# Rely-Guarantee References for Refinement Types Over Aliased Mutable Data

Colin S. Gordon, Michael D. Ernst, and Dan Grossman
University of Washington
{csgordon,mernst,djg}@cs.washington.edu

## Abstract

Reasoning about side effects and aliasing is the heart of verifying imperative programs. Unrestricted side effects through one reference can invalidate assumptions about an alias. We present a new type system approach to reasoning about safe assumptions in the presence of aliasing and side effects, unifying ideas from reference immutability type systems and rely-guarantee program logics. Our approach, *rely-guarantee references*, treats multiple references to shared objects similarly to multiple threads in rely-guarantee program logics. We propose statically associating rely and guarantee conditions with individual references to shared objects. Multiple aliases to a given object may coexist only if the guarantee condition of each alias implies the rely condition for all other aliases. We demonstrate that existing reference immutability type systems are special cases of rely-guarantee references.

In addition to allowing precise control over state modification, rely-guarantee references allow types to depend on mutable data while still permitting flexible aliasing. Dependent types whose denotation is stable over the actions of the rely and guarantee conditions for a reference and its data will not be invalidated by any action through any alias. We demonstrate this with refinement (subset) types that may depend on mutable data. As a special case, we derive the first reference immutability type system with dependent types over immutable data.

We show soundness for our approach and describe experience using rely-guarantee references in a dependently-typed monadic DSL in Coq.

***Categories and Subject Descriptors*** D.2.4 [*Software/Program Verification*]: Correctness Proofs; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

***Keywords*** reference immutability, rely-guarantee, refinement types

## 1. Introduction

A common way to reason about side effects in imperative languages is to restrict (disable) mutating some state in some code sections. This is seen most clearly in reference immutability [22, 44, 49, 50], but also in ownership [14] and region-based type systems [6]. The common approach is to attach permission/ownership/region information to references, where certain operations (mainly writes to the heap) through references with certain permissions are prohibited.

The program logic literature includes work ensuring that actions by one section of code do not interfere destructively with the assumptions of another section of code. This appears most often in the form of concurrent program logics, where the goal is to prevent destructive interference between threads. This reaches at least as far back as Owicki and Gries's technique [37], which checks thread non-interference by ensuring that no action would invalidate any intermediate assumption of another thread. Jones abstracted cross-thread interactions to a *rely* relation bounding interference by other threads, and a *guarantee* relation bounding actions of the current thread [27]. Each thread's local proof then requires all local actions to fall within its guarantee, and that all of its intermediate assertions are stable with respect to (that is, not invalidated by) any possible action permitted by the rely. Parallel composition of threads is then safe if each thread's guarantee implies each other thread's rely.

Our central idea is to treat aliases to objects similarly to threads of control in rely-guarantee program logics. Each reference's type carries a rely and a guarantee, bounding actions on an object through other references (rely) and bounding actions through the reference itself (guarantee). We call these augmented reference types *rely-guarantee references*. The type system maintains the invariant that the guarantee of any reference implies the rely of any alias. The type system checks these constraints when a program duplicates an alias. This raises the issue that some references cannot soundly coexist: no two references to the same object can each guarantee nothing (the reference permits arbitrary actions) and rely on restricted behavior through aliases. This presents us with a logical account of aliasing: some references may not be aliased without weakening the rely or guarantee of the source, and a reference with an empty rely necessarily has no aliases.

Rely-guarantee references generalize *reference immutability* [22, 44, 49, 50] to finer-grained control over interference through aliases. The traditional reference immutability qualifiers correspond to simple rely and guarantee conditions. For ref $\tau$[R,G] as a reference to data of type $\tau$ with rely R and guarantee G:

- readable $\tau \equiv$ ref $\tau$[any interference, no writes]
- writable $\tau \equiv$ ref $\tau$[any interference, any writes]
- immutable $\tau \equiv$ ref $\tau$[no interference, no writes]

Rely-guarantee references let us reason about some refinements of referents. Let a *stable* predicate over a reference be one that is preserved by its rely. Then a stable predicate cannot be invalidated by actions through an alias, and any new predicate that is stable and ensured by a guarantee-permitted action (on an object satisfying the old predicate) is true after the action, providing a form of strong update on arbitrarily aliased mutable data. (An action allowed by the guarantee that preserves the current predicate is a special case.)


*PLDI'13,* June 16–19, 2013, Seattle, WA, USA.

### 1.1 Contributions

***Refinement Types Over Mutable Data*** Rely-guarantee references permit refinement types [21] that depend on mutable data, without requiring any aliasing restrictions to support strong updates. We leverage the notion of a *stable assertion* from rely-guarantee program logics, allowing any refinements that are not invalidated by actions performed through other references. We prove that our type system is sound.

***Generalizing Reference Immutability*** We generalize reference immutability by combining it with rely-guarantee techniques. This is of independent interest, but also outlines an effort/precision spectrum from unrestricted references to reference immutability to rely-guarantee references.

***A Prototype Implementation*** We prototype an implementation as a shallow monadic embedding in Coq. We have used it to verify the examples in the paper, including implementing reference immutability as a special case. We briefly discuss our experience implementing a language as a Coq DSL and the manual proof burden for our technique versus purely functional versions. The implementation is available at
`https://github.com/csgordon/rgref/`.

We believe rely-guarantee references make a compelling argument that rely-guarantee reasoning is a promising way to statically reason about aliasing. Further, any technique traditionally used to reason about thread interference can be adapted to modularly reason about effects in the presence of aliasing (we present rely-guarantee references as a type system, but our ideas could be implemented in other ways, such as a program logic). Ultimately we believe the proper way to support unknown aliases in program verification is by treating aliases as different threads of control.

## 2. Rely-Guarantee References

A rely-guarantee reference is a reference to a heap structure of a given type, as in ML's ref $\tau$, with three additional type components:

- A refinement predicate $P$ over the $\tau$ and a heap $h$ that can enforce local properties and/or data-structure well-formedness.
- A guarantee relation $G$ over pairs of $\tau$s and heaps, restricting the effects to the referent (and state heap-reachable from that referent) that may be performed through this reference or those produced by dereferencing it.
- A rely relation $R$ specifying the actions permitted by (the guarantees of) other aliases to the referent.

We use the form $\mathsf{ref}\{\tau \mid P\}[R,G]$ for a rely-guarantee reference. Predicates and relations are defined not only over the $\tau$ a reference refers to, but also over heaps, to refine data reachable from the immediate referent. For a rely-guarantee reference type to be well-formed, the predicate $P$ must be *stable* with respect to the rely $R$: for all values and heaps for which the predicate holds, if the rely $R$ allows another value and heap to be produced by actions on another alias, then the predicate holds for the new value and heap as well: $P\ v\ h \wedge R\ v\ v'\ h\ h' \implies P\ v'\ h'$. This ensures that actions through aliases do not invalidate the refinement, and that all actions that may invalidate the refinement are local, so reasoning about such changes allows strong updates to the refinement. These issues are formally treated in Section 4.

A simple example of rely-guarantee references is a monotonically increasing counter, which we can represent as a value of type

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

where any is the trivial (always true) refinement, and increasing (Section 3.1) is a relation on natural numbers and heaps that requires the second `nat` to be greater than or equal to the first. Given a variable $x$ with the type above, $x \leftarrow !x + 1$ type-checks (! is ML's dereference operator). By contrast, incorrect code that decrements the counter cannot satisfy the guarantee relation increasing.

A read-only alias to an increasing counter can be expressed as:

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

where $\approx$ is a relation permitting no change.

We might wish to know more about a counter value, for example that it is greater than 0 so it is safe as a divisor to compute an average. Any write to the counter via any reference will increase its value, and may therefore conclude the result is greater than 0.[1] Furthermore, it is safe to continue assuming the value is greater than 0 because the reference's rely ensures no alias can decrease the value. We say $\lambda x : \texttt{nat}.\, \lambda h : \mathsf{heap}.\, x > 0$ is *stable* with respect to the rely increasing. When a write establishes a new stable predicate over the data, strong updates to the reference's predicate (changing the predicate in the type) are sound. (Similarly, when a write invalidates a reference's predicate, a strong update is required, to a new predicate stable over the rely.)

