Verifying Concurrent Programs by Controlling Alias Interference

Colin S. Gordon

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Michael D. Ernst, Chair

Dan Grossman, Chair

Matthew Parkinson, Microsoft Research, Cambridge, UK

Program Authorized to Offer Degree:
UW Computer Science and Engineering

University of Washington

Abstract

Verifying Concurrent Programs by Controlling Alias Interference

Colin S. Gordon

Co-Chairs of the Supervisory Committee:
Professor Michael D. Ernst
Computer Science and Engineering

Associate Professor Dan Grossman
Computer Science and Engineering

This dissertation proposes a family of techniques for static verification of sequential and concurrent imperative programs by leveraging fine-grained characterizations of mutation. The key idea is that by attaching to each reference in a program (1) a restriction on mutations permitted using that reference, and (2) a characterization of possible interference through other aliases, a type system can reason about what properties are preserved by all mutations in a program.

This thesis develops four variations on this idea: (1) We adapt *reference immutability* to support data-race-free concurrent programming. (2) We generalize reference immutability to *rely-guarantee references*, allowing two-state invariants to express usage restrictions between read-only and arbitrary mutation. (3) We extend rely-guarantee references to prove invariants and functional correctness of lock-free concurrent data structures. (4) We evaluate rely-guarantee references' utility for existing Haskell programs.

Together these variations show that reasoning about aliasing and reasoning about concurrent (imperative) programs are the same fundamental challenge, and that by taking the right foundational approach to reasoning about sequential programs, the gap to reasoning about concurrent programs is significantly reduced.

TABLE OF CONTENTS

Page

List of Figures iv

Chapter 1: Introduction 1
1.1 Modular Verification: Simple, Pure, and Imperative 2
1.2 Protocols for Shared Mutable State 5
1.3 The Granularity of Modularity 7
1.4 Explanation of Thesis 8
1.5 Published Content 10

Chapter 2: General Related Work 12
2.1 Global Reasoning 15
2.2 Separation and Isolation 16
2.3 Isolation with Serialized Sharing 24
2.4 Read-sharing 27
2.5 Interference Summaries 32
2.6 Implementing Verification Systems 43
2.7 Relating Sequential and Concurrent Correctness 44
2.8 Dependent Type Theory 45

Chapter 3: Uniqueness and Reference Immutability for Safe Parallelism 49
3.1 Reference Immutability, Uniqueness, and Parallelism 51
3.2 Types for Reference Immutability and Parallelism 58
3.3 Type Soundness 66
3.4 Polymorphism 84
3.5 Evaluation 95
3.6 Related Work 106
3.7 Conclusions 112

Chapter 4: Rely-Guarantee References . . . 113
4.1 Introduction . . . 113
4.2 Rely-Guarantee References . . . 116
4.3 Examples . . . 119
4.4 A Type System for Rely-Guarantee References . . . 128
4.5 Implementation . . . 142
4.6 Future Work: Extensions and Adaptations . . . 149
4.7 Related Work . . . 150
4.8 Conclusion . . . 154

Chapter 5: Concurrent Refinement Types and Functional Correctness via Rely-Guarantee References . . . 162
5.1 Correctness for Lock-Free Datastructures . . . 164
5.2 Suitability of RGREFs for Concurrent Programming . . . 166
5.3 RGREFs for Concurrency . . . 168
5.4 Refinement from Rely-Guarantee Reasoning . . . 196
5.5 Implementation . . . 203
5.6 Related Work . . . 204
5.7 Conclusions . . . 208

Chapter 6: Rely-Guarantee References for Existing Code . . . 210
6.1 Background: State and Refinement Types in Haskell . . . 212
6.2 Embedding Rely-Guarantee References in Liquid Haskell . . . 217
6.3 Defining Liquid RGREFs . . . 222
6.4 Using Liquid RGREFs . . . 228
6.5 Related Work . . . 239
6.6 Fruitful Directions for Further Investigation . . . 240
6.7 Conclusions . . . 241

Chapter 7: Directions for Future Work . . . 243
7.1 Direct Extensions to RGREFs . . . 243
7.2 Richer Specification Languages . . . 248
7.3 Gradual Verification . . . 249

Chapter 8: Conclusion . . . 251

Bibliography . . . 253

Appendix A: Background Reading 278
A.1 Dependent Types 278
A.2 Program Logics: Hoare Logic, Separation Logic, and Beyond 279

LIST OF FIGURES

Figure Number — Page

2.1 Selected proof rules for sequential and concurrent separation logic, variable side conditions elided. . . . 18

3.1 External uniqueness with immutable out-references. . . . 52

3.2 Qualifier conversion/subtyping lattice. . . . 53

3.3 Core language syntax. . . . 60

3.4 Subtyping rules 61

3.5 Core typing rules. . . . 61

3.6 Program typing 62

3.7 Recovery rules 63

3.8 Type rules for safe parallelism. IsoOrImm is defined in Figure 3.7 64

3.9 Typing derivation for adding a node to an isolated doubly-linked list. . . . 66

3.10 Definition of Well-Regioned 69

3.11 Denoting types and type environments. . . . 72

3.12 The thread interference relation $\mathcal{R}_0$. . . . 74

3.13 Weak type environment denotation, for framed-out environments. The differences from Figure 3.11 are the permissions required by readable and writable references, and the way π bounds the state's permissions. . . . 78

3.14 Method call extensions to the language and program logic. . . . 80

3.15 The proof tree for validity of method summaries when the summary is not for an isolated receiver. Environment reordering steps are not shown explicitly. Parentheticals on the right hand side indicate the method by which the assertion on the same line was proved from the previous assertion and/or statement when it is not an obvious derivation like framing. . . . 85

3.16 Grammar extensions for the polymorphic system. . . . 86

3.17 Auxiliary judgements and metafunctions for the polymorphic system. . . . 87

3.18 Generics typing rules. All structural, recovery, and parallelism rules remain as in Figures 3.5, 3.7, and 3.8, extended with Δ in conclusions and appropriate hypotheses. . . . 90

3.19 Well formed polymorphic programs. In T-Method*, the bound on the method permission p is present if and only if p is a permission variable. 91

3.20 Denoting types and type environments in the polymorphic system, differences from Figure 3.11. The main change is to the subclass clause of the writable and isolated cases, which require the runtime type's field types to exactly match the supertype's. 93

3.21 A simplified collection with a polymorphic enumerator. 100

3.22 Extension methods to add regular and parallel map to a linked list. 102

4.1 RGref code for a prepend-only linked list. 123

4.2 Combining reference immutability permissions, from [105]. Using a p reference to read a $q\ T$ field produces a $(p \triangleright q)\ T$. 126

4.3 Syntax, omitting booleans ($b : \texttt{bool}$), unary natural numbers ($n : \texttt{nat}$), 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 [14] pseudoterms. 130

4.4 Typing. For standard recursors for naturals, booleans, pairs, and identity types [129], as well as well-formed contexts, and (pure) expression/type conversion ($\Gamma \vdash \tau \rightsquigarrow \tau$), see (Figure 4.6). 155

4.5 Typing, continued. 156

4.6 Selected auxiliary judgments 157

4.7 Auxilliary functions 158

4.8 The simplest term we can imagine that observes the multiple-dereference reduction 158

4.9 Values, typing of runtime values, and main operational semantics for RGref. 159

4.10 Structural / context operational semantics for RGref. 160

4.11 Dynamic state typing. 161

5.1 A positive monotonically increasing counter using RGrefs, in our Coq DSL. The `rgref` monad for imperative code is indexed by input and output environments Δ of substructural values, not used in this example. 171

5.2 Atomic increment for a monotonically increasing counter. This code uses a standard read-modify-CAS loop. Until the increment succeeds, it reads the counter's value, then tries to compare-and-swap the old value with the incremented result. `RGFix` is a fixpoint combinator, and `rgret` returns a value. 171

5.3 A Treiber Stack [251] using RGrefs, omitting proofs. The relation `local_imm` (not shown) constrains the immediate referent to immutable; `any` is the always-true predicate. 172

5.4 A lock-free union find implementation [7] 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, making it a safe index into the array. $\sqcap$ conjoins predicates. 174
5.5 Core typing rules for concurrency-safe RGRefs. See also Figure 5.7 for metafunction definitions. 178
5.6 Primitive to introduce new refinements based on observed values. 179
5.7 Metafunctions for side conditions used in Figure 5.5. 179
5.8 Views state space. "Hats" (e.g., $\widehat{M}$) indicate explicit annotations of RGRef components on locations, while the erased ($L\lfloor - \rfloor$) omits this. 185
5.9 Embedding into the Views Framework. We lift stack lookup ($S(-)$) to terms as well, substituting stack contents for variables. 186
5.10 Views Framework parameter instantiation, according to Dinsdale-Young et al. [70]. 191
5.11 Predicates for ensuring call-by-value reduction of pure terms (1) accesses the heap at most once, and (2) produces a value with no deferred dereferences. . 195
5.12 Relational program traces, thread-specialized concrete program traces, trace abstraction, and selected trace equivalence and refinement rules. $\mathcal{T}$ and $\mathcal{C}$ are coinductive, modeling possibly-infinite execution traces. 200
5.13 Tracing type judgments. We describe the algorithm by traversal over a well-typed AST, assuming we can trivially map from the AST to the associated fragment of a typing derivation. 202
6.1 Example usage of Liquid Haskell axioms. 216
6.2 RGRefs in Liquid Haskell 223
6.3 A sequential monotonically increasing counter implemented in Liquid Haskell + Liquid RGRefs. 225
6.4 A lock-free increment for a monotonically increasing counter implemented in Liquid Haskell + Liquid RGRefs. 226
6.5 Excerpt from a lock-free linked list implemented using Liquid RGRefs: definitions. 233
6.6 Excerpt from a lock-free linked list implemented using Liquid RGRefs: deletion. 236

ACKNOWLEDGMENTS

I have lost track of how many times I have written and deleted these acknowledgements, convinced that through some incredible feat of eloquence I could somehow do justice to the thanks deserved by those who helped me reach this point. I've now resigned myself to the impossibility of this task, and what follows is but a poor approximation of the thanks deserved.

My co-advisors, Dan Grossman and Mike Ernst have been invaluable sources of encouragement, guidance, and insight. Most doctoral students do not stray as far from their advisors' expertise as I have, and my success in doing so is a testament to their advising ability. I have benefited greatly from their perspectives on research ideas, approaches, writing, and more. It is literally not possible to list all of the ways they have helped me develop as a researcher and a person.

Matthew Parkinson, my final committee member, provided not only encouragement and guidance, but introduced me to new areas of work that I would not have found time or courage to explore in depth otherwise. My resulting exposure to concurrent program logics provides one of the key ingredients for this dissertation.

The road to this particular thesis began with a fruitful collaboration with my former colleagues at Microsoft, particularly Joe Duffy, Jared Parsons, and Aleksandra Culver, who brought me onboard to work on a new programming language at Microsoft for the summer. This work, beyond producing a publication that forms the basis of Chapter 3, deeply colored my view on program verification, in ways I'm still just figuring out.

Niki Vazou, Eric Seidel, and Ranjit Jhala also deserve my thanks, not only for building Liquid Haskell (upon which I base a portion of my dissertation), but for fixing what seems like an endless stream of minor bug fixes and change requests I sent them, usually within a day, all while insisting that *I* was helping *them*. This is far more support than one can

typically assume for research software, and without their assistance the final evaluation of this dissertation would be sorely lacking.

No graduate school experience is complete without fellow students and post-docs whose commiseration, feedback, encouragement, distraction, and friendship are often present at just the right moment. For this I'd like to thank, in no particular order, Todd Schiller, Brian Burg, Ben Lerner, Brandon Lucia, Tom Bergan, Ben Wood, Kivanc Muslu, Werner Dietl, Yuriy Brun, Ivan Beschastnikh, Laura Effinger-Dean, Matt Kehrt, Marius Nita, Konstantin Weitz, Sai Zhang, Gilbert Bernstein, Pete Hornyack, Brandon Holt, Adrian Sampson, and Darioush Jalali.

I've benefited from the influence and encouragement of many other faculty members over time, including Shriram Krishnamurthi (who lured me into program verification as an undergraduate), Maurice Herlihy, Tom Doeppner, Steve Gribble, Zachary Tatlock, and of course the late David Notkin.

Last but far from least, I owe endless thanks to my supportive wife Natalie, who tolerated both my physical and mental absences in the course of this work, without whose support I never would have finished this work.

DEDICATION

to my wife, Natalie, and son, Henry

Chapter 1

INTRODUCTION

> Aliasing is only the simplest case of the more general phenomenon of interference.
>
> John C. Reynolds, *Syntactic Control of Interference* [223].

Today's software is remarkable in both size and inherent complexity. Both of these factors exacerbate a problem that exists even in small and relatively simple programs, which is ensuring that a program is *correct*: that relative to some *specification* of its expected behavior (whether full functional input-output behavior, or more modest properties like the absence of crashing errors), the software behaves as intended.

To cope with both size and complexity, modular [209] software development has been proposed. A program is decomposed into some number of *modules*: independently organized units of functionality, with clearly documented interfaces for client code to interact with the module's functionality. The goal for this decomposition is generally to produce modules limited in size and complexity. In theory and in practice, this decomposition helps to reduce error rates in software, by proscribing a development approach that often localizes mistakes, and clarifies which part of the program is "to blame" in the event of an error.

No approach to program development is a panacea, modular design included. In practice, the internal implementation of a module may remain quite complex, and the interfaces between modules may be complex as well. This leads to the natural desire for tools supporting automatic verification that up to the bounds of some formal specification of a module's correctness and interface, the module (program) is correct.

The core challenge in supporting automatic formal verification of a modular program is in ensuring that not only is the program modular, but that the specification is as well,

for all ways developers may compose program components. For imperative programs, the specifications used are typically only valid for *sequential* composition — totally ordered interleaving of different program components, with the interleaving explicitly chosen by the program. Supporting *parallel* composition of imperative program components — executing components simultaneously, potentially with simultaneous arbitrarily interleaved access to shared structures — generally requires adding additional layers of specification used only for coordinating parallelism and concurrency, which serve no value for sequential composition. Thus verifying concurrent programs currently requires enriching specifications and reasoning with many additional constructs, creating a significant gap in required effort and background between verifying sequential and concurrent programs. This difference is typically viewed as fundamental. The main contribution of this dissertation is the proposal and study of a specification style that is sensible for both styles of composition, relying on exactly the same verification concepts, with only minor differences, for the sequential and concurrent programs. This shows that while different, verification for concurrent programs is *not fundamentally harder* than sequential verification, given the right approach to specification. Ultimately, we wish to support reasoning about program components interacting through shared state, in a way that is agnostic to the particular interleaving of program components' actions.

Before explaining how we develop our contribution, we first give a brief overview of the styles of specification and composition that have been well-explored, and those that show promise for both sequential and concurrent programs.

1.1 Modular Verification: Simple, Pure, and Imperative

Static type systems offer a flexible approach to specifying and checking some forms of correctness and correct use of module interfaces. In the simplest cases, a simple type system ensures that code only applies operations sensible for a given data (for example, prohibiting the accidental application of (machine) integer addition to a string and a boolean). This is the level of modularity ensured by type systems like Java or C#. Somewhat more sophisticated systems [176, 160, 161, 231, 4] can allow a developer to state a desired modular architecture, including holding some representations abstract [224, 222] from other modules,

and statically check that this discipline is respected. The canonical example of this is the module system of Standard ML [176]. But these systems, while preventing a significant number of errors, are not sufficient to ensure a module does what it should; they only ensure the absence of specific classes of runtime errors (a well-typed program does not "go wrong" [175] or "get stuck" [270]), and ensure that representation changes in one module that preserve functionality are unobservable to other modules [224]. But these simple cases ensure only correctness properties common across all programs. To allow the statement and enforcement of richer, program-specific specifications, a variety of approaches have been explored in the context of pure functional programs and imperative programs, with different trade-offs in terms of the types of programs supported or how complex the specifications must be. Ideally we would support specifications well-integrated into programs, as well as rich reasoning about how mutable state changes *over time*.

In the case of code that does not use mutable state, particularly rich type systems can even ensure full functional correctness of a module. Dependent type theories [165, 62, 60, 114, 113, 248, 14, 166] may exploit the Curry-Howard Correspondence [130, 239] which identifies a proposition with a type and a proof of a proposition with a term of appropriate type.[1] This gives rise to a situation where a program and proofs about it are written in the same language [164, 197], so verification need not introduce any additional concepts.

But this theory is only well-developed for programs that do not use mutable state, while most programs written today make heavy use of state in programming. In this setting, mutation of data structures shared across modules (or even within modules) forms an implicit communication channel between program components. This communication channel carries its own interface for proper communication, but these can be even more complex than the input-output interfaces in the functional case due to the addition of mutation: part of the interface for these structures and modules is expectations on how a structure may (or may not) change over time (for example, a counter may be required to only increment). This is a temporal property, typically ignored or left informal in specifications for

[1]This is occasionally referred to as the Curry-Howard-Lambek Correspondence, pointing out that both types and logical systems have categorical analogues as well — type theories may be used to formalize and reason in the internal language of a category [146, 136].

imperative programs. Worse, there is often an implicit protocol for using structures where different parts of the program are expected to mutate the state differently. The problem is then exacerbated by the presence of aliasing: not only might state be mutated, but these mutations might occur through different names for the structures, adding a new layer of complexity to an already very difficult problem. So the specifications for communication via mutable memory must address (1) invariants for what it means for a structure to be valid, (2) temporal restrictions on use and modification of a structure, and (3) specifying that different parts of the program (modules, routines) may be required to access the structure in different ways (play different roles in the structure's protocol, such as distinguishing producers and consumers for a work queue). This last point implies that (4) different program components may have different invariants they care about. Checking that a specification is followed by a program requires additionally resolving questions of whether actions through one reference might affect another because they are aliased. Adding shared memory concurrency to the mix complicates all of the above challenges, as well as adding its own by adding non-determinism to the interleaving of program actions.

In practice, developers often do not record these specifications for shared data structures. This makes it very easy for a developer to accidentally violate the assumptions of some remote program fragment by writing code that mutates a structure, then calls into a module expecting the same structure to be in a different state. Current mainstream programming languages offer no support for stating or checking whether some mutable structure is shared, and if so, what the permitted modifications are. The ease of introducing these accidental assumption invalidations is so frequent that there are names for common classes of these bugs: representation exposure [67], protocol violations [24], and data races, to name only a few. The desire to reason correctly about program components interacting through shared state has driven decades' worth of work into type systems, program logics, and other areas.

Program logics, beginning with Floyd [91] and Hoare [126], were designed explicitly to reason about the functional correctness of these *imperative*, *stateful* programs, using so-called triples to build a proof theory over pre- and post-conditions in imperative programs: $\{ P \} C \{ Q \}$ is derivable in such a logic only if executing the command C in a state satisfying assertion P will cause no runtime faults and produce a state satisfying assertion Q (or,

in the most common variants, produce a state satisfying Q or fail to terminate). The approach has proven useful for program specifications as diverse as reasoning about exact heap structure [221] or even file system contents [98], and these proof techniques have been applied to truly substantial, complex systems [143].

1.2 Protocols for Shared Mutable State

Imperative program logics focus solely on specifying the *behavior of code*, rather than *protocols for use of state*. For straight-line imperative code, within a single module, mutating only data structures owned wholly by that module, this code-centric focus is generally sufficient. For richer software designs, this is inflexible. In practice, software often shares mutable structures across multiple modules, each module modifying the data structure in a specific manner. The focus on code-centric verification requires that any properties of these shared structures are carried through a program in pre- and post-conditions. Beyond cluttering proofs, this obscures some specifications and makes others difficult or impossible to express. For example, if a counter should only be incremented, there is nothing in standard Hoare logic or separation logic that prevents the counter from being passed into a decrement function! This moves the burden of ensuring that a data structure is used consistently *across all modules accessing it* to humans: humans must verify that the modules verified for pre- and post-conditions never violate the protocol internally [201].

Consider a data structure shared across modules: a work queue, with one module enquing work for the other (a producer-consumer scenario). In this case, it is desirable to specify that each module only enqueues or dequeues (neither module does both). This is a strong module specification; it says that each module will only perform one type of action, ever, even if it later reverses some effects. This is a specification of *how to access data*, *temporally*, *per point of access*. In standard modular Hoare logics (e.g., separation logic with modules [202]), this is difficult to specify because the only special accommodation the modular frameworks make is for hiding state private to one module, and satisfying a fixed invariant upon entry/exit to the module. Thus for one of these temporal protocols for state shared across multiple modules, one must specify that each operation of the producer module has as (part of) its precondition that the queue exists with some elements, and as

(part of) its postcondition that the queue contains the original elements plus more at the end. The consumer specification is dual to the producer. This specification is spread across a whole module, when it is really about one data structure. Checking that this desired specification is enforced requires examining many specifications.[2]

So modular reasoning about the correctness of a program requires some form of temporal reasoning, and some support for exposing different, cooperating notions of expected temporal behavior, sufficient for proving the program invariants and behavioral properties in specifications.

One classic notion of cooperating protocols for shared state is *rely-guarantee reasoning* [139], which allows giving *relational* specifications constraining how each state update changes the state. This is an idea from concurrent program verification where each thread of a program may be verified independently from others using rely and guarantee relations: binary relations on program states constraining mutation. Each time a thread modifies the program state, rely-guarantee reasoning requires that the old and new states be in the thread's guarantee relation. Thus the guarantee is an upper bound on local behavior. The rely is the reverse; it is a sound summary of other threads' behavior, which is sufficient to reason about what properties are preserved by other threads' actions. Those preserved properties are called *stable*: for example, if other threads may at most increment a shared counter, then the assertion that the counter is positive is stable (preserved by any possible action of other threads). By limiting verification to stable properties, rely-guarantee reasoning can verify invariants of thread-shared data structures. Crucial to this approach is the idea that the rely is a *sound* (accurate) summary of other threads, which is enforced by checking, at the time concurrency is introduced, that each thread's guarantee (possible local actions) is a subrelation of each other thread's rely (summary of other threads possible actions). We call this a check for *compatibility*, but this is not standard terminology.

Rely-guarantee reasoning offers temporal reasoning (two-state invariants) about state changes, including exposing role-specific views on this protocol (differentiating rely and

[2]It may be tempting to call this an object invariant, but it is neither universal across all operations, nor an invariant. A single queue may be subject to both enqueues and dequeues, and object invariants do not relate pre- and post-states.

guarantee). It includes a way to ensure two local views of a protocol agree in some way (compatibility). And it can prove important program properties.

1.3 The Granularity of Modularity

Unfortunately, rely-guarantee reasoning as originally conceived presumes that one wants to verify threads individually, but with full knowledge of all possible program states (the rely and guarantee constrain global views of program state). Over the years, rely-guarantee reasoning has been adapted numerous ways to add slightly different forms of modularity [259, 260, 83, 82, 71, 70, 242, 73, 157]. But with only one exception [269], each of these focuses on *thread modularity* and/or assumes that modules are strictly specified and privately own their state. These systems are useful, but only for specific modular decompositions. Part of the reason for this is that rely-guarantee reasoning grew out of an attempt to tame thread interference [204], the pervasive problem of threads using shared memory concurrency unexpectedly modifying thread-shared data structures.

This focus on threads ignores the fact that interference is not only an issue for concurrent programs, but for sequential programs as well. It also ignores that at different points of time, developers may wish to reason about modularity at different granularities — between modules, procedures, or even segments of the same procedure. The scope at which interference and protocols must be applied is also important: sometimes different components may be looking at interference over the same segment of state, but at other times one module may be considering interference on only part of a larger structure.

Aliasing of shared mutable state plays another role in considering modularity and interference: state mutations made through different aliases interfere the same way as actions through a single global name. Aliases also serve a unique role: there is no finer decomposition of shared mutable state than an individual alias to (part of) a shared structure. So if we can reason about the most fine-grained modular decompositions (aliases) and relate these decompositions (e.g., relating the actions performed through an alias to the interior of a data structure to the aggregate effects on the overall structure), we can use this to reason modularly at *any* granularity — thread, module or procedure summaries can be reconstructed from precise reasoning about alias usage.

1.4 Explanation of Thesis

A consistent challenge in the verification of imperative programs — both sequential and concurrent — is reasoning about the combination of mutation and aliasing. Specifically, given two names for mutable program resources, how does mutation through one name affect observations through the other? Or stated differently, given two program components viewing the same shared state through aliases, how do we ensure that the components' expectations about the shared state are compatible with each other? This core challenge has driven a wide variety of research across type systems, program logics, points-to analyses, and even dynamic analyses. This thesis offers new techniques for reasoning about the combination of mutation and aliasing, in a way that spans a variety of choices for the granularity of reasoning.

The core technique studied in this thesis is *reference-based mutation control* — the notion of coupling every heap reference in a program with both a restriction on use (alternatively, permissions for certain actions) and a summary of restrictions on actions through aliases (an *interference summary*). As long as those summaries are sound (no reference's permissions permit actions not assumed by an alias's interference summary), this supports sound reasoning in the presence of mutation and aliasing, without requiring strict tracking of alias locations or heap shape. Further, the same specifications make sense in both the sequential and concurrent settings, and proofs using reference-based mutation control are nearly the same in sequential and concurrent programs, in contrast to the standard assumption that concurrent program verification is fundamentally harder than sequential verification. By exploring these ideas in depth, we make the case that

> **Reasoning about *interference* between individual *aliases* is a useful and powerful verification approach, and reduces the gap between verification of sequential and concurrent programs.**

After discussing related work and themes in the treatment of mutation and aliasing in Chapter 2, we develop a series of more powerful systems for reference-based mutation control, spanning both sequential and concurrent systems, and evaluate a simplified version

on real codebases:

- Chapter 3 further develops the existing concept of *reference immutability* [28, 252, 275, 276] to ensure data race freedom. This includes a novel treatment of conversion among various mutability restrictions, a precise treatment of generics for mutability permissions, and a discussion of a Microsoft team's experience using a prototype C# extension using these concepts over an extended period of time.

- Chapter 4 generalizes (sequential) reference immutability to allow describing *how* state may be changed using a given reference, rather than merely *whether* state may be changed (as in reference immutability). This system, *rely-guarantee references* (RGREFs) enables the statement of refinement types over segments of the heap that are preserved by interference from other aliases, offering the most general combination of refinement types and mutation to date. This work blends ideas from concurrent program verification into reference-based mutation control, requiring solutions to new issues due to nested references.

- Chapter 5 adapts the system of Chapter 4 for fine-grained concurrency. It makes the system usable for proving invariants of lock-free data structures, and builds a trace-refinement system for proving functional correctness on top of the semantic information embedded in rely-guarantee references. It also offers a new soundness proof for rely-guarantee references by embedding into the Views Framework [70], giving a slightly more denotational account of RGREFs.

- Chapter 6 explores a restriction of RGREFs to a fragment that is more likely suitable for actual developers, analyzing experience converting existing Haskell programs to use rely-guarantee references for enforcing invariants.

Finally, Chapter 7 describes promising directions for future work, and Chapter 8 synthesizes lessons learned from across these systems.

This thesis also explores, informally, the notion of *gradual verification*. As opposed to the typical approach to verification where the guarantees of an analysis on some program mod-

ule are sound only when composed with other fully-verified code, the systems proposed here support gradually refining well-typed but otherwise unverified code for additional guarantees. In principle it should be possible to gradually evolve a program from type-safe unverified code to some use of reference immutability, and on to some use of rely-guarantee references, taking the switch to concurrent programing at any time, without needing to verify precise specifications for the entire program at once. We leave thorough evaluation of this development approach to future work, but show that it is at least technically feasible.

1.5 Published Content

This dissertation builds on the content of two peer-reviewed conference papers.

1. Chapter 3 extends the material from an OOPSLA paper [105].

   Changes from Conference Version This presentation reorders some material, and inlines what were previously technical report [106] appendices on method and generics soundness into the main presentation flow. Additionally, it provides some small updates on the experiences of a Microsoft team building a large system using it.

2. Chapter 4 extends the material from a PLDI paper [103].

   Changes from Conference Version This presentation again reorders some material from the conference version, and inlines part of a technical report containing soundness proofs [104]. It has also been extended with an additional example highlighting the use of (read-only) higher-order store in the pure fragment of the language.

3. Chapter 5 extends material from a draft paper.

The material in Chapter 6 is entirely new to this dissertation. My contributions to these papers were as follows:

1. I distilled an implemented production language prototype's type system to a manageable core calculus representative of the actual system, including formalizing precise

generics over mutability permissions, and proved that the type system enforces its intended properties. I also formulated a novel form of borrowing and permission conversion. I wrote the entire article, which received additional editing from Matthew Parkinson. I contributed to the implementation, including some work on handling receiver permissions for methods and properties. I also prototyped an integration of the reference immutability system with a Code Contracts style contract system, which was not discussed in the initial paper due to disclosure timing issues, but is discussed briefly here.

2. I conceived of the research independently, performed the research on my own, implemented a prototype as a DSL in the Coq proof assistant, and wrote the article (which received edits from Dan Grossman and Michael Ernst).

3. I conceived of the research independently, performed the research on my own, implemented a prototype in the Coq proof assistant, and wrote the draft paper (which received edits from Dan Grossman, Michael Ernst, and Matthew Parkinson).

Funding

The work described in this thesis was supported at various points in time by a Microsoft internship, NSF grants CCF-1016701 and CNS-0855252, and DARPA contracts FA8750-12-C-0174 and FA8750-12-2-0107.

Chapter 2

GENERAL RELATED WORK

This chapter surveys work on static program verification that is closely related to our proposal. We organize our discussion around a set of themes that arise in the related work, on schools of approaches to reasoning about mutation and aliasing that span both sequential and concurrent program verification. The identification of these themes is itself a contribution of this thesis, because unlike other overviews of program verification, we focus on themes that are common across both sequential and concurrent program verification, with applications from the literature in both cases.

Invalidation of remote assumptions is a pervasive problem in modern software. Well-known software bug patterns are often instances of this, including protocol violations [24], representation exposure [67], and data races. The challenges of reasoning about remote actions' effects on static assumptions have driven a wide variety of work in program logics, type systems, and other areas. But this work falls broadly into a relatively small set of technique families, guided by different intuitions on non-local effects. Broadly, these techniques — surveyed in the remainder of this chapter — are:

Global Reasoning (§2.1) These techniques attempt to reason about global state, either by making global assertions, or restricting assertions enough that they can never be invalidated. Examples include Hoare Logic [126] and systems with fixed assumptions (like the fixed content type of the heap cell referred to by an ML reference).

Separation and Isolation (§2.2) These techniques partition state, preventing thread-sharing or aliasing. With only one point of access to data, no remote actions can invalidate assumptions. Examples include separation logic [134, 199, 221], linear and unique types [265, 112], and static regions [249].

Isolation with Serialized Sharing (§2.3) These techniques combine isolation with highly restricted, serialized sharing, where multiple access paths exist to data with certain invariants, and unpacking of any possibly-aliased shared state is serialized. Examples include concurrent separation logic [43, 258], most work on race freedom with mutexes [3, 87, 86, 85], and single-threaded systems that permit a single existential unpacking at a time [229, 230, 195, 80, 24, 183].

Read-sharing (§2.4) These techniques provide ways to ensure that for the duration of sharing (whether concurrency or caller-callee aliasing), side effects may be somehow prevented. Examples include various verification tools augmented with permissions [41, 36], effect systems for "compatible effects" [35], and reference immutability [28, 252, 275, 276].

Interference Summaries (§2.5) These techniques systematically identify and summarize interference from other sources at each place an assumption is made. Examples are mostly limited to rely-guarantee program logics [139, 193], and variations thereof [260, 82, 73, 83]. Interference summaries may be thought of as a generalization of read-sharing: rather than prohibiting interference outright, interference is precisely characterized (and may in fact be non-interference).

Each of these approaches has its own strengths and weaknesses. Global approaches are straightforward but non-compositional. Separating approaches (with and without serialized sharing) can be very precise in well-segmented programs and neatly handle explicit memory management, but handle sharing less gracefully. Read-sharing is highly effective and lightweight, but by design very coarse. Interference summaries have nearly dual strengths and weaknesses to separation, easily supporting sharing, but only weakly supporting private data. All approaches have been successfully used for safe concurrency (race freedom, or concurrency-aware verification). Note that the approaches are not mutually exclusive, and some of the most interesting verification tools blend ideas from multiple categories. These technique families can be thought of as potentially-complementary verification styles; adding support for directly stating specifications in one technique's style to a system based

on another technique can add flexibility or power. We do not claim they completely summarize the space of verification techniques, but we are not aware of any techniques that fall completely outside these categories.

A relatively underpopulated region of related work is use of explicit interference summaries for aliasing issues. Rely-guarantee reasoning has been well explored for thread interference, but not for aliasing, with the exception of very recent work [174] and first-order applications of ideas from concurrent program logics [269].

Based on these strengths and weaknesses, there is no obvious victor, no system with maximal naturality and expressiveness. But there is another important criterion on which we can evaluate these systems: interoperability with mostly-unverified code (we will allow ourselves to assume some basic type system). Only read-sharing (Section 2.4) and interference summaries (Section 2.5) support natural integration with unverified code; other techniques require reasoning explicitly about the behavior of all code (e.g., whether it preserves global invariants) at once.

The remainder of this chapter surveys each of these techniques in order, beginning with a high level overview of the approach, then recounting some of the most important developments for that reasoning technique. We conclude this chapter with some discussion focused on orthogonal axes: implementation strategies for verification systems (Section 2.6), and the relationship within systems between sequential correctness and concurrent correctness (Section 2.7). This dissertation builds primarily on material from Sections 2.4, 2.5, 2.6, and 2.7. The other portions of this chapter serve to contrast the style of verification we do (reasoning about explicit interference between program components as in Section 2.5) with more widely-known approaches to verification.

The themes given above are each a high-level approach to reasoning about program behavior, but there is the separate matter of what technical machinery is employed to instantiate the reasoning approach: how the high-level approach is embodied in a formal mathematical system. We focus our attention on two specific styles of formalization: type systems and program logics. These verification techniques are both used in formalizations in the rest of this dissertation, and have the philosophically desirable property of expressing an explicit proof of correctness for a program, in particular directly exposing such things as

induction principles over repetitive program behavior (loops, recursion). Thus it is possible to use these systems as tools to guide program design based on the correctness argument for an abstract algorithm. Related static analyses — symbolic execution, bounded model checking, abstract interpretation, and verification-condition generators which dispatch obligations to SMT solvers in the style of ESC/Java [89] and Boogie [66, 17] — do not serve this goal as well because they focus on essentially inferring behavior of unconstrained programs or synthesizing conditions under which a given program behaves as desired, rather than being prescriptive about program structure. We discuss these uses of these techniques less than program logics and type systems, though each of the high level reasoning techniques we cover could in principle be instantiated for these formalisms as well.

2.1 Global Reasoning

The canonical example of global program verification is Hoare logic [126]. This program logic proves program correctness using a judgment for pre- and post-conditions on imperative commands that modify state. Its judgment takes the form $\vdash \{\, P \,\}\, C \,\{\, Q \,\}$ (often called a Hoare triple), where P and Q are assertions written in a specific logic, which describe respectively the global program state before and after the execution of C (which may be the composition of more primitive commands). Typically, soundness for these program logics takes the form of a guarantee that for all states S, if $S \models P$ (assertion P holds in the state S) and $\vdash \{\, P \,\}\, C \,\{\, Q \,\}$, then executing C in state S does not fault, and if $(C, S) \rightarrow^* (\mathsf{skip}, S')$ (execution of C from state S terminates with final state S'), then $S' \models Q$. Variants exist for total correctness (guaranteed termination) but we restrict our attention to partial correctness (correctness modulo non-terminating programs).

The main problem with global verification techniques that prove very strong state properties is that composition of program components becomes quite difficult, and aliasing greatly complicates proof rules. Specifically, a procedure verified in Hoare Logic must specify a *global* pre- and post-condition, potentially requiring a new specification triple (and re-verification) if the same procedure is reused in a new program. For example, a procedure's proof may include the derivation $\vdash \{\, y = 7 \,\}\, x := 3 \,\{\, y = 7 \,\}$, exploiting the fact that actions unrelated to verified properties are unrestricted. This may suffice in one program,

but reusing this in a program that also relies on global variable x requires new, stronger specifications. To amend this in the case of only global program variables, procedure summaries can be extended by *modifies clauses* specifying all global variables a procedure modifies, allowing some modularity. But without additional tools such as read-sharing (e.g., [208]), modifies clauses buy relatively little modularity in the presence of heaps and aliasing.

For concurrent programs, the best-known extension of global program logics to concurrency is the Owicki-Gries method [204]. Within a thread, verification proceeds roughly as in Hoare Logic, but additional checks are performed at points of structured parallel composition ($C_1 \parallel C_2$). Owicki and Gries describe the intuition for the checks as ensuring that the actions of one thread do not interfere with the *proof* of another thread. At a high level, when verifying $C_1 \parallel C_2$, the proof system requires ensuring that every intermediate assertion in the proof of C_1 is preserved by every atomic action in C_2 (and vice versa, plus additional checks to ensure deadlock freedom). This is effective, at the cost of increasing proof burden quadratically in the number of threads.

An alternative approach to global reasoning that permits greater modularity is to choose a relatively weak set of assertions and/or disallow operations that could invalidate any assertion. In a sense this is the approach taken by most type systems for languages with heap allocation, where the standard syntactic type soundness proof uses a monotonically growing heap typing ($\Sigma : \mathsf{loc} \rightharpoonup \tau$) where each location has the same simple type in successive heaps. This is also the approach taken by richer languages with allocation such as DML [272, 273] and ATS [48], which allow refinements of mutable values (refined in relation to only immutable data, not other mutable data), enforcing that every update satisfies the refinement. These systems have the limitation of proving only weak properties, especially in the presence of unrestricted shared memory: type-correct data races remain the source of many bugs in modern software.

2.2 Separation and Isolation

A natural step to improving modularity is to make assertions local and independent. This is the approach taken by a broad swath of work on Separation Logic (Section 2.2.1), as well as substructural type systems (Section 2.2.2), which we survey in this section. These

systems have the advantage that local reasoning is straightforward, assumptions are only invalidated locally, and complete state separation lends itself naturally to safe concurrency (by preventing all interference). The obvious disadvantage to strictly isolating state is that sometimes sharing is desirable for convenience or performance.

2.2.1 Separation Logic

Separation logic (SL) [199, 221] is a Hoare-style program logic using the logic of bunched implications [134] as an assertion language. The logic uses a "small footprint" approach to specification, where the specification for a command mentions only the program state it requires to execute safely. This means that specifications need not mention that state omitted from the specification remains the same, making the specifications reusable regardless of what other program state exists. Core to the program logic is the use of separating conjunction ($*$) in the specification language, where $P * Q$ is an assertion that the program state splits — at verification time — into two completely disjoint sections, one satisfying P, the other Q.[1] A related connective — separating implication — expresses properties of conjoining a heap with certain other disjoint heaps: $P \mathrel{-\!\!*} Q$ is an assertion true of a heap when extending that heap with another disjoint heap satisfying P would produce a heap satisfying Q. This is then embodied in the entailment $P * (P \mathrel{-\!\!*} Q) \vdash Q$, which says that any heap that splits into two heaps satisfying P and $P \mathrel{-\!\!*} Q$ respectively also satisfies Q.

Facts about the heaps, so-called "points-to" facts, are represented by assertions of the form $n \mapsto v$, which asserts that the heap contains a location n storing a value v. In *classical*[2] SL this also implies that no other locations are present in the heap (unless spatially conjoined with other assertions), though this distinction is important only for safe memory deallocation; the remainder of this section will discuss only material that applies to both classical and intuitionistic separation logics. Aliasing can be expressed (e.g., $\{x \mapsto n * y \mapsto n * n \mapsto v\}$), but reasoning about the high-level effects of changing an aliased heap cell

[1] These heaps may be arbitrarily large. For example, any heap section satisfies the assertion True.

[2] In *intuitionistic* separation logic, a heap assertion that is true in one heap is also true in any larger heap. In *classical* separation logic, assertions must describe the relevant heap completely, which is necessary for safe memory deallocation: this means that deallocating the heap fragment satisfying $n \mapsto v$ deallocates only the single heap cell at location n, and nothing else.

$$\textsc{Frame}\quad \frac{\{\,P\,\}\,C\,\{\,Q\,\}}{\{\,P * R\,\}\,C\,\{\,Q * R\,\}} \qquad \textsc{Parallel}\quad \frac{\{\,P_1\,\}\,C_1\,\{\,Q_1\,\} \qquad \{\,P_2\,\}\,C_2\,\{\,Q_2\,\}}{\{\,P_1 * P_2\,\}\,C_1 \parallel C_2\,\{\,Q_1 * Q_2\,\}}$$

$$\textsc{Ramify}\quad \frac{\{\,P\,\}\,C\,\{\,Q\,\} \qquad R \vdash P * (Q \mathrel{-\!\!*} R')}{\{\,R\,\}\,C\,\{\,R'\,\}}$$

Figure 2.1: Selected proof rules for sequential and concurrent separation logic, variable side conditions elided.

requires combining many low-level heap assertions together. Data structure invariants in separation logic are typically defined by recursive predicates that expand to a collection of points-to facts.

Separating conjunction naturally gives rise to the Frame rule in Figure 2.1, which states that any command with a specification $\{\,P\,\}\,C\,\{\,Q\,\}$ may be executed in a state satisfying P and additional (disjoint[3]) properties, and those additional properties are preserved. This reflects the highly local nature of individual operations in a straightforward way, enabling local reasoning in broader contexts. The frame rule is semantically justified by two essential correctness properties for separation logic: *safety monotonicity*, which specifies that a computation safe to run in one state also runs safely in a larger state; and the *frame property*, which informally states that for a triple $\{\,P\,\}\,C\,\{\,Q\,\}$, executing C in a state satisfying P does not touch any state not described by P (or allocated by C itself).

This local nature played important roles in subsequent work generalizing separation logic to support rich notions of abstraction and modular composition [202, 206, 207], where module- or object-private state can be handled by abstraction rules that hide spatially

[3]Most separation logic rules carry side-conditions restricting variable modification, since the standard separation logic assertion language does not divide the stack. In fact, these side conditions allow read-sharing of stack variables. These variable conditions can be elided by treating variables as explicit resources [205], and most separation logic papers now note this possibility or adapt the alternative approach of carrying Reynold's *syntactic control of interference* [223] into separation logic [219].

disjoint state satisfying an invariant from client code.

The disadvantage of this strict separation is that sharing is completely disallowed. Subsequent iterations on separation logic mix in ideas from each other verification theme, including serialized sharing [43], read-sharing [36], and interference summaries [260, 83, 82, 73, 71], typically focused on safe concurrency rather than taming aliasing.

Complete separation naturally gives rise to some forms of safe concurrency, as threads acting upon disjoint memory segments do not interfere. The simplest form of this is the notion of *disjoint concurrent separation logic*, characterized by the simple PARALLEL rule in Figure 2.1. Disjoint concurrent separation logic is rarely studied in its own right because it is actually a degenerate case of concurrent separation logics with shared resources [43, 200, 107, 127, 44, 258, 219], by omitting critical section support.

One disadvantage of strict spatial conjunction is that while aliasing is possible to express, it is not straightforward to reason about the non-local effects of a heap modification on a larger data structure. In theory separating implication alleviates this difficulty; recall that a state satisfies $P \mathrel{-\!\!*} Q$ if extending it with any state satisfying P would produce a state which all together would satisfy Q, reflected in the proof rule $P * (P \mathrel{-\!\!*} Q) \vdash Q$. But until recently its practical applicability was poorly understood. An exciting recent development is Hobor and Villard's recognition of the power of separating implication in separation logic sequents (formula implications), codified by their *ramification* rule [128] shown in Figure 2.1. RAMIFY essentially uses a local specification of a command together with a proof that a "less local" assertion satisfies not only the local precondition of the command, but a (spatially disjoint) remainder of state that would satisfy the "less local" postcondition if extended with state satisfying the local postcondition. The RAMIFY rule exploits the underappreciated richness of the sequent calculus for separation logic assertions, allowing a developer to prove a family of "ramification" lemmas for updating internal nodes of data structures. Hobor and Villard develop a small library of semantically justified sequent axioms to simplify ramification goals, as well as a family of axioms for updating interior nodes of trees, DAGs, and directed graphs. They show that RAMIFY is not only an improvement over prior approaches that contort both proofs and code to fit updates to data structures with sharing into the frame rule [37, 274], they show it generalizes earlier

attempts to deal with interior sharing in structures in a principled way [49]. On top of all this, they show that it is actually derivable in any separation logic with a frame rule and rule of consequence, making it widely applicable.

The ramification rule does not alleviate all concerns about non-local effects on assertions. Notably, taking advantage of ramification requires either shoe-horning a structure into one of the forms they have already proven soundness for, or *extending the metatheory* for separation logic *for each new data structure*. The former is undesirable, though not as unpleasant as proofs without the insights from Hobor and Villard's work, while the latter requires new semantic proofs outside separation logic.

A commonly-cited limitation of separation logic is that it lacks higher-order specifications: the ability to parameterize a proof by some abstract specification, which is necessary to verify code using higher-order procedures like `map`,[4] whose effect depends on the function f passed in. Higher-order separation logic addresses this problem by allowing specifications to quantify over other specifications [25]. Krishnaswami's thesis [144] provides some clear examples of its utility. A notable use is to support some aliasing by using an abstract assertion logically conjoined with a concrete specification to verify read-only access to structures.

Separation logic for higher-order stores (supporting heap-accessing and modifying procedures stored in the heap) are less often presented due to the complexity of the metatheory. Typically program logics are proven sound using denotational methods, and the denotation of higher-order imperative state is notoriously subtle because it is an impredicative construction [155, 10, 9, 31, 220, 32, 233, 25]. The alternative — rarely used — is to prove soundness syntactically [192].

2.2.2 Substructural Type Systems

Substructural type systems are type systems with restricted "structural" rules: those typing rules which permit duplication, reordering, or dropping of assumptions. The most prevalent, and relevant to our purposes, are linear and affine types [265], beginning with Girard's work

[4] $\mathsf{map}\ f\ [] = []$, $\mathsf{map}\ f\ x : xs = (f\ x) : (\mathsf{map}\ f\ xs)$

on linear logic [101] (of which separation logic is also a descendant). Here we survey a few key examples of substructural type systems, for restricting duplication of values, and for restricting duplication of abstract capabilities to perform actions (including capabilities that might describe the current state of the heap).

Walker gives an excellent overview of (pure) substructural type systems [265], where values of substructural types are treated as assumptions in the corresponding substructural logic. In imperative settings, affine types have become more recognizable through uniqueness and external uniqueness types [112]. Totally unique references permit strong updates to heap locations [238, 266], changing the type of values stored in a single heap location, as no aliases exist to make stale assumptions (separation logic permits this as well). Similar to disjoint concurrent separation logic, uniqueness and external uniqueness are well-suited to partitioning disjoint datasets among threads. We focus here on verifying non-trivial properties like structural invariants with strong updates.

Alias types [238, 266] is a substructural type system capable of specifying and maintaining similar structural properties. Typing assumptions are linear, and use singleton types (types that identify a specific run-time value as their only inhabitant) to describe data layout: $\ell_0 \mapsto (\mathsf{ptr}(\ell_1), \mathsf{ptr}(\ell_2))$ states that location ℓ_0 points to a tuple whose values are the pointers ℓ_1 and ℓ_2. Storing a known value to a field allows updating the singleton field type. This offers similar structural expressiveness to early separation logics [221], though there is no correspondingly rich assertion language.

Haller gives an excellent overview of unique types and external uniqueness [112]. Unique values are affine values. Informally, externally-unique references are the only reference into a set of heap allocations that may alias each other, but are unreachable from other objects except through paths using the externally unique reference. External uniqueness is less useful for verification than total uniqueness due to internal aliasing (making it useless for fine-grained memory reclamation) but is a lightweight way to support disjoint regions [249] and is useful for safe message passing [133, 111] and data parallelism [105]. Truly unique values are useful for not only basic strong updates, but updates to computationally-irrelevant type parameters used in systems for ownership [180] and related notions [102].

Another important class of substructural type systems are those with linear capabilities

granting access to shared/duplicable resources. One recent example of interest to this dissertation is Mezzo [216, 13], which extends an ML-like language with linear capabilities for strong updates to memory, and non-linear capabilities for immutable state. The advantage of this style of reasoning is that code may, for example, allocate a mutable list node with a unit type in place of a tail pointer, as a partially-initialized list node, and complete initialization later: replacing the unit with an actual list reference (and updating the capability appropriately) then permits converting the list node to immutable, even though the node was incrementally constructed in several steps. This can also be used to ensure data race freedom, in a style similar to ideas explored in Chapter 3.

One common limitation of substructural type systems is the fixed choice of which resources are substructural, which may not be appropriate for all use cases. Krishnaswami et al. [145] offer a solution to this issue in the form of a novel sharing rule that allows *developer* choice of substructural resources. Given an element of a type family (type with one parameter) indexed by a commutative monoid (as a type) and operations over that type, the sharing primitive produces a new set of operations that split the resource according to the monoidal structure — as long as each provided operation preserves monoidal frames in the index. For example, they can derive general duplicable references in the context of a linear type system using the sharing rule, by using the unit monoid. They can derive a sharable monotonically increasing counter where each reference carries a lower bound on the counter's value by choosing $(\mathbb{N}, \sqcup)$ as the monoid, as long as the type of the increment function proves that it preserves the frame: that for all x and y, incrementing a $\mathsf{Counter}(x \sqcup y)$ produces a $\mathsf{Counter}((x + 1) \sqcup y)$, the $\mathsf{Counter}$ type itself interprets its argument as a lower bound, and y represents the framed out lower bounds of all other points of access to the counter. The system also supports extended operations for splitting and joining the resulting types, again based on the monoidal structure.

Regions Regions [249, 92] are (static or dynamic) partitions of memory, within which data structures may be allocated. With explicit region types and runtime region allocation, by adding the region of a resource to its type (or in program logics, relevant assertions about a state [71]) and making a static access token for the region linear, it becomes easy to support

safe memory deallocation: deallocation consumes the region token, and subsequent memory accesses to that region no longer type check due to its absence. This can be used not only for manual memory management (as in Cyclone [109, 125, 244]), but to implement a memory-safe copying garbage collector [267, 93]. Regions do not need to be explicit; as mentioned above, external uniqueness corresponds to a region system, and one straightforward way to model external uniqueness in a type system's metatheory is using regions [105, 106].

Hoare Type Theory Hoare Type Theory (HTT) [188, 185, 212, 186, 190] is an elegant combination of ideas from Hoare logic and separation logic with dependent types. Impure computations are embedded as a monadic type [177, 247] indexed by pre- and post-condition assertions. This allows a powerful and flexible combination of rich dependently-typed pure code with impure computations at a low level of abstraction. Additionally, this offers a form of program logic integrated directly into a programming language, whereas most program logic implementations are separate analysis tools from a language's compiler (a common non-technical criticism of program logics in practice). HTT is also a higher-order separation logic [25], because programs may abstract over effectful terms and their specifications. Because HTT and Ynot build on separation logic and Hoare logic (depending on which indexed monad is used), they suffer from the same difficulties expressing aliasing as the corresponding program logic (but could now leverage Hobor and Villard's ramification work [128]). Effective automation of proofs in HTT has been explored in the Ynot system [189, 52, 162, 190], described in more detail in Section 2.6.

Dijkstra Monad Swamy et al. propose the *Dijkstra Monad* [245], a monadic encoding of Dijkstra's predicate transformers [69] in F* [243, 241]. This weakest-precondition approach is powerful enough to embed HTT into it, and F*'s use of SMT solvers to automate implications in its parametric refinement logic makes it fairly effective on the smaller examples the Dijkstra Monad has been used for. The monadic presentation is effectively a system of one substructural value (the heap).

2.3 Isolation with Serialized Sharing

Complete isolation of state is sometimes inconvenient for succinct or efficient communication, whether through aliases shared among distant portions of a sequential program, or for communication among threads without copying. To verify programs exhibiting these types of communication, a common design principle is to serialize access to the shared resource. Multiple references to a shared resource may exist, with the knowledge that the shared resource satisfies a particular invariant. Accessing the state addressed by this invariant is associated with a form of existential unpacking, at which point details of the state are usable and the invariant is assumed to hold. The program may violate the unpacked invariant but must restore the invariant and re-pack the state before other parts of a program may access the state. In concurrent systems, additional dynamic semantics may be associated with this unpacking and re-packing: essentially, these correspond to acquiring and releasing mutual exclusion locks with an invariant over state guarded by that lock. In sequential settings with aliasing and unrestricted recursive types (for example, objects, where a class definition may reference itself as in Java or C#), different restrictions are often imposed, such as only permitting a single object to be unpacked at once, to avoid unsoundness issues with naïvely unpacking the same object twice and updating assumptions about two aliases as if there were no interference. Here we describe the appearance of serialized sharing in the form of lock invariants in concurrent separation logic, data structure or object invariants, and typestate.

2.3.1 Concurrent Separation Logic with Locks

Concurrent Separation Logic (CSL) with locks [43, 200, 107, 127, 44, 258, 219] is a perfect exemplar of isolation with serialized sharing. Many variations of CSL exist with different proofs and different styles of mutexes, but all share a common design. Separation logic is extended with "shared resources" (a.k.a.: mutexes guarding known state). The mutexes guard access to certain state satisfying a particular invariant. Upon acquiring a resource, the resource invariant is added to the local state, and verification continues as in sequential separation logic. Releasing a resource requires reestablishing that resource's invariant

in local state, and that resource is removed from local state. The dynamic semantics of resources are as expected for a mutual exclusion lock: only one thread may hold the lock at a time, so while a thread holds some resource, it is safe for it to assume non-interference on that resource's invariant.

The clearest and most direct proof of soundness for concurrent separation logic is due to Vafeiadis [258], who gives direct semantics to CSL judgments in terms of operational semantics. The original CSL soundness proof style [43, 200, 219] gives the semantics in terms of interleaved execution traces, which is more difficult to reason about or extend, while Vafeiadis extends his technique to CSL with permissions [36], and a variant (RGSep) with some interference [260]. The new proof also gives a very direct explanation for why resource invariants must be precise (unambiguously identify a single subheap) for the conjunction (of specifications) rule to be sound. This same style of proof method extends to more recent generalizations of concurrent program logics [70].

2.3.2 Data Structure Invariants

A more widely-recognized instance of isolation with sharing is the use of data structure invariants [18, 229, 230]: properties of a structure that are expected to hold when it is not in the middle of being modified. This is a prevalent technique for ensuring correctness of object-oriented software, taught in undergraduate curricula and incorporated into cutting-edge program verification tools. Invariants also help keep static verification assertions small. Instead of precisely tracking heap shape and depth, structures can be summarized by their (possibly recursive) invariants. Modification is supported by *unpacking* the invariant (in the sense of unpacking an existential type), modifying the structure, and *repacking* after checking the invariant has been restored. The danger in doing so arises from being unable to distinguish aliases to the same structure: if two aliases are unpacked separately without care, operations through one alias may not be reflected in the local assertions about the other, resulting in inconsistency and unsoundness. Different techniques handle this differently, such as by permitting at most one unpacking at a time [195] or tracking some heap shape [102].

Nistor and Aldrich have proposed a particularly expressive system of *object proposi-*

tions [195, 196]: a program-logic-style type system using abstract predicates [206] (per object) and separation-logic-inspired connectives to specify *object refinements*, essentially an enriched typestate much closer to a full predicate logic. Updating objects requires unpacking them, updating them (intermediate states may violate the object proposition), and repacking them with the proposition satisfied. An object's proposition is fixed while aliased, but may be changed if unaliased. Only one object may be unpacked at any time.

2.3.3 Typestate

Typestate is an approach to statically enforcing protocols for resource usage. For example, enforcing that an iterator over a collection is only advanced after checking that it is not at the end, and is not used after the collection is modified, is statically enforceable using typestate. A class declares some number of abstract states it may be in, and methods of the class declare how they advance the object (and/or its arguments) through different states of the protocol. Early designs imposed draconian aliasing restrictions, but later work [24, 183] added a notion of access permissions that grants different permissions to change or make assumptions about an object's typestate. The notion of access permissions [24] bears strong resemblance to mutability qualifiers in reference immutability (Section 2.4.1), in that there are variations on unique references; a single reference with the ability to update typestate aliased by many read-only instances; and immutable permissions. Fractional permissions [41] (Section 2.4) are used to split and recombine the permissions (giving them a partial commutative monoid structure), which allows natural expression of rich aliasing patterns while retaining the ability to soundly reason about protocols and recover stronger permissions. Recent work added an elegant borrowing system to those permissions [183]. Each typestate can be associated with predicates on the object's state, allowing static reasoning about changes in typestate, sometimes aided by additional dynamic checks.

Militão, Aldrich, and Caires propose a notion of a *typestate view* [171] to manage typestate with aliasing. Each typestate may decompose into multiple *views*, essentially component typestates, each corresponding to a physically disjoint portion of the referent. Aliases may be created to different views of the same object and, because they address physically

disjoint memory, may operate and evolve independently. Later, compatible component views may be rejoined to a single typestate for the whole object.

A key limitation of all typestate work is that protocols are finite-state. Typestate cannot prove that a counter is only ever incremented, though it can prove that a counter moves through certain partitions of the number space (such as zero or non-zero) in a given order.

2.4 Read-sharing

When a program fragment does not require the ability to modify part of a program's state, it is often useful to make that purity evident, whether by passing state as read-only to callees (to preserve caller assumptions) or ensuring non-interference among multiple threads that read from the same portion of the heap (to avoid incurring the performance overhead of serializing concurrent accesses or acquiring even reader-writer locks). We call these techniques *read-sharing*. They often exist in combination with isolation and serialization techniques in the same system. We focus here on splittable read permissions, effects for non-interference, and reference capabilities that restrict mutation.

Perhaps the most important technique in this space is Boyland's fractional permissions [41]. Boyland proposes attaching fractions to typing or verification assumptions, where the fractions represent the share of "permission" a context receives to a shared resource. Receiving a permission of 1 (all permission) allows modification of a resource, a nonzero fraction less than 1 grants read-only access, and absence of permission to a resource prohibits all access. Boyland then uses this approach to prove race-free deterministic execution in a language with fork-join concurrency: providing each child thread with half the parent thread's permission allows the children to read from but not write to shared state. Because permissions are preserved by each thread, when child threads terminate, the parent context regains its original total permission, meaning a thread can hold full write access to some state, execute read-only operations over the state in parallel, and regain write access after the parallel fragment completes. This approach to temporary sharing has worked its way into dozens of other verification systems, often generalized to partial commutative monoids. A small wrinkle in the usability of this approach is the requirement to choose *concrete* permission fractions for read access when splitting permissions: the exact value chosen is

irrelevant, but can hamper refactoring and code reuse where the arbitrary permission chosen as input for some procedure is larger than the arbitrarily chosen fraction available at a would-be call site. Heule et al. give a concise overview of the problems with concrete permissions, and describe an approach for *abstract read permissions* [123], where uses of read permissions are parameterized by an abstract fraction, and the verification tool solves for feasibility of instantiating the fractions as part of verification.

Bornat et al. [36] describe a style of permission accounting for separation logic directly inspired by Boyland, where points-to assertions are associated with a fraction, inheriting Boyland's write/read distinction where an assertion with nonzero fraction permits reading from the state described, but a full permission is required to modify the resulting state. This allows read sharing among parallel processes, while preserving precise verification information in each thread.

Alias types [238, 266] (Section 2.3) include rules that permit exchanging a linear reference typing assumption for one that may be duplicated, and the duplicated versions prohibit strong updates, producing a form of read-sharing. Because alias types are described for typed assembly language (which obscures high level control flow), there is no structural way to recover a reference assertion as linear.

Read-sharing can also be presented as an effect system [159], an approach epitomized by Deterministic Parallel Java (DPJ) [35]. DPJ ensures deterministic execution by computing read and write effects to different regions [249] of memory, and only permitting parallel composition of two computations if the effects are compatible: each region written by one computation is untouched by the other, and regions accessed by both computations are only read by both. DPJ also includes a sophisticated region system to make this approach to safe concurrency more practical, including nested regions and rich treatment of arrays whose elements point to disjoint regions.

The limitation of read-sharing is that it prohibits useful communication. This is the motivation for many of the concurrent program logics described in Section 2.5, and for subsequent iterations of DPJ [34] adding non-determinism through software transactional memory [235].

2.4.1 Reference Immutability

Another example of lightweight read-sharing is reference immutability [28, 252, 275, 276, 105, 132], which is also present in owner-as-modifier interpretations of ownership and universe types. The core idea is to attach qualifiers to reference types that restrict not only actions to the immediate referent, but also to state reached through that reference. Because write-permitting qualifiers exist as well, single-threaded uses of reference immutability do not suffer from the same stricture as read-sharing for safe concurrency: a callee may always require a write-permitting reference argument instead of a read-only reference. Of course, this is really a choice between using or not using read-sharing, since neither is "wrong" in any well-defined way for single-threaded uses.

Unfortunately, each instantiation of reference immutability tends to redefine and/or rename the qualifiers and their semantics slightly, so there is no truly canonical set of qualifiers and semantics. A minor exception to this is the immutable qualifier, which means some form of immutability in several systems including ours [275, 276, 105], but the semantics differ in their interaction with generics. Because we build on reference immutability in this thesis, we will use the qualifier names we use in Chapter 3, and compare the related systems using a common terminology.[5]

The main reference immutability permissions are writable, readable, and immutable. writable references are standard heap references, permitting reads and writes. readable references allow reads of referent fields, but not writes. Additionally, if a field is read through a readable reference, the result type of the read will never be a writable reference type: the read-only semantics of readable apply transitively, to all references acquired by reading through a readable reference. This is called *deep reference immutability*. readable and writable references may alias each other, so a caller can pass a readable copy of a writable reference to a callee, and be certain the callee will not modify state reachable through that

[5]Our qualifier choice is derived from the prototype C# extension described in Chapter 3 and a prior publication [105]. Early systems have some overlap in choice of permission names, but none of the overlap is shared with our terminology. Most other systems use mutable where we use writable due to the preference of a Microsoft team. Most other systems use readonly (an unreserved term in Java) where the C# prototype described in Chapter 3 uses readable because in C#, readonly is a reserved keyword with the same semantics as Java's const. Renaming was necessary in the C# prototype to avoid confusion and/or shadowing an existing language feature.

reference (assuming no writable references to that reachable state are also passed). immutable refers to data that is truly immutable, and may alias readable references (but not writable references!). Immutability also applies transitively, so any reference read through an immutable reference will also be immutable.

These permissions were designed with the intent of enforcing caller-callee read-only contracts; passing a readable reference to a mutable object to a callee ensures the callee cannot modify the object directly (aliases may exist).

Prior reference immutability systems build on similar bases, but different goals lead to different designs.

Original Formulation [28] Reference immutability — both the technique and name — is originally due to Birka and Ernst [28]. They describe a system JAVARI2004[6] with writable and readable, where the former is implicit (i.e., absence of readable is interpreted as writable, making writable the default). The design includes initial proposals for object immutability and generics. The original implementation's generics were limited to a single permission parameter per class or method, and employed a templating approach to soundness [27]: the method was checked twice, once for each possible permission. This was also the only reference immutability implementation to employ runtime representation of mutability per reference, which incurred slight runtime overheads but permitted downcasts from readable to writable to be dynamically checked.

Javari [252] is largely similar to our first two qualifiers, with some exceptions. In addition to their readonly qualifier, the (unnamed) default field qualifier is explicitly polymorphic in the permission used to reach the object. Their mutable qualifier has semantics similar to our writable for local variables (in conjunction with the default path-polymorphic field qualifier), but reading a mutable field grants write permission on the result regardless of the path used to access the field; an explicitly mutable field *always* grants write access. They also include assignable for fields that may be assigned through a read-only reference (but not necessarily

[6]The name was originally JAVARI, but that name was repurposed for a system with slightly different semantics by Tschantz and Ernst [252], and the latter implementation is still maintained today so we permute the original system's name.

read out as mutable; essentially the opposite of C++'s `const` or Java's `final`). It includes a class templating solution to allow methods limited polymorphism over permissions to the receiver.

Immutability Generic Java (IGJ) [275] uses Java's generics to track permission qualifiers, enabling direct use of generics for generic permissions. It also adds object immutability and an immutable permission to the ideas in JAVARI. (This design was a strong influence on the design described in Chapter 3.) Their approach to object immutability (creating immutable objects) is constructor-based, using an initialization pattern, thereby forcing developers to decide on making an object immutable at allocation sites. Once the constructor returns the object is either mutable or immutable permanently; objects cannot be allocated, then mutated, then made immutable later based on run-time results. IGJ also adds support for covariant subtyping of generic types (e.g. `Immutable List<Integer> <: Immutable List<Object>`), which is safe for deep read-only access (in contrast to the well-known unsoundness in Java's covariant array subtyping).

Ownership Immutability Generic Java OIGJ [276] combines ownership (see below) with reference immutability, using owner-as-dominator ownership to ease creation of immutable objects by constraining aliases to immutable objects during construction (similar to our own use of external uniqueness for this purpose). It addresses polymorphism over reference immutability qualifiers, but much of its simplicity stems from permissions being *shallow* rather than transitive for generic fields: retrieving the first element of an immutable List<P T> returns a P T, without modification (the collection itself is immutable, but the values are not). This choice allows expressing some useful patterns inexpressible in other systems (e.g., a collection with immutable structure, but whose elements remain mutable).

Because these systems all include shallow permissions or a mutability override to allow writes through a theoretically read-only reference, they are difficult bases to build a safe concurrent language upon, though race freedom was suggested in several of these systems as a natural application of reference immutability (sometimes suggested in combination

with other analyses). These sometimes-shallow permissions are entirely consistent with the original design intent for these earlier reference immutability systems, where the idea was that the deep interpretation applied only to the conceptual abstract state of an object. Generic parameters, or overriding permissions such as an explicit mutable were considered to move some field referent out of the governed abstract state. This is a design decision made by these systems' authors; there is no technical basis for these deep and shallow behaviors being right or wrong from a technical point of view. Our work in Chapter 3 makes a different choice given the explicit aim of producing a system that ensures data race freedom with relatively simple static checks.

Ownership type systems [39, 38, 65, 54] and Universe types [68] describe a notion of some objects "owning" others. The "owner-as-modifier" interpretation resembles reference immutability, as references to objects not owned by the current object are deeply read-only. The "owner-as-dominator" interpretation requires all heap paths from the root owner to an object to pass through that object's owner, resembling external uniqueness. Changing ownership is similar to permission conversion, and can be done flexibly with support for external uniqueness [180, 153]. Ownership and Universe types have been used for safe concurrency [39, 38, 65], but mainly for associating locks (owning objects' implicit locks) with the data they protect (fields of directly owned objects).

Inference

Inference for reference immutability systems is trivial, because there always exists an uninteresting solution in the sequential case: if all references are writable, the program will type check. Finding a non-trivial, useful solution is quite difficult. This is similar to the challenge faced in inference for ownership/universe type systems [131]. Nonetheless, effective inference is possible [218, 132], and the solutions are likely to work with concurrent extensions (Chapter 3) as well given a properly annotated concurrency library.

2.5 Interference Summaries

The final family of verification techniques we consider is that of interference summaries. These techniques allow some aliasing of shared resources, whether considered between

threads (which has been well explored) or individual references (which has only recently garnered attention). Each alias is restricted to some set of permissible actions on the shared state, and also has a summary of actions other aliases are permitted to take on shared resources. Assumptions that are stable with respect to the interference summary (preserved by any action the summary permits) are safe to make.

This family, more than any other, subsumes other techniques to varying degrees because interference techniques are typically expressed in terms of arbitrary binary relations on state. In some sense, this family naturally permits expressing specifications directly in terms of a state model, rather than with some restricted assertion language. Separation is generally encodable by each alias receiving exclusive update permission to some substate and assuming any possible interference on other state. Serialized sharing is similar, with the permission set and interference summary conditioned on shared state such as mutexes. If an alias can be split in two where each resulting alias confers only read access to some state, this is a form of read-sharing, which is a very coarse but remarkably useful form of interference summary. The exact limits of this subsumption depend on the exact structure of the permission set and interference summary: for example, the original rely-guarantee program logic [139] has an interference summary with a fixed (global) footprint, and the rules do not permit much change to the summary beyond weakening, so it is not trivial to encode separation logic. At the same time, first-class support for other verification approaches within a system can make interference summaries easier to use, such as Local Rely-Guarantee's use of multiple interference summaries for different regions of memory [82] (described in more detail in Section 2.5.1).

This section describes rely-guarantee reasoning (Section 2.5.1) and subsequent modularity improvements to it, recent work on characterizing interference between *aliases* rather than threads (Section 2.5.2), and particular uses of interference in verifying sophisticated concurrent data structures (Section 2.5.3).

This section's techniques are of particular interest to us because Chapters 4, 5, and 6 explore interference between aliases in sequential and concurrent settings. We build primarily upon the original rely-guarantee reasoning style (Section 2.5.1), but the verification systems we propose are closely related to the techniques in Sections 2.5.2 and 2.5.3 as well. Subse-

quent chapters give direct contrasts against some of the work described here. Our discussion of future work (Chapter 7) includes additional comparisons focusing on how combinations of ideas from the work described here with the systems in subsequent chapters can produce more natural or more powerful verification frameworks.

2.5.1 Rely-Guarantee

Rely-Guarