined in Section 8's comparison with VeriFast. A more substantial extension is building a full program logic atop concurrent RGREFS as a way to verify functional correctness and linearizability. We have early results on the former [Gordon 2014], but proving pre- and post-conditions without reasoning about the granularity of thread interference is only part of the full correctness criteria for FCDs. Rely-guarantee reasoning has already been exploited for proving linearizability [Vafeiadis et al. 2006; Liang and Feng 2013], and integrating these ideas with RGREFS seems a natural next step.

APPENDIXES

A. SOUNDNESS FOR CONCURRENT RGREFS

We have proven soundness for our type system. At a high level, the proof is decomposed into two steps. First, we rely on a repetition of sequential soundness [Gordon et al. 2013] of the pure fragment (for the simpler pure fragment here) to ensure soundness within a given thread. Second, we prove soundness for our richer imperative fragment. This is a non-trivial extension because it binds variables from the imperative context in each others' types, which was not supported in the original system; `observe-field` could not be given a proper type in the previous core language. Third, we compose soundness proofs of threads by synthesizing classic rely-guarantee reasoning among threads [Jones 1983] from the rely relations embedded in types in the typing context for each thread. The latter steps are handled by our embedding into the Views Framework [Dinsdale-Young et al. 2013], detailed in the remainder of this section.

The Views Framework abstracts core concepts of concurrent type systems and program logics into an *abstract* program logic framework, with the goal of providing a reusable core for proving soundness of proof systems for concurrent programs. Instantiating a handful of parameters for atomic actions and axioms about instrumented and concrete program state spaces yields a sound semantic basis for reasoning about safety properties of concurrent programs in the Views Framework's semantic logic—using subsets of the state space as assertions. Proving that a given source language judgment (e.g., a typing derivation [Gordon et al. 2012] or program logic judgment) can always be embedded to a valid Views derivation then implies soundness for the original source judgment (up to the embedding preserving the intended meaning of the source judgment—that not all assertions embed to True). In our case, type environments serve as assertions, and flow-sensitive type judgments embed to Hoare triples in the Views logic. Our embedding of type environments requires that the predicates of any reference hold for the referenced heap segment.

In the richest instantiation, a *view* represents a thread-local assumption about the global program state and is *stable* with respect to an *interference relation*.[31] *Compatibility* is ensured by making the view a partial commutative monoid, so, for example, if one thread's view contains capabilities that are not modeled by the interference relation on another view, the composition of those views is undefined.

While designed for first-order languages and subsequently extended for higher-order specifications [Svendsen et al. 2013] (but not higher-order store), we side step some higher-order issues by extending the Views Framework with commands to interpret small-step call-by-value semantics for an extension of CIC and using syntactic λ-terms as components of the Views model $\mathcal{M}$. The Views Framework interprets loops and conditionals as non-deterministic loops and branches, respectively, so we actually embed into a language with more relaxed semantics than the core language. Because our typing rules are not sensitive to branch and loop conditions in the imperative fragment, this relaxation is not problematic (dependent elimination in the pure fragment is preserved by our embedding). We restrict the imperative fragment to first-order for simpler presentation, but the RGREF-specific proof approaches used here are orthogonal to the Views extensions used to enable higher-order imperative contexts; they should be straightforward (though tedious) to combine.

Following the parameter scheme from the main Views article [Dinsdale-Young et al. 2013], we use an interference-stabilized separation algebra for our view and instantiate atomic commands and axioms as one might expect relative to that state. In particular,

[31]The terminology is drawn directly from rely-guarantee reasoning [Jones 1983], and the latter can be proved sound by embedding into the Views framework.

$$\begin{array}{rrcl rrcl} \text{Locations} & \widehat{Loc} & ::= & \ell_{A,P,R,G} & \text{Terms} & \widehat{M}, \widehat{N} & \in & \text{source terms} \mid \widehat{Loc} \\ \text{Heaps} & \widehat{H} & \in & Loc \rightharpoonup \widehat{M} & \text{Store} & \widehat{S} & \in & Var \rightharpoonup \widehat{M} \end{array}$$

$$L\lfloor \ell_{A,P,R,G} \rfloor \overset{\text{def}}{=} \ell \quad \text{and lifted to terms, heaps, etc.}$$

$$\mathcal{M} \overset{\text{def}}{=} \{m \in \text{Heap} \times \text{Heap Type} \times \text{Env} \times \text{Env Type} \times \text{Env Type} \mid \mathsf{Valid}_{\mathcal{M}}(m)\}$$

$$\frac{\begin{array}{c} \vdash \Sigma \quad \vdash H : \Sigma \quad \vdash \Gamma \quad \forall x \in \Delta.\, \Gamma \vdash \Delta(x) \quad \vdash S : \Gamma, \Delta \quad \forall \ell_{A,P,R,G} \in \mathsf{cod}(H + S).\, P\ H(\ell)\ H \wedge \mathsf{stable}\ P\ R \\ \forall \ell_{A,P,R,G} \in \mathsf{cod}(H + S) \forall \ell'_{A,P',R',G'} \in \mathsf{cod}(H + S).\, \ell \equiv \ell' \Rightarrow ((G' \Rightarrow R) \wedge (G \Rightarrow R') \vee (\ell == \ell')) \\ \forall \ell_{A,P,R,G} \in \mathsf{cod}(H + S).\, \Sigma; \emptyset; \epsilon \vdash \mathsf{ref}\{A \mid P\}[R, G] \end{array}}{\mathsf{Valid}_{\mathcal{M}}(H, \Sigma, S, \Gamma, \Delta)}$$

$$(H, \Sigma, S, \Gamma, \Delta) \bullet (H', \Sigma', S', \Gamma', \Delta') \overset{\text{def}}{=} (H \cup H', \Sigma \cup \Sigma', S \uplus S', \Gamma \cup \Gamma', \Delta \uplus \Delta') \text{ where } \mathsf{Valid}_{\mathcal{M}} \quad \text{otherwise } \bot$$

$$m \in P * Q \iff \exists m'.\, \exists m''.\, m' \in P \wedge m'' \in Q \wedge m' \bullet m'' = m$$

$$(H, \Sigma, S, \Gamma, \Delta) \mathcal{R} (H', \Sigma', S', \Gamma', \Delta') \overset{\text{def}}{=} \Sigma \subseteq \Sigma' \wedge (S, \Gamma, \Delta) = (S', \Gamma', \Delta') \wedge H\ \mathcal{R}^{S}_{H}\ H'$$

$$H\ \mathcal{R}^{S, x \mapsto v}_{H}\ H' \overset{\text{def}}{=} \left(\bigcap_{\ell_{A,P,R,G} \in v} H \downarrow_{\ell} = H' \downarrow_{\ell} \vee R\ H(\ell)\ H'(\ell)\ H \downarrow_{\ell}\ \ H' \downarrow_{\ell} \right) \cap (H\ \mathcal{R}^{S}_{H}\ H')$$

$$H\ \mathcal{R}^{\epsilon}_{H}\ H' \overset{\text{def}}{=} \mathsf{True} \qquad \mathcal{S} \overset{\text{def}}{=} \text{Heap} \times \text{Env} \qquad \lfloor (H, \Sigma, S, \Gamma) \rfloor : \mathcal{M} \to \mathcal{S} \overset{\text{def}}{=} (L\lfloor H \rfloor, L\lfloor S \rfloor)$$

Fig. 14. Views state space. "Hats" (e.g., $\widehat{M}$) indicate explicit annotations of RGREF components on locations, while the erased ($L\lfloor - \rfloor$) omits this.

our views take the form $\{p \in \mathcal{P}(\mathcal{M}) \mid \forall m \in p.\ \mathcal{R}(m) \subseteq p\}$[32] with (type-)instrumented states $\mathcal{M}$, interference relation $\mathcal{R}$ (which calculates a global interference relation from the references in context), the join operator $*$ on views defined as in Figure 14, and the collection of component-wise empty maps as the unit view. This selects the subsets of instrumented states which are closed under transitions in the interference relation. That figure also defines concrete execution states $\mathcal{S}$, an erasure $\lfloor - \rfloor$ from views ($\mathcal{M}$) to concrete states, and an interpretation $[\![-]\!](-)$ of *atomic commands* (C) on runtime states $\mathcal{S}$. Figure 15 defines $[\![-]\!](-)$ on Views $\mathcal{M}$ for clarity and indicating how view state is updated to match actual execution. This is used not only to define the dynamic semantics of atomic commands but also the specification of updates to the non-concrete portions of the view state necessary for proving soundness. Figure 16 describes how we instantiate the remaining Views parameters.

Our view states $\mathcal{M}$ are the well-formed ($\mathsf{Valid}_{\mathcal{M}}$) subset of 5-tuples of heap and stack, with heap and stack typings. The validity check enforces basic well-typing of heap and stack components (including enforcing that they contain no dereference expressions), compatibility between references, and that all refinements are true.

We instantiate the framework's atomic statements to the statements of the imperative fragment, modulo an embedding $\downarrow - \downarrow$ on commands that decomposes loops and conditionals into the Views Framework's non-deterministic forms. Every command containing a pure term M steps by fully (call-by-value) reducing the term. This respects correct thread interleaving semantics because reduction of the pure fragment may not access the heap.

We must also prove the axioms respect execution. Axiom soundness intuitively states that the axioms for atomic commands (the embedding of Figure 4's imperative typing

[32]In CIC/COQ, intuitively $\{p : \mathcal{M} \to \mathsf{Prop} \mid \forall xy.,\ p\ x \wedge \mathcal{R}\ x\ y \to\ p\ y\}$.

$$
\begin{array}{rcl}
[\![-]\!] & : & \mathsf{Atom} \to \mathcal{M} \to \mathcal{M} \\
[\![x := \mathsf{alloc}_{A,P,R,G}\ M]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H[\ell \mapsto S(M) \Downarrow_{cbv}, \Sigma[\ell \mapsto A], S[x \mapsto \ell_{A,P,R,G}], (\Gamma, ?), (\Delta, ?)) \\
 & & x : \mathsf{ref}\{A \mid P\}[R,G] \text{ placed by splitting, } \ell \text{ fresh} \\
[\![x :=!N]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H, \Sigma, S[x \mapsto H(N \Downarrow_{cbv})], (\Gamma, x : (\mathsf{fold}\ G\ A)), \Delta) \text{ where } \Gamma \vdash N : \mathsf{ref}\{A \mid P\}[R,G] \\
[\![[x] := N]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & \mathsf{let}\ \ell = S(x)\ \mathsf{in}\ (H[\ell \mapsto S(N) \Downarrow_{cbv}], \Sigma, S, \Gamma, \Delta) \text{ where } \mathsf{NoDerefs}(N) \\
[\![[x] := y]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & \mathsf{let}\ \ell = S(x)\ \mathsf{in}\ (H[\ell \mapsto S(N) \Downarrow_{cbv}], \Sigma, S, \Gamma, \Delta/y) \\
[\![x := \mathsf{interp}_\tau(M)]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H, \Sigma, S[x \mapsto S(M) \Downarrow_{cbv}], (\Gamma, x : \tau), \Delta) \\
[\![x := \mathsf{interp'}_\tau(M)]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H, \Sigma, S[x \mapsto S(M) \Downarrow_{cbv}], \Gamma, (\Delta, x : \tau)) \\
[\![x := \mathsf{CAS}(r, M, M')]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & \mathsf{let}\ \ell = S(r)\ \mathsf{in} \begin{cases} (H[\ell \mapsto S(M') \Downarrow_{cbv}], \Sigma, S[x \mapsto \mathsf{true}], (\Gamma, x : \mathbb{B}), \Delta) & \text{if } H(\ell) = S(M) \Downarrow_{cbv} \\ (H, \Sigma, S[x \mapsto \mathsf{false}], (\Gamma, x : \mathbb{B}), \Delta) & \text{otherwise} \end{cases} \\
[\![\mathcal{R}_{\mathbb{N}} Z\ x]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H, \Sigma, S, \Gamma', \Delta) \text{ if } H(S(x)) = Z, \Gamma' \text{ extends } \Gamma \text{ as in } \mathcal{R}_{\mathbb{N}} \text{ rule's } Z \text{ case} \\
[\![\mathcal{R}_{\mathbb{N}} S\ x\ n]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H, \Sigma, S[n \mapsto v], \Gamma', \Delta) \text{ if } H(S(x)) = S\ v, \Gamma' \text{ extends } \Gamma \text{ as in } \mathcal{R}_{\mathbb{N}} \text{ rule's } S \text{ case} \\
[\![\mathcal{R}_{\mathsf{ref}} x\ r]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & (H, \Sigma, S[r \mapsto v], \Gamma', \Delta) \text{ if } H(S(x)) = v, \Gamma' \text{ extends } \Gamma \text{ as in T-}\textsc{RefineRef}
\end{array}
$$

$$
\begin{array}{rcl}
[\![-,-]\!] & : & \mathsf{Env} \times \mathsf{Env} \to \mathcal{P}(\mathcal{M}) \\
[\![\epsilon,\epsilon]\!] & \stackrel{\text{def}}{=} & \{(\emptyset, \emptyset, [], \epsilon)\} \\
[\![(\Gamma, x : N), \epsilon]\!] & \stackrel{\text{def}}{=} & [\![\Gamma, \epsilon]\!] * \{m \in \mathcal{M} \mid m.\Sigma; m.H; m.\Gamma \vdash m.S(x) : S(N) \wedge m.\Sigma; m.H; m.\Gamma \vdash N : \mathsf{Prop} \wedge m.\Gamma(x) = N\} \\
[\![\Gamma, (\Delta, x : N)]\!] & \stackrel{\text{def}}{=} & [\![\Gamma, \Delta]\!] * \{m \in \mathcal{M} \mid m.\Sigma; m.H; m.\Gamma \vdash m.S(x) : S(N) \wedge m.\Sigma; m.H; m.\Gamma \vdash N : \mathsf{Prop} \wedge m.\Delta(x) = N\}
\end{array}
$$

Fig. 15. Embedding into the Views Framework. We lift stack lookup ($S(-)$) to terms as well, substituting stack contents for variables.

Parameter A. (Atomic Commands) The atomic commands as in Figure 4. The compositional commands are simply inherited from the Views Framework language.
Parameter B. (Machine States) $\mathcal{S}$ as in Figure 14.
Parameter C. (Interpreting Atomic Commands) $[\![-]\!](-)$ as in 15, projected to its actions on $\mathcal{S}$ rather than $\mathcal{M}$.
Parameter D. (Views Commutative Semigroup) Instantiated as the stabilized View monoid induced by $\mathcal{M}$ and $\mathcal{R}$ in Figure 14.
Parameter E. (Axiomatization) Embedding of the typing rules for atomic commands from Figure 4, using the environment embedding from Figure 15.
Parameter H. (Separation Algebra) $(\mathcal{M}, *, \mathsf{emp})$ as in Figure 14.
Parameter I. (Separation Algebra Reification) Erasure $\lfloor - \rfloor$.
Parameter K. (Interference Relation) $\mathcal{R}$ as in Figure 14.
Parameter L. (Axiom Soundness III) Lemma A.1, which then induces Parameters G and J.
We require no instantiation of Parameters M–Q, as we require no equivalent of conjunction or disjunction of assertions, and use only the standard frame rule.

Fig. 16. Views Framework parameter instantiation, according to Dinsdale-Young et al. [2013].

rules for atomic actions) on views ($\mathcal{R}$-stabilized $\mathcal{M}$s) coincide with the actual semantics of the commands on concrete runtime states $\mathcal{S}$: that any time a command is executed in a runtime state consistent with the erasure of its axiom's precondition view, the resulting state is consistent with the erasure of the same axiom's postcondition view. We give a stylized proof by defining our atomic action semantics on $\mathcal{M}$ instead of $\mathcal{S}$ in Figure 15. The basic interpretation can be obtained by paying attention to only the H and S components of $\mathcal{M}$, with the other components essentially demonstrating how the postcondition views would need to be modified to be consistent with the new concrete states. For each atomic command, lifting the definition to a set of valid $\mathcal{R}$-stable $\mathcal{M}$s (views), the output $\mathcal{M}$ will also be a set of valid $\mathcal{R}$-stable $\mathcal{M}$s.

LEMMA A.1 (AXIOM SOUNDNESS). *For all $\Gamma, \Delta, \Gamma', \Delta'$, and atomic command a, if $\Gamma; \Delta \vdash a \dashv \Gamma'; \Delta'$ (i.e., if $p \in [\![\Gamma, \Delta]\!]$ and $q \in [\![\Gamma', \Delta']\!]$ then $(p, a, q) \in \mathsf{Axiom}$), for all $m \in \mathcal{M}$, $[\![a]\!](\lfloor p * \{m\} \rfloor) \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$.*

Proof. By induction on the atomic command typing. Most cases are variations on writes, which depend on guarantee checks, global compatibility, and the calculated global interference for framed-out views.

—Interpret: Progress, preservation, and strong normalization hold of the pure fragment (see original RGref proof [Gordon et al. 2013]). Thus, the complete call-by-value reduction of a pure term terminates with the appropriate type. As in previous proofs, folding and restrictions on reads ensure that global compatibility invariants are preserved. The only change to the underlying state $\mathcal{S}$ is binding a new stack variable, which trivially composes correctly with the (unmodified) framed view. The only deviation from previous proofs is the presence of field reads, which are equivalent to field projection following a standard read.
—Write: By the global reference compatibility invariant and the fact that conversion can only weaken reference types, compatibility is preserved in the current view, and the update is also permitted (by obliviousness or compatibility) according to the calculated global rely for m. By compatibility and the proof of predicate preservation, the predicates are preserved for all view components.
—Allocation: This case is in some ways a simplification of the write case, only establishing the predicate.
—CAS: Similar to the write case. □

Lemma A.1 concerns only the soundness of atomic commands rather than the full language. As long as all typing derivations can be embedded to a valid Views derivation, full soundness for larger structures (loops, sequencing, conditionals, etc.) follows from Lemma A.1 and the Views Framework's soundness (for structural rules).

Theorem A.2 (RGref Soundness). *For all* Γ, Δ, C, Γ', *and* Δ'

$$\Gamma;\Delta \vdash C \dashv \Gamma';\Delta' \Longrightarrow \{[\![\Gamma,\Delta]\!]\} \vdash_{\downarrow} C \downarrow\dashv \{[\![\Gamma',\Delta']\!]\}.$$

Proof. By induction on the source derivation. The atomic command cases appeal to Lemma A.1, while the compound command cases proceed mostly by the inductive hypothesis and soundness of the underlying Views Framework Logic. Loops and conditionals become slightly more complex due to desugaring. The only complex cases are the refiners, which embed to nondeterministic choice of specialized commands for each constructor case, which get stuck if run with the wrong constructor (this is a nondeterministic model of execution semantics that would only run the correct branch). Reference refiners simply desugar to binding and the refiner body (i.e., $\downarrow \mathcal{R}_{\mathsf{ref}}(x, r \Rightarrow C) \downarrow = \mathcal{R}_{\mathsf{ref}}(x, r); C$).

(1) Heap writes: Follows from Lemma A.1.
(2) Loops, conditionals, sequencing, parallel composition: Straightforward by command embedding and the inductive hypothesis.
(3) Environment weakening: By the Views frame rule.
(4) Refiners: Each refiner embeds to possibly nondeterministic choice of constructor-specific refinement commands, each of which:
—Gets stuck if it tries to match the wrong constructor in the heap, and
—Binds the constructor components appropriately if the constructor matches.
Each constructor-specific command is sequentially followed by the body for that constructor case and embeds via the inductive hypothesis. □

B. SEMANTICS FOR CONVERT

Most primitives in the pure fragment are standard elements of CIC or modest extensions (e.g., references with no eliminator) with obvious semantics. The exception is

$$\frac{e \rightarrow e'}{\mathsf{convert}_\tau(e) \rightarrow \mathsf{convert}_\tau(e')} \qquad \frac{}{\mathsf{convert}_{\tau * \tau'}((e, e')) \rightarrow (\mathsf{convert}_\tau(e), \mathsf{convert}_{\tau'}(e'))}$$

$$\frac{\tau \in \{\top, \mathsf{unit}, \mathbb{N}, \mathbb{B}\}}{\mathsf{convert}_\tau(e) \rightarrow e} \qquad \frac{}{\mathsf{convert}_{\sigma \rightarrow \tau}((\lambda x : \sigma'.\, e)) \rightarrow (\lambda y : \sigma.\, (\lambda x : \sigma'.\, e)\ (\mathsf{convert}_{\sigma'}(y)))}$$

$$\frac{}{\mathsf{convert}_{\mathsf{ref}\{A|P'\}[R',G']}(\ell_{A,P,R,G}) \rightarrow \ell_{A,P',R',G'}}$$

Fig. 17. Dynamic semantics for convert.

convert, for which we give full semantics in Figure 17. Recall that, after elaboration, locations at runtime are explicitly annotated with their base type, predicate, rely, and guarantee:

$$\ell_{A,P,R,G}$$

Similarly, in the elaborated core language, convert is annotated with its result type:

$$\mathsf{convert}_\tau(e).$$

Every use of convert is subject to the following elaborated type rule:

$$\text{T-Conv-Elab}\quad \frac{\Sigma; H; \Gamma \vdash M : N \qquad \Gamma \vdash N \rightsquigarrow N'}{\Sigma; H; \Gamma \vdash \mathsf{convert}_{N'}(M)}$$

REFERENCES

Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound modular verification of c code executing in an unverified context. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*.

Richard J. Anderson and Heather Woll. 1991. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Symposium on Theory of Computing (STOC'91)*.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag.

John Boyland. 2003. Checking interference with fractional permissions. In *Proceedings of the Static Analysis Symposium (SAS'03)*.

Stephen Brookes. 2004. A semantics for concurrent separation logic. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'04)*. http://dx.doi.org/10.1007/978-3-540-28644-8_2

Venanzio Capretta. 2004. A Polymorphic Representation of Induction-Recursion. Retrieved September 12, 2012 from http://www.cs.ru.nl/∼enanzio/publications/induction_recursion.pdf.

Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*.

Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2009. Effective interactive proofs for higher-order imperative programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*.

Vasek Chvatal. 1983. *Linear Programming*. Macmillan.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOL'09)*.

Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. 2010. Local verification of global invariants in concurrent programs. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'10)*.

Thierry Coquand and Gerard Huet. 1988. The calculus of constructions. *Inform. Comput.* 76, 2 (1988), 95–120.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A logic for time and data abstraction. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'14)*.

Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional reasoning for concurrent programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'10)*.

Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. 2009. Deny-guarantee reasoning. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'09)*.

Peter Dybjer. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symbol. Logic* 65, 02 (2000), 525–549.

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying linearizability proofs with reduction and abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*.

Xinyu Feng. 2009. Local rely-guarantee reasoning. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*.

Cormac Flanagan and Martín Abadi. 1999. Types for safe locking. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'99)*.

Cormac Flanagan and Stephen N. Freund. 2000. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*.

Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-inductive definitions. In *Proceedings of the International Workshop on Computer Science Logic*. Springer, 454–468.

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*.

Colin S. Gordon. 2014. *Verifying Concurrent Programs by Controlling Alias Interference*. Ph.D. Thesis. University of Washington.

Colin S. Gordon, Michael D. Ernst, and Dan Grossman. 2013. Rely-guarantee references for refinement types over aliased mutable data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*.

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*.

Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Symposium on Distributed Computing (DISC'01)*.

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2006. A lazy concurrent list-based set algorithm. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'05)*.

Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*.

Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. http://doi.acm.org/10.1145/114005.102808

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. 2012. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*.

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*.

Jonas Braband Jensen and Lars Birkedal. 2012. Fictional separation logic. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'12)*.

Ranjit Jhala. 2015. LiquidHaskell: Refinement Types for Haskell. Retrieved from http://www. degoesconsulting.com/lambdaconf-2015/#talk-6f64790e6c.

Ranjit Jhala. 2016. Programming with Refinement Types. Retrieved from http://www.thestrangeloop.com/2015/programming-with-refinement-types.html.

Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying functional programs with abstract interpreters. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'11)*.

C. B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*.

Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. 2012. Deterministic parallelism with liquid effects. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*.

Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. 2009. Type-based data structure verification. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*.

Ming Kawaguchi, Patrick M. Rondon, and Ranjit Jhala. 2010. Dsolve: Safety verification via liquid types. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'10)*.

Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. 2015. Asynchronous liquid separation types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'15)*.

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*.

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*.

Barbara H. Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841.

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.

Maged M. Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504.

Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'96)*.

Filipe Militão, Jonathan Aldrich, and Luís Caires. 2014. Rely-guarantee protocols. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'14)*.

Filipe Militão, Jonathan Aldrich, and Luís Caires. 2016. Composing interfering abstract protocols. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'16)*.

Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. 2009. Towards type-theoretic semantics for transactional concurrency. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'09)*.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán André s Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'14)*.

Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent types for imperative programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*.

Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. 2010. Structuring the verification of heap-manipulating programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*.

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the International Workshop on Computer Science Logic (CSL'01)*.

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'10)*.

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Inform.* Issue 6 (1976), 319–340.

Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'11)*.

François Pottier. 2008. Hiding local state in direct style: A higher-order anti-frame rule. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS'08)*.

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent local subjective logic. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'15)*.

John C. Reynolds. 1988. *Preliminary Design of the Programming Language Forsythe*. Technical Report CMU-CS-88-159. Carnegie Mellon University.

Patrick Rondon. 2012. *Liquid Types*. Ph.D. Dissertation. University of California, San Diego.

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. 2012. CSolve: Verifying C with liquid types. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'12)*.

Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*.

Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and verifying concurrent algorithms with histories and subjectivity. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'15)*.

Jan Smans, Dries Vanoverberghe, Dominique Devriese, Bart Jacobs, and Frank Piessens. 2014. *Shared Boxes: Rely-Guarantee Reasoning in VeriFast*. Technical Report CW622. KU Leuven.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'14)*.

Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. 2013. Modular reasoning about separation of concurrent data structures. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'13)*.

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F⋆. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*.

R. Kent Treiber. 1986. *Systems programming: Coping with parallelism*. Technical Report. IBM Thomas J. Watson Research Center.

Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding reference immutability to java. In *Proceedings of the ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA'05)*.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*.

Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. 2006. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*.

Viktor Vafeiadis and Matthew Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *Proceedings of the International Conference on Concurrency Theory (CONCUR'07)*.

Niki Vazou. 2016. LiquidHaskell: Verification of Haskell Programs with SMTs. Retrieved from http://cufp.org/2016/t6-niki-vazou-liquid-haskell-intro.html.

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'15)*.

Niki Vazou, Patrick Rondon, and Ranjit Jhala. 2013. Abstract refinement types. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'13)*.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014a. LiquidHaskell: Experience with refinement types in the real world. In *Haskell Workshop*.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014b. Refinement types for haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*.

John Wickerson, Mike Dodds, and Matthew Parkinson. 2010. Explicit stabilisation for modular rely-guarantee reasoning. In *Proceedings of the European Symposium on Programing Languages and Systems (ESOP'10)*.

Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. 2007. Object and reference immutability using java generics. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*.

Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and immutability in generic java. In *Proceedings of the ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*.

Received August 2015; revised November 2016; accepted January 2017