Many verification techniques for imperative programs struggle to verify examples of this kind. Reference immutability and fractional permissions [7, 28] can only allow or outright prevent mutation, not control it. Separation logic cannot concisely specify the counter's intended semantics, only code's behavior. Rely-guarantee and related systems can express the semantics among threads [18, 27, 45], but only coarsely [46] among different program sections. Most program logics can constrain the actions of a function on an argument, but the specification must deal with aliasing, either by giving linear semantics to knowledge of the counter (as in separation logic), or by explicitly treating aliasing (as in more traditional Hoare logics [24]).

With rely-guarantee references, functions are written without concern for aliasing among their arguments. A function cannot be called with unsafe aliasing among arguments: since each alias's guarantee must imply each other's rely, each function explicitly accounts for its possible actions. If two arguments of the same type have conflicting rely/guarantee conditions, they cannot be aliased.

### 2.1 Subtleties of Rely-Guarantee References

While the intuition behind rely-guarantee references is straightforward, this section overviews some more subtle features of our system that avoid problems.

***Non-duplicable References*** A reference may be freely duplicated if its guarantee implies its own rely, as with the monotonically increasing counter. But consider a reference

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

Making an alias to $y$ where the the alias has the same type as $y$ violates soundness, because the guarantee of the duplicate does not imply the rely of $y$! Instead, aliasing $y$ requires splitting it into two aliases with weaker rely/guarantee conditions. We support such splitting via a novel substructural resource semantics (Section 4).

***Reference to References*** We need a reference's guarantee to restrict all actions performed using that reference, which must include actions performed via references acquired by dereferencing the first reference. Otherwise, reading a reference out of the heap and writing through it could violate the original reference's guarantee, violating the "capability to perform effects in the guarantee" intuition, and potentially invalidating a predicate. So reading from the heap must somehow transform the type of the referent to restrict resulting references. Reference immutability systems can give a simple binary function on permissions [22], to capture the transitive meaning of

[1] Because the type nat of natural numbers contains no negative numbers.

qualifiers. For example, dereferencing a `readable` reference to a `writable` reference returns a `readable` reference (assuming a *deep* interpretation of reference immutability, where permissions apply transitively). By contrast, our type system combines *arbitrary* relations (Section 3.2). Furthermore, if one reference points to another, how should the rely of the "inner" reference be related to the outer one? It is unsound if it permits more interference than the outer rely, so our type system prevents this.[2]

***Footprint*** How much of the heap may a rely-guarantee reference's predicate or conditions mention? It is not productive or sound to let a reference constrain unrelated heap data: letting a reference arbitrarily constrain the heap could lead to allocating a new heap cell whose rely is not implied by existing references. The type system restricts the expressiveness of these predicates to ensure sound and tractable reasoning: predicates and relations may depend only on the heap reachable from the reference.

***Cycles*** Many useful data structures contain cycles, so we wish to reason effectively about them. The solution turns out to be simple (propositions describing cycles require finite proofs, and recursion based on heap structure is not permitted in predicates), but was not immediately obvious to us.

## 3. Examples

We present examples using rely-guarantee references to verify programs. The examples are small, but highlight distinct capabilities of rely-guarantee references. Rather than writing examples in our core language RGREF (Section 4), we present them using a slight simplification of our shallow embedding in COQ [12]. The embedding is largely in the style of YNOT [11, 34], using axioms for heap interactions.

***Reading* COQ *Source*** COQ's language for defining functions and types is based strongly on ML, though many keywords are different: `Definition` and `Fixpoint` for non- and recursive definitions, `Inductive` for defining inductive variant datatypes by specifying constructors. `Parameter` declares assumptions, external functions, or abstract elements in a module signature. Functions and parameterized type definitions can put some arguments in braces rather than parentheses; these arguments are implicit, and inferred when possible from later arguments. Another notable syntactic change from ML is that = is an operator for *propositional* equality, not a boolean decision procedure for structural equality. Therefore, := is often used where ML would use = in definitions. The set of types is much richer than ML, not only due to dependent types but because there are universes (types of types): Prop is the type of propositional types (erasable during extraction, such as proof terms with conjunction, implications, etc.), and Set is the type of normal (computationally relevant) data types. COQ also includes a notation feature that allows users to extend the grammar with additional parsing rules, allowing programs to use syntax closer to mathematical definitions (such as $\mathsf{ref}\{T \mid P\}[R,G]$). Our notation uses ML's dereference operator (`!`) and uses $r \leftarrow e$ for writing $e$ to the location referenced by $r$. We introduce further notations as they arise.

The PROGRAM extension [42] (used via definitions prefixed with `Program`) allows the omission of explicit proof terms in programs. Omitted terms are either solved automatically via a (customizable) proof search tactic, or set aside for subsequent manual interactive solving, improving readability.

[2] This is actually a design decision that simplifies checking stability. An alternative design could check a predicate for stability over any change permitted by any reference reachable from the predicate's target referent.

### 3.1 Monotonic Counter

Consider again our running example of a monotonically increasing counter. Generally, rely and guarantee conditions must be defined over pre- and post-heaps as well as values, to describe the interference they tolerate on reachable substructures. For a simple counter, there is no other reachable data, so the pre- and post-heaps may be ignored. Thus the relation for increasing over time is defined as:

```
Definition increasing (n n':nat) (h h':heap) : Prop :=
  n' ≥ n.
```

Code to allocate a counter is straightforward:

```
Program Definition mkCounter (_:unit)
  : ref{nat|any}[increasing,increasing] :=
    alloc 0.
```

The allocation function `mkCounter` generates well-formedness proof obligations for the resulting type:

- that any is stable with respect to increasing
- that any and increasing are precise: they access only the (empty) heap segment reachable from the natural number they apply to
- that any is true of 0

In our prototype implementation (Section 5), most of these obligations are proven automatically by lightly-guided automatic proof search. Type errors for actions that fall outside the guarantee (or ill-formed rely/guarantee relations, or predicates that are not precise, etc.) manifest as unsolvable proof obligations.

Using a monotonic counter is also straightforward:

```
Program Definition example (_:unit) :=
let x = mkCounter () in x ← !x + 1;
```

An assignment typechecks only if the change implied by the write is permitted by the reference's guarantee relation, for any pre-heap and pre-value satisfying the reference's refinement. In this case, the assignment generates a proof obligation of the form

$$\forall x, h.\, \texttt{any}\ (!x)\ h \implies \texttt{increasing}\ (!x)\ (!x+1)\ h\ h[x \mapsto h[x]+1].$$

which is easily solved, with little effort beyond what is required to verify a pure-functional increment function (see Section 5.3). Each read also generates a proof obligation that the guarantee `increasing` is "reflexive": it allows a reference to be used without modifying the heap ($\forall n, h.\, \mathsf{increasing}\ n\ n\ h\ h$). By contrast, an empty guarantee relation would disallow using a reference.

The monotonically increasing counter was proposed by Pilkiewicz and Pottier [40] as a challenging goal for program verification. Unlike their solution and another in fictional separation logic [26], we can state the monotonicity property plainly and require no abstraction to prevent unchecked interference. On the other hand, their solutions verify that the increment occurs, while ours ensures that increment is the only permitted action.

### 3.2 Monotonic List

We can define a monotonically growing (prepend-only) list, either using a mutable reference to a pure-functional list, or using mutable nodes. The former approach is similar to the monotonic counter, so to show the power of rely-guarantee references for recursive data structures, Figure 1 shows the latter.[3]

We first define `hpred` and `hrel`, type-level functions that allow shorter type declarations. We use them throughout this paper. Next we define a linked list structure, with restricted references to the

[3] The most natural design uses mutual inductive types where one indexes the other, which we assume here. COQ does not support this, so we use an encoding, discussed further in Section 5 and our technical report [23].

```
Definition hpred (A:Set) := A -> heap -> Prop.
Definition hrel (A:Set) := A -> A -> heap -> heap -> Prop.
Inductive list : Set :=
  | nil : list
  | cons : forall (n:nat),
                ref{list|any}[list_imm,list_imm] -> list
with (* list tails are immutable (_imm) *)
  list_imm : list -> list -> heap -> heap -> Prop :=
  | imm_nil : forall h h', list_imm nil nil h h'
  | imm_cons: forall n tl h h',
       list_imm h[tl] h'[tl] h h' ->
          list_imm (cons n tl) (cons n tl) h h'.
(* Convenient allocation functions *)
Program Definition Nil {P:hpred list}
  : ref{list|P}[list_imm,list_imm] := alloc nil.
Program Definition Cons {P P':hpred list}
    (n:nat)(tl:ref{list|P}[list_imm,list_imm])
  : ref{list|P'∩(λ l h=>l=cons n tl)}[list_imm,list_imm]
  := alloc (cons n tl).
(* A prepend-only list container *)
Record list_container (P:hpred list) :=
  mkList { head : ref{list|P}[list_imm,list_imm] }.
Inductive prepend : hrel (list_container P) :=
  | prepended : forall c c' h h' n,
       h'[head c']=cons n (head c) -> prepend c c' h h'
  | prepend_nop : forall c h h', prepend c c h h'.
Program Definition newList (P:hpred list)
  : ref{list_container|any}[prepend,prepend] :=
    let x = Nil in alloc (mkList P x).
Program Definition doPrepend {P:hpred list}(n:nat)
  (l:ref{list_container|any}[prepend,prepend]) : unit :=
    let x = Cons n (head l) in
        l ← mkList P x.
```

**Figure 1.** RGREF code for a prepend-only linked list.

tail. `list_imm` constrains the tail to be immutable. For the reader unfamiliar with COQ, `list_imm` is a GADT [48] constructing a proposition on different constructions of lists. The first constructor declares that an empty list must remain empty, regardless of heaps. The second constructor accepts a `nat`, a tail, two heaps, and a proof that `list_imm` holds over the tail of the list in those heaps, returning a declaration that a cons cell must remain constant. The immutability requirement is not essential to this example (we could, for example, permit the numbers to change but require the length to increase), but is included for completeness. We then define convenient helper functions for heap-allocating nil or cons cells.

We enforce the prepend-only behavior through reference to a `list_container` structure, which holds a reference to a list parameterized by some predicate. The `prepend` relation on `list_containers` allows prepending and no-ops (required for reading the reference). Finally we have helper functions to allocate a new list and to prepend the list with a new cell. `prepend` is essentially the specification of what `doPrepend` is permitted to do with the list.

Note the predicate conjunction (∩) in the return type of `Cons`. This, along with the predicate conversion rule, is how flow-sensitive assumptions can be handled (notice that the equality is stable with respect to `list_imm`). This is important in `doPrepend`, where information from the result of one write (inside `Cons`) must be carried into another (the assignment through `l`), because it is otherwise unavailable in the expression stored.

Not shown in Figure 1 are implicit obligations such as $\forall h.\, P\,\texttt{nil}\,h$ in `Nil`. Other such obligations include:

- $\forall tl, h.\, P\,tl\,h \implies P'\,(\texttt{cons}\,n\,tl)\,h$ in `Cons`
- That `prepend` permits the write in `doPrepend` (under the trivial assumption that `any` holds of the initial list container and heap). This obligation requires a richer type for the allocation result, because `mkList` must know `x` is a cons cell whose tail is the old list. This information is not available locally (within the write statement itself). Other systems propagate hypotheses separately, but we only need to track variables: the required equality is present in `x`'s predicate because of `Cons`'s return type.
- The stability and precision properties that must hold for various predicates and `list_imm`.
- Propagations of these obligations to indirect polymorphic callers, such as `newList` and `prepend`.

Also omitted in Figure 1 are obligations related to *folding* and *containment*. Folding is the restriction of read result types to ensure that for any reference $r$ with guarantee $G$, references produced via reads of $r$ do not allow actions exceeding those permitted by $G$ on $r$'s referent. This ensures actions via a reference read from inside a data structure cannot invalidate predicates over the whole structure. Containment is a check that the rely $R$ for a reference $r$ captures all interference allowed by the interference summaries of references reachable from $r$. This ensures that any predicate preserved by $R$ is also preserved by actions on aliases to internal structures.

Both operations require projecting a given relation componentwise onto a datatype's members. For our prepend-only list, projecting `prepend` is trivial (it does not constrain the list cells' values), and the result of projecting `list_imm` is logically equivalent to `list_imm` itself.

### 3.3 Reference Immutability as a Special Case

Reference immutability [22, 44, 49, 50] adds permissions (type qualifiers) to references to permit or disallow side effects through a particular reference. Multiple aliases at different permissions may coexist if compatible: for example, there may be write-permitting and read-only aliases to an object. We can define the permissions of reference immutability like this:

```
Definition havoc {A:Set} : hrel A :=
    fun x => fun x' => fun h => fun h' => True.
Definition readable (T:Set) := ref{T|any}[havoc,≈].
Definition writable (T:Set) := ref{T|any}[havoc,havoc].
Definition immutable (T:Set) := ref{T|any}[≈,≈].
```

Our definitions encode the standard semantics for reference immutability qualifiers: only `immutable` assumes limited interference via other aliases, and `readable` and `immutable` disallow mutation through a reference. Restrictions on aliasing among reference immutability permissions are reflected in the rely and guarantee relations: no heap cell may have writable and immutable aliases simultaneously, as the guarantee of the writable reference (`havoc`) is not a subrelation of the immutable rely (≈). The "containment" requirement (Section 3.2) for rely conditions on nested datatypes is satisfied by the rely for readable and writable, and for immutable the rely prevents it from (transitively) referencing mutable data.

Reading through one of these references requires considering how the rely and guarantee affect the read's result. In reference immutability, result types are adapted using a simple binary function on permissions (Figure 2). Our rely-guarantee reference type system must combine arbitrary relations (folding), using the type of the referent. Intuitively, folding is projection of the reference's guarantee onto the referent type. Any projection of `havoc` and ≈ correspond with the simplified version in Figure 2. Projecting `havoc` is equivalent to reading through a `writable` reference, which simply produces the inner type. Projecting ≈ is equivalent to the weakening that occurs when reading through `readable` or `immutable` references.

We can also give a reference immutability system with limited dependent types by a small extension:

```
          _   ▷   immutable   =   immutable
  immutable   ▷   _           =   immutable
   readable   ▷   _           =   readable
   writable   ▷   q           =   q
```

**Figure 2.** Combining reference immutability permissions, from [22]. Using a $p$ reference to read a $q$ $T$ field produces a $(p \triangleright q)$ $T$.

```
Definition refined (T:Set) (P:hpred T) := ref{T|P}[≈,≈].
```

At first glance this is weaker than proposed systems that let mutable data's type depend on arbitrary immutable data, because we require any reference predicate to access only heap state reachable from its referent. At the cost of some space the referent could maintain its own extra reference to relevant immutable data. Careful code extraction work can improve the space overhead in executables.

Another benefit of implementing reference immutability via rely-guarantee references is interoperability between reference immutability and richer rely-guarantee references. For example, a function accepting a `readable` reference to a natural number can be passed a read-only monotonically-increasing counter from Section 3.1. This offers a natural path for gradually adding stronger verification guarantees to code using reference immutability (which itself is a gradual refinement of unrestricted references [22]).

### 3.4 RCC/Java with Reference Immutability

The core of RCC/Java [20] is also implementable as a small library using our Coq DSL, and we present a translation of an early version [19]. The key idea in these type systems and related systems is to parameterize the type of a reference by the identity of a particular lock. The type system tracks the set of held locks and permits reads and writes through a reference only when the reference's lock parameter is statically known to be held.

A Coq module wraps standard acquire and release primitives and exposes a new reference type that quantifies over a lock. The RCC reference type can be abstracted with a module signature, but can be concretely represented by a RGref `ref` type:

```
(* Signature *)
Parameter rccref : forall (A:Set),
    hpred A -> hrel A -> hrel A -> lock -> Set.
...
(* Implementation *)
Definition rccref A P R G (l:lock) := ref A P R G.
```

The module then exposes its own read and write primitives, and external ways of proving goals like guarantee satisfaction that do not expose the internal `rccref` representation. This mostly consists of re-exporting existing axioms using new names. Then an explicit lock witness (since the type system is not specialized to track lock witnesses) can be abstracted using:

```
Parameter lockwitness : lock -> Set.
Parameter locked : forall {l:lock}, hpred (lockwitness l).
Parameter unlocked : forall {l:lock}, hpred (lockwitness l).
```

The acquire and release operations must respectively produce and consume a witness that a lock is held, that permits release. Using a binary operator `-->` on predicates that produces a relation allowing changes from states where the first predicate holds to states where the second holds (a limited encoding of protocols), a witness may have type $\mathsf{ref}\{\mathsf{lockwitness} \mid \mathsf{locked}\}[\mathsf{empty}, \mathsf{locked}\texttt{-->}\mathsf{unlocked}]$. Using an empty rely implies uniqueness, and requiring such a witness to release the lock prevents splitting the witness, which would requiring weakening the rely of both resulting references. The read and write for `rccref`s would need to also require some witness that the lock was held:

| | | | | |
|---|---|---|---|---|
| Expressions | $e$ | ::= | $x$ | Variable |
| | | \| | $e\ e$ | Application |
| | | \| | $!e$ | Dereference |
| | | \| | $e[e]$ | Heap Select |
| | | \| | $\mathsf{alloc}\ e$ | Allocation |
| | | \| | $(\lambda x:\tau.e)$ | Pure Function |
| | | \| | $(\lambda_{\mathcal{M}}(x:\tau).e)$ | Procedure |
| | | \| | $\tau$ | Types |
| | | \| | $x \leftarrow e$ | Store |
| | | \| | $\mathsf{swap}(x,e)$ | Atomic swap |
| Types | $\tau$ | ::= | $=$ | Propositional equality |
| | | \| | $\mathsf{Prop}$ | Propositions |
| | | \| | $\mathsf{ref}\{\tau \mid e\}[e,e]$ | Reference |
| | | \| | $\mathsf{Type}$ | Type of Prop |
| | | \| | $\Pi x:\tau \to \tau'$ | Dep. Product (pure) |
| | | \| | $\tau \xrightarrow{\mathcal{M}} \tau'$ | Impure Function |
| | | \| | $\mathsf{heap}$ | Heaps |
| | | \| | $e$ | Expressions |

**Figure 3.** Syntax, omitting booleans ($b$ : `bool`), unary natural numbers ($n$ : `nat`), unit, pairs, propositional True and False, and standard recursors. The expression/type division is presentational; the language is a single syntactic category of PTS-style [2] pseudoterms.

```
Program Definition rcc_read ...{l:lock}
    (w:ref{lockwitness l | locked}[empty,locked-->unlocked])
    (r:rccref A P R G l)... := (!r,w).(* return the witness *)
```

This encoding of RCC/Java using dependent types is not novel, but note the rely, guarantee, and predicate of the underlying reference are exposed, yielding the first combination of race-free lock acquisition and reference immutability we are aware of, in addition to exposing the full power of rely-guarantee references over lock-guarded data.

## 4. A Type System for Rely-Guarantee References

Figure 3 gives the syntax for a core language RGref with rely-guarantee references. The expressions combine features from the ML-family (e.g., references) and dependently typed languages (e.g., dependent product), specifically from the Calculus of Constructions [2, 13]. We include a few basic datatypes (natural numbers, booleans, pairs, unit), a type for propositional equality, and their standard recursors [25]. We also distinguish effectful functions through the term former $\lambda_{\mathcal{M}}$ and the effectful non-dependent function type former $\tau \xrightarrow{\mathcal{M}} \tau'$ ($\mathcal{M}$ for mutation).

The language supports reasoning about heaps: not only is there a standard form of dereference, but there is a term for dereferencing a reference in a particular heap, used to specify predicates and rely/guarantee relations using the propositions-as-types principle. The language is designed to use propositions-as-types to specify predicates and relations, and to use the pure sublanguage as a computational language amenable to rich reasoning, but to use external means for discharging obligations like a write satisfying a guarantee. The presence of current-heap-dereference makes the pure term language itself unsound as a logic, internally offering assurances similar to Cayenne [1]. In general, the language for predicates and relations can be distinct from the term language, and the term language does not require advanced types; our design is motivated by our implementation as a Coq DSL (Section 5).

Figure 4 presents the primary typing rules for the core language. There are two key judgments: $\Gamma \vdash e : \tau$ for pure terms (useful for proofs), and $\Gamma \vdash e : \tau \Rightarrow \Gamma'$ for impure and substructural computation. Those pure rules omitted here (recursors for the assumed inductive types, typing the primitive types in Prop, etc.) are standard for a pure type system.

The imperative typing judgment $\Gamma \vdash e : \tau \Rightarrow \Gamma'$ is flow-sensitive to allow reasoning about when references are duplicated. Crucially, it allows reasoning about when a reference must not be duplicated

because its guarantee does not imply its own rely. For this reason, we have two impure variable rules: one consumes the variable (V-$\emptyset$), and the other uses an auxiliary relation $\Gamma \vdash \tau \prec \tau \divideontimes \tau$ to split a type (V-$\divideontimes$). Primitive types (`nat`, `bool`, `unit`) freely split into two copies of themselves. We require that any variable captured by a closure has a self-splitting type ($\Pi$-I and FUN), and thus functions may be duplicated freely. We require that only values of self-splitting types are captured by dependent type constructors ($\Pi$-F and the not-shown propositional equality rule), so types are also self-splitting. Variables read in pure computation must also be self-splitting (V).

References (and structures that may contain them, like pairs) are the only types with non-trivial splitting. Reference types split into reference types that may coexist (each guarantee implies the other rely, both relies are no stronger than the original rely, stable predicates, etc.), and pairs split into pairs of the component-wise split results. For example, the problematic reference from Section 2.1 has non-trivial splitting behavior, mediated by REF-$\divideontimes$:

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

Splitting this reference requires consuming the original and producing two "weaker" references: each guarantee may permit at most what the original guarantee allowed, and each rely must assume at least as much interference as the original. For example,

$$\begin{aligned}\varepsilon \vdash\ & \mathsf{ref}\{\texttt{nat} \mid \mathsf{any}\}[\mathsf{decreasing}, \mathsf{increasing}] \\ \prec\ & \mathsf{ref}\{\texttt{nat} \mid \mathsf{any}\}[\mathsf{havoc}, \mathsf{increasing}] \\ \divideontimes\ & \mathsf{ref}\{\texttt{nat} \mid \mathsf{any}\}[\mathsf{havoc}, \mathsf{increasing}]\end{aligned}$$

The natural use of simply duplicating a reference whose guarantee implies its own rely (as in the monotonic counter) is a degenerate case of the very general rule REF-$\divideontimes$.

The conversion relation $\Gamma \vdash \tau \rightsquigarrow \tau$ is a directed call-by-value $\beta$-conversion (so for example $\beta$-reduction is not used with arguments containing dereferences) plus reducing abstractions whose bound variable is free in the result, and what amounts to subtyping by converting predicates and relations to weaker versions: P-$\Rightarrow$ weakens the predicate; R-$\subset$ assumes more interference may occur; and G-$\subset$ sacrifices some permissions; P and R changes may affect stability.

***Mutation*** The most interesting rules are those for mutation, particularly for writing to the heap (WRITE). This rule requires (beyond basic type safety) that the effects fall within the guarantee, assuming the reference's predicate holds in the current heap.[4] It also allows the option of a strong update to the reference's predicate, if the change establishes some new stable predicate. For example,

$$\begin{aligned}& x : \mathsf{ref}\{\texttt{nat} \mid \lambda x.\, \lambda h.\, x = 3\}[\mathsf{empty}, \mathsf{havoc}] \vdash \\ & \quad x \leftarrow 4 : \texttt{unit} \\ & \Rightarrow x : \mathsf{ref}\{\texttt{nat} \mid \lambda x.\, \lambda h.\, x = 4\}[\mathsf{empty}, \mathsf{havoc}]\end{aligned}$$

Thus RGREF naturally supports strong updates on unique references as a degenerate case. The atomic swap operation (which permits modifying substructural fields) leverages the heap-write rule's premises. Allocation simply requires a well-formed type as a result and establishing the predicate over the value in any heap. The imperative part of the language also includes non-dependent function types, application, and the use of pure expressions.

Dereference uses the *relation folding* function $[R, G] \gg \tau$ (Figure 5) to reason about the rely and guarantee in result types. It has no effect for non-reference types. For types containing references, the result type is rewritten by intersecting the projection of the guarantee onto each component with the stated guarantee for the component itself. This can cause some precision loss. The effect of folding on the rely has no impact: because the rely for any well-formed reference type has to contain / admit the effects allowed by the rely of any reachable reference type, the intersection on the rely component would produce a relation semantically equivalent to the syntactically present rely on the inner reference (the same type of relation projection is used to check rely containment as is used in folding the guarantee). In general, relation folding and checking containment are straightforward for types whose members are always reflected in a type index (e.g., pairs, references). Folding for richer types, such as full inductive types [39] is left to future work. The Deref rule also checks that the source type is self-splitting. This ensures that the (possibly weaker) guarantee of the result implies the original location's rely, and the original value's guarantee(s) will imply the (unaltered) rely relations of the result.

The rule for typing reference types themselves (WF-REF) imposes several additional requirements on predicates and relations. First, the predicate $P$ must be stable with respect to $R$. Second, the relations and predicates must be *precise*: all depend only on heap state reachable from the referent. This prevents code from rendering the system unsound by allocating a new cell whose rely condition requires the heap to be invariant: that rely would be undesirable if enforced as it prevents all mutation and allocation, but unsound if ignored because all predicates are stable over such a rely. Finally, the rely must be *closed* (contained): any changes permitted by relies of references reachable from the referent are also permitted by the checked rely. This ensures that checking $\mathsf{stable}\ P\ R$ is sufficient to ensure $P$ is not violated (otherwise $P$ could depend on other references reachable from the referent, whose rely relations might permit $P$ to be invalidated). Figure 5 defines these notions precisely. WF-REF also requires as a side condition that $P$, $R$, and $G$ are free of dereference expressions ($!e$), since implicitly heap-dependent predicates are not sensible.

We foresee no technical difficulty in building the system directly atop stronger calculi such as the Calculus of Inductive Constructions [4] (CIC) beyond richer treatment of folding for the full spectrum of inductive types (Section 5.2). There are a few essential qualities required for soundness. First, effectful terms and abstractions are encapsulated in a separate judgment (which corresponds to a monadic treatment of effects in translation to a pure system). Any term in the pure fragment must be reflexively splittable, to avoid introducing resource semantics into the pure sublanguage. Captured variables have reflexively splittable types ($\Gamma \vdash \tau \prec \tau \divideontimes \tau$). In principle this could be weakened, but we prefer the simplicity of allowing function terms to be arbitrarily duplicated. We build upon CC for simpler presentation.

### 4.1 Soundness Sketch

Soundness follows a preservation-like structure. Evaluation must preserve a couple invariants beyond standard heap soundness:

- For each reference $r : \mathsf{ref}\{T \mid P\}[R, G]$ in the stack, heap, or expression under reduction, there exists a proof of $P\ (h[r])\ h$ for the current heap $h$.
- For each pair of references $p : \mathsf{ref}\{T \mid P\}[R, G]$ and $q : \mathsf{ref}\{T \mid P'\}[R', G']$ in the stack, heap, or expression under reduction, if $p$ and $q$ alias (point to the same heap cell) then $\emptyset \subset G' \subseteq R$ and $\emptyset \subset G \subseteq R'$.

Initially there are no references, so these hold trivially. On allocation, the predicate for the new object is true in all heaps by inversion on the allocation typing, so the result is immediate. On any action through a reference, the type system ensures the action falls within the guarantee. The action either preserves the predicate or produces a new (stable) one, and the proof for that reference's new refinement is easy to construct from the typing derivation results. For aliases, the used reference's guarantee must imply the rely of any alias, and the alias's predicate is stable over its rely, and therefore preserved by the

[4] The reflexivity goals DEREF generates could also assume the predicate (i.e., reflexive on states satisfying the predicate), but we haven't needed this.

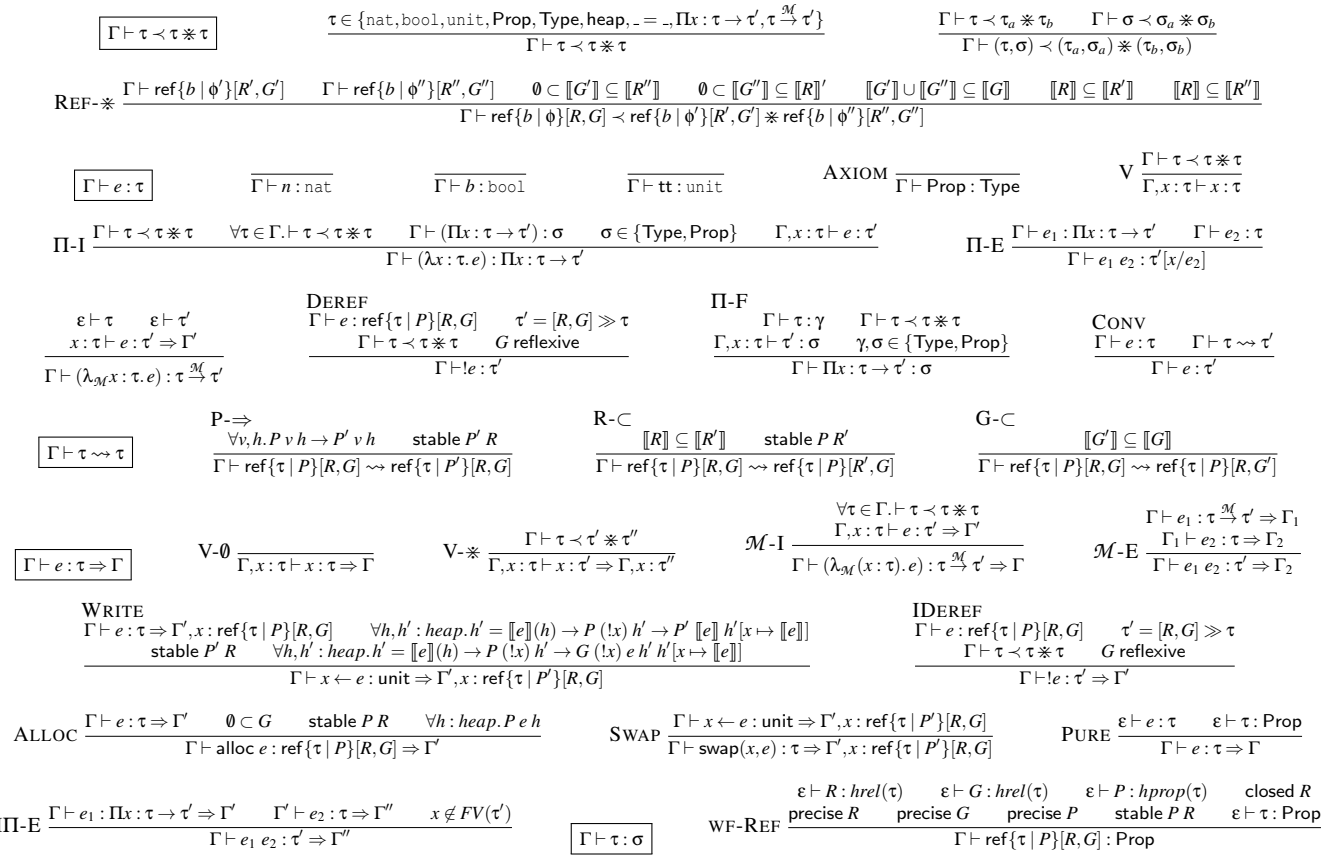


**Figure 4.** Typing. Not shown: standard recursors for naturals, booleans, pairs, identity types [25]. Also not shown: standard well-formed contexts, most of (pure) expression/type conversion ($\Gamma \vdash \tau \rightsquigarrow \tau$) (see our technical report [23]).

action within the used reference's guarantee. For other references, the action will fall within its rely (by containment) and thus a similar use of a stability proof suffices, or the changed cell is not reachable from the reference in question, in which case the predicate is preserved by precision.

This proof is simplified by slightly nonstandard operational semantics. First, we execute an elaborated language where conversions, strong updates, and substructural reads of variables are explicit (translation from a typed source program is straightforward). To prove soundness of substructural manipulation, impure code manipulates a mutable stack, and references are "fat pointers" of the underlying pointer, rely, guarantee, and predicate. Variable capture in imperative closures occurs via a rule that inlines all free variables when reducing to a value. Folding has a runtime representation, since folded types sometimes occur as part of runtime values. Finally, the semantics sometimes reduce multiple dereferences at once inside pure expressions, when type equivalence requires the dereference expressions to be definitionally equal. This avoids trouble where the naïve reduction might reduce definitionally equal dereferences in different heaps (with different values). We have yet to conceive of desirable computational code that observes this behavior. Our technical report provides more details [23].

## 5. Implementation

To understand RGREF's effect on data structure design and the effort required for verification, we have implemented RGREF as a shallow embedding in COQ, and used it to implement Section 3's examples. This includes implementing reference immutability, meaning our RGREF implementation can be used to write programs using reference immutability, and to gradually refine parts of those programs to use more fine-grained rely and guarantee conditions and predicates. Overall, we found that RGREF required careful choice of type refinements, but did not affect algorithm design and had reasonable proof burden (commensurate with the complexity of the code verified).

The implementation is done largely in the style of YNOT [11, 34], with axioms for heap interaction, and using COQ's notation facilities to elaborate source terms to COQ terms with proof holes, which are then elaborated and semi-automatically solved by Sozeau's PROGRAM extension [42]. Each structure typically requires its own slightly customized PROGRAM tactic for effectively solving most goals, but we find that following the tactic development style Chlipala recommends [10] tends to work well, as each module typically handles its own family of predicates and relations. Proofs involving heaps are carried out using a small set of axioms reflecting invariants maintained by the semantics. The most useful axiom is $\mathsf{heap\_lookup} : \forall h, A, P, R, G. \forall r : \mathsf{ref}\{A \mid P\}[R, G]. P\ (h[r])\ h$ which means that in any heap, the type system ensures there is a proof of the refinement for every valid reference. The implementation also relaxes some restrictions present in the formal language, such as allowing values to be projected into predicates (as in Section 3.2); predicates, rely and guarantee relations must simply abstain from dereference.

$$\mathsf{stable}_\tau\ (P : \mathit{hprop}(\tau))\ (R : \mathit{hrel}(\tau)) \stackrel{\text{def}}{=} \forall x : \tau.\, \forall h, h' : \mathsf{heap}.\, P\ x\ h \to R\ x\ h\ h' \to P\ x\ h'$$

$$\mathsf{precise}_\tau\ (P : \mathit{hpred}(\tau)) \stackrel{\text{def}}{=} \forall x, h, h'.\, (\forall l, \mathsf{ReachFromIn}\ l\ x\ h \to h[l] = h'[l]) \to P\ x\ h \to P\ x\ h'$$

$$\mathsf{precise}_\tau\ (R : \mathit{hrel}(\tau)) \stackrel{\text{def}}{=} \forall x, x', h, h', \overline{h}, \overline{h'}.\, (\forall l, \mathsf{ReachFromIn}\ l\ x\ h \to h[l] = h'[l]) \to (\forall l, \mathsf{ReachFromIn}\ l\ x'\ \overline{h} \to \overline{h}[l] = \overline{h'}[l]) \to R\ x\ x'\ h\ \overline{h} \to R\ x\ x'\ h'\ \overline{h'}$$

$$\mathsf{closed}_\tau\ (R : \mathit{hrel}(\tau)) \stackrel{\text{def}}{=} \begin{cases} \mathsf{True} & \text{if } \tau \in \{\mathtt{nat}, \mathtt{bool}, \mathsf{unit}, \Pi x : \tau' \to \tau'', \tau \xrightarrow{\mathcal{M}} \tau', \mathsf{True}, \mathsf{False}, \mathsf{heap}, \_ = \_\} \\ \mathsf{closed}\ R.1 \wedge \mathsf{closed}\ R.2 & \text{if } \tau = (\sigma * \sigma') \\ \mathsf{closed}\ R' \wedge \forall l, h, h'.\, R'\ h[l]\ h'[l]\ h\ h' \to R\ l\ l\ h\ h' & \text{if } \tau = \mathsf{ref}\{\tau' \mid P'\}[R', G'] \end{cases}$$

$$[R, G] \gg \tau \stackrel{\text{def}}{=} \begin{cases} \tau & \text{if } \tau \in \{\mathtt{nat}, \mathtt{bool}, \mathsf{unit}, \Pi x : \tau' \to \tau'', \tau \xrightarrow{\mathcal{M}} \tau', \mathsf{True}, \mathsf{False}, \mathsf{heap}, \_ = \_\} \\ ([R.1, G.1] \gg \sigma * [R.2, G.2] \gg \gamma) & \text{if } \tau = (\sigma * \gamma) \\ \mathsf{ref}\{\sigma \mid P\}[R', \begin{array}{l}(\lambda a, a', h, h'.\, G'\ a\ a'\ h\ h' \wedge \\ \quad (\forall l, h[l] = a \to h'[l] = a' \to G\ l\ l\ h\ h'))\end{array}] & \text{if } \tau = \mathsf{ref}\{\sigma \mid P\}[R', G'] \end{cases}$$

where for $R : \mathit{hrel}(\sigma * \sigma')$: $R.1 \stackrel{\text{def}}{=} \lambda x, x' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall y : \sigma'.\, R\ (x, y)\ (x', y)\ h\ h'$ and $R.2 \stackrel{\text{def}}{=} \lambda y, y' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall x : \sigma'.\, R\ (x, y)\ (x, y')\ h\ h'$

**Figure 5.** Selected auxiliary judgments and predicates

We made compromises to fit into Coq. Notably, Coq lacks support for mutual inductive types where one indexes the other (e.g., a datatype simultaneously defined with an inductive predicate on that type). Our implementation adapts a standard encoding [8] of induction-recursion [17] outlined in our technical report [23] to support examples like the list in Section 3.2. Any use of this encoding somewhat complicates generated proof obligations and data structure designs. Thus our current implementation is best-suited to "functional-first" designs that make only light use of references, as is common in OCaml, Scala, and F# code. We stress that this is a limitation of our implementation by embedding in Coq, not a fundamental limitation of rely-guarantee references.

To use Coq's rich support for inductive types, we require trusted user-provided definitions of relation folding, immediate reachability (without heap access) of references from a pure datatype, and relation containment. These are provided as typeclass instances. The definitions are fairly mechanical, and can be synthesized automatically for simple types if we extend Coq's support for inductive datatypes.

We also move some proof obligations such as stability, precision, and containment from type formation to allocation. This allows the definition of functions over ill-formed types, but such functions are unusable: only well-formed types may actually be constructed. This avoids some redundant proof obligations.

Our implementation is publicly available at `https://github.com/csgordon/rgref/`.

### 5.1 Proving Obligations with Dependent Types

RGref contains as a sublanguage the full Calculus of Constructions (CC). Specifically, it contains a full Pure Type System (PTS) with sorts $\mathcal{S} = \{\mathsf{Prop}, \mathsf{Type}\}$, axioms $\mathcal{A} = \{\mathsf{Prop} : \mathsf{Type}\}$, and product formation rules $\mathcal{R} = \{(s_1, s_2) \mid s_1, s_2 \in \mathcal{S}\}$ as formulated by Barendregt [2]. This sublanguage is part of the pure ($\Gamma \vdash e : \tau$) subset of RGref. Thus the language is amenable to embedding directly into CC with a few extensions (natural numbers, etc.) and RGref-specific axioms. Our technical report details the translation [23].

This provides us with an approach to proving predicate and guarantee obligations in a way well-integrated with the source language, justifying the use of proof terms in RGref's implementation. This also allows straightforward translation of proof goals from our typing derivations into Coq, where we can use tactic-based theorem proving to solve proof obligations.

The only subtleties arise from the fact that our embedding treats dereference as an uninterpreted function, allowing two potential inconsistencies. First, we permit recursion through the store, so applying a function read (via dereference) from the heap could result in an infinite loop; by treating dereference as an uninterpreted function in the embedding, this potential recursion is lost. The prototype may accept proofs about non-terminating terms. Second, there is a potential to equate dereference expressions that will be evaluated in different heaps (e.g., when a returned pure closure dereferences some reference). Our implementation only uses full equational reasoning for cases where all dereferences occur in the same heap, and otherwise abstracts terms with only their type.

### 5.2 Data Structure Design

We were able to use natural data structures for the examples in Section 3, but the types require careful consideration for propagating information through data flow. For example, the return type of `Cons` in Figure 1 must carry the additional refinement that the reference points to a cons cell whose tail is the `tl` argument, or the obligation to prove that the write prepends a cell in `doPrepend` is unprovable.

***From Simple Types to Inductive Types*** For the types whose splitting, folding, and containment we have examined formally (first-order data types, pairs, and references), the structure of the types is simple enough to provide a straightforward structural projection for each type. Much imperative code (e.g. in C, Java, etc.) heap-allocates similarly-simple structures. We have not worked out the theory for full inductive types. For types whose constructor arguments are not reflected as type indices, splitting and the like depend on the values passed to constructors, complicating the definitions for splitting because they then depend not only on type indices, but the actual value being potentially-split. This is an issue even though we expect to only require support for small inductive types. For self-referential datatypes, such as the list, we have only used guarantees for which folding is idempotent. In general, folding a restricted guarantee when dereferencing an datatype defined with concrete relations on "recursive" references is not expressible in the current system; if the guarantee on the recursive member changes, the result may not match a constructor of the type! Supporting this would require some sort of datatype-generic support, or a hybrid dereference-and-pattern-match to avoid directly representing not-quite-typed read results. We leave full support for inductive types to future work.

### 5.3 Proof Burden

RGref imposes proof obligations for precision, folding, containment, and guarantee satisfaction. For verifying the examples in Sections 3.1, 3.2, and 3.3, the proof burden is not substantially different from verifying the analogous pure-functional version. This section will call out which parts were particularly straightforward, as well as the few challenging aspects.

Precision obligations are typically easy to prove when they are true, as are the reflexive relation goals generated when references are used for reading: most are discharged by a simple induction, use of constructors for the relevant relation, and/or first order reasoning (e.g., Coq's `firstorder` tactic). When the goals are not true (e.g., a relation or predicate is not precise), the goal is simply not provable,

and it is up to the developer to recognize this. In this respect, RGREF is similar to verifying a functional program in COQ: a developer can waste time on unsolvable (false) proof goals.

In cases where relation folding is a no-op (e.g., $[R,G] \gg \tau = \tau$ as in the monotonic counter) folding goals are a simple matter of reduction and basic equality. In cases where relation folding is not a no-op (e.g., for references to references where the outer reference's guarantee bounds effects on other reachable objects), the folding obligations' complexity depends on the relations involved.

The most difficult proof obligations generated are those checking that heap writes satisfy the guarantee.[5] This is partly because these goals sometimes require reasoning about reads from an updated heap, in particular proving non-aliasing between references to different base types. Some goals are also complicated by non-identity folding results in types. In the formal system, we abstract away the mechanism for checking guarantee satisfaction through a denotational semantics for writes. Our implementation uses COQ's notation facilities for a sort of "punning," to duplicate expressions into two contexts with different semantics for dereference.

The normal program's use of the dereference expression chooses the appropriate relation folding type class instance, while the duplicated version used to check guarantee satisfaction is placed inside a context where a no-op fold instance overrides all others. This way the guarantee and predicates, which are specified as predicates over a type $A$, can be applied directly to $!x$ at type $A$ in the proofs instead of at a weakened type. The disadvantage of this approach is that the expression duplication also duplicates proof goals. Many of the smaller goals are automatically discharged when using COQ's PROGRAM extension, but because the generated goals are formed in slightly different contexts, the solved lemmas' proof terms have different arities, and are therefore not interchangeable in equalities. We encountered this twice, and solved it by using the proof irrelevance axiom on applications of the equivalent lemmas.

Because stability, reflexivity, and satisfaction results tend to be reused within a module and by clients, it should be considered proper practice for modules exporting a given API to also export most goals proven internally about properties of rely and guarantee relations, and predicates, as lemmas registered in a module-specific hint database. This is best practice for verifying purely functional programs in COQ as well; in general most of the useful habits in verifying functional programs can be reused in our implementation.

## 6. Future Work: Extensions and Adaptations

The type system we present in this paper has a few limitations (these are orthogonal to the implementation-specific limitations described in Section 5). We briefly mention a couple here, and discuss them in more detail in our technical report [23].

One weakness is that every individual write must fall within a guarantee relation. This means that a series of writes that individually violate a guarantee but in aggregate satisfy it are disallowed. This is a common challenge for verification techniques [3, 5, 41], and we expect related work to inspire sufficient solutions.

Also, as execution progresses, the actions permitted on an object through all references strictly decreases, by weakening and splitting guarantees. We have no way to strengthen guarantees based on additional information, as in the *recovery* technique [22] used to combine reference immutability with uniqueness, allowing uniqueness to be temporarily lost and then recovered. A resource-based approach to splitting, as in deny-guarantee program logics [16], suggests one promising approach to preserving possible actions.

[5] Not all are difficult; Section 3.1's guarantee obligation is discharged by automatic proof search with arithmetic hints: `auto with arith`.

Finally, it would be useful to separate out a logically-consistent fragment of the pure sublanguage (possibly using techniques like F*'s kind system [43]).

## 7. Related Work

The most closely related work falls into three categories: restricting mutation on a per-reference basis, techniques for reasoning about interference among threads (which can be adapted to interference among aliases), and dependent types for imperative languages.

***Alias-based Mutation Control*** Many techniques exist to control side effects by restricting actions through particular references. Notable examples include reference immutability [22, 44, 49, 50], and the owner-as-modifier interpretation of ownership and universe types [14]. Rely-guarantee references generalize reference immutability permissions (Section 3.3), allowing precise control over what modification is permitted through a given alias, not simply a choice between arbitrary mutation and local immutability. The fact that reference immutability is a special case of rely-guarantee references suggests a natural transition path from reference immutability to stronger verification guarantees. Developers could employ a "pay-as-you-go" model for verification, where a code base first transitions to using reference immutability (which need not be onerous [22]), and then gradually enrich the types for some parts of the system where more assurance is desired.

Typestate approaches typically use reference immutability like access permissions to control sharing of objects in a certain typestate, which is a weak form of refinement [5, 29, 31, 35]. Nistor and Aldrich describe a program-logic style type system [35] using abstract predicates [38] and connectives inspired by separation logic to specify *object propositions*, essentially an enriched typestate much closer to a full predicate logic; we believe object propositions and RGREF have similar expressivity in terms of what predicates they can verify. They pair access permissions with refinements to let aliases share coinciding views of an object's properties, and handle non-atomic updates. However, propositions on aliased objects cannot change over time and all aliases must have identical capabilities (simply preserving the proposition if the object is aliased), while RGREF allows asymmetric permissions and some strong updates to refinements even with aliasing.

Militão and Aldrich present a system that splits objects into substates that each carry their own typestate, and ways to merge particular substates into a typestate of the parent object [29]. Aliases to an object may exist and allow modification provided each reference is to a different substate, rather than to the full substate.

Concurrently with our work, they have proposed a rely-guarantee typestate [30], where the rely is a typestate each alias assumes an object may be in, and the guarantee is a typestate each alias is required to produce when unpacked. The properties they can verify are limited to those expressible as typestate (finite state partitioning) and the system requires the programmer to add many dynamic typestate checks to aid verification, but using their system is easier than the theorem-proving RGREF requires. RGREF can express stronger properties and forces no extra dynamic checks.

***Reasoning About Thread Interference*** Generally, any technique that can reason about interference among threads can be adapted to reason about interference among references. The most closely related system for reasoning about thread interference is rely-guarantee reasoning [27] described in Section 1.

The original rely-guarantee approach focused on global relations and assertions, hampering modularity. Several adaptations exist to treat interference over disjoint state separately. Vafeiadis and Parkinson integrated rely-guarantee reasoning with separation logic [45], allowing separation of state with linear resource semantics from shared state with interference. Feng later generalized this

to add separating conjunction of rely and guarantee conditions [18]. The conditions split into separate relations over separated pieces of shared state. Rely-guarantee references are heavily inspired by these approaches. However, RGREF allows substantial overlap among heap segments.

Dodds et al. adapted standard (non-separating) rely-guarantee reasoning to give resource semantics to rely and guarantee relations as assumptions in a context [16]. This allows the interference on shared state to change over time as permission to modify disjoint parts in particular ways is split, rejoined, and split again differently. This style of strong changes to the rely and guarantee over time could be adapted in a rely-guarantee reference system to allow the natural rely-guarantee reference generalization of the *recovery* technique of Gordon et al. [22], which allows recovering unique (or immutable) references from writable (or readable) references in a flow-sensitive type system given some constraints on the input context to a block of code.

Wickerson et al. [46] apply a modularized rely-guarantee logic to treat (non-)interference in the degenerate case of sequential access to the UNIXv7 memory manager. Related systems [15, 18, 45] could be applied similarly, but to our knowledge haven't. Most have only first-order treatment of interference. Only Concurrent Abstract Predicates [15] can (with some effort) store capabilities into the heap, while RGREF naturally supports this since mutation capabilities are tied to data. Our design closely follows a technique already shown successful in large-scale uses (reference-immutability [22]).

***Dependent Types for Imperative Code*** Many others have worked on integrating dependent types into imperative programming languages. Most take the approach of using refinement types [21, 43] that restrict modification to mutable data, but the refinements themselves may depend only on immutable data. Examples include DML [47], ATS [9], and X10's constrained types [36]. The refinement language is often also restricted to some theory that can be effectively decided by an SMT solver, as in Liquid Types [41]. RGREF allows refinements to depend on mutable heap data, and does not artificially restrict the properties that can be verified (at the cost of requiring manual guidance for proofs).

A notable approach to dependent types in imperative code is Hoare Type Theory (HTT) [32, 33] and its implementation YNOT [11, 34]. HTT uses a monadic Hoare Triple to encode effectful computation. It allows using effectful code in specifications: it decides equality of effectful specification terms by using canonical forms where traditional dependent type systems use β-conversion. This approach could be adapted for rely-guarantee references as well. YNOT implements the core ideas of HTT as a domain specific language embedded in COQ. It supports traditional Hoare logic specifications and, by embedding, separation logic specifications as well. A later version [11] builds a family of targeted proof search tactics that can automatically discharge most separation logic proof goals generated while typechecking YNOT programs. We modeled our implementation after YNOT, and are currently building a library of combinators and transformers in hopes of supporting similarly robust automatic proof discharge.

HTT (and separation logic in general) support proving functional correctness rather than the somewhat weaker safety properties verified by rely-guarantee references (and most other stable-assertion-based approaches [18, 27, 45]). But this comes at the cost of specifications explicitly specifying aliasing constraints through choice of standard or separating conjunction. Separation logic specifies the behavior of code, not the restrictions on data transformation. Rely-guarantee reference types specify the possible evolution of data in the description of data itself. This means that assumptions and permission to modify state follow data-flow, rather than the control-flow-centric passing of assertions in most program logics.

## 8. Conclusion

We have introduced *rely-guarantee references*, an adaptation of rely-guarantee program logics to reasoning about interference among aliases to shared objects. The technique generalizes reference immutability, connecting two previously-separate lines of research and addressing a fundamental problem in verifying imperative programs. We have shown the technique's usefulness by verifying correctness for several small examples (which are difficult to specify or verify with other approaches) in a prototype implementation. Our experience suggests that at least for small examples, the proof burden is reasonable. Rely-guarantee references demonstrate that aliasing in program verification can be addressed by adapting ideas from reasoning about thread interference.

## Acknowledgments

This material is based on research sponsored by NSF grants CNS-0855252 and CCF-1016701 and by DARPA under agreement numbers FA8750-12-C-0174 and FA8750-12-2-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. We thank the anonymous referees for their comments, which helped improve the paper.

## References

[1] L. Augustsson. Cayenne — A Language with Dependent Types. In *ICFP*, 1998.
[2] H. Barendregt. *Lambda Calculi with Types*. 1991.
[3] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. *Commun. ACM*, 54(6):81–91, June 2011.
[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
[5] K. Bierhoff and J. Aldrich. Modular Typestate Checking of Aliased Objects. In *OOPSLA*, 2007.
[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.
[7] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *POPL*, 2005.
[8] V. Capretta. A Polymorphic Representation of Induction-Recursion. Retrieved 9/12/12. URL: `http://www.cs.ru.nl/~venanzio/publications/induction_recursion.pdf`, March 2004.
[9] C. Chen and H. Xi. Combining Programming with Theorem Proving. In *ICFP*, 2005.
[10] A. Chlipala. *Certified Programming with Dependent Types*. `http://adam.chlipala.net/cpdt/`.
[11] A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective Interactive Proofs for Higher-order Imperative Programs. In *ICFP*, 2009.
[12] Coq Development Team. The COQ Proof Assistant Reference Manual: Version 8.4, 2012.
[13] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.
[14] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, 2007.
[15] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, 2010.
[16] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *ESOP*. 2009.
[17] P. Dybjer. Inductive Families. *Formal Aspects of Computing*, 6:440–465, 1994.
[18] X. Feng. Local Rely-Guarantee Reasoning. In *POPL*, 2009.
[19] C. Flanagan and M. Abadi. Types for Safe Locking. In *ESOP*, 1999.
[20] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.
[21] T. Freeman and F. Pfenning. Refinement types for ml. In *PLDI*, 1991.

[22] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, 2012.

[23] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-Guarantee References for Refinement Types Over Aliased Mutable Data (Extended Version). Technical Report UW-CSE-13-03-02, University of Washington, March 2013.

[24] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[25] M. Hofmann. *Syntax and Semantics of Dependent Types, in Semantics and Logics of Computation*, chapter 3. 1997.

[26] J. B. Jensen and L. Birkedal. Fictional Separation Logic. In *ESOP*, 2012.

[27] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM TOPLAS*, 5(4):596–619, Oct. 1983.

[28] K. R. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, 2009.

[29] F. Militão, J. Aldrich, and L. Caires. Aliasing Control with View-based Typestate. In *FTfJP*, 2010.

[30] F. Militão, J. Aldrich, and L. Caires. Rely-Guarantee View Typestate. Retrieved 8/24/12, July 2012. URL `http://www.cs.cmu.edu/~foliveir/papers/rgviews.pdf`.

[31] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A Type System for Borrowing Permissions. In *POPL*, 2012.

[32] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and Separation in Hoare Type Theory. In *ICFP*, 2006.

[33] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *ESOP*. 2007.

[34] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In *ICFP*, 2008.

[35] L. Nistor and J. Aldrich. Verifying Object-Oriented Code Using Object Propositions. In *IWACO*, 2011.

[36] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained Types for Object-Oriented Languages. In *OOPSLA*, 2008.

[37] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, pages 319–340, 1976.

[38] M. Parkinson and G. Bierman. Separation Logic and Abstraction. In *POPL*, 2005.

[39] C. Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. In *Typed Lambda Calculi and Applications*, 1993.

[40] A. Pilkiewicz and F. Pottier. The Essence of Monotonic State. In *TLDI*, 2011.

[41] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-Level Liquid Types. In *POPL*, 2010.

[42] M. Sozeau. Program-ing Finger Trees in Coq. In *ICFP*, 2007.

[43] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP*, 2011.

[44] M. S. Tschantz and M. D. Ernst. Javari: Adding Reference Immutability to Java. In *OOPSLA*, 2005.

[45] V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*. 2007.

[46] J. Wickerson, M. Dodds, and M. Parkinson. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In *ESOP*, 2010.

[47] H. Xi and F. Pfenning. Dependent Types in Practical Programming. In *POPL*, 1999.

[48] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. In *POPL*, 2003.

[49] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability Using Java Generics. In *ESEC-FSE*, 2007.

[50] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. In *OOPSLA*, 2010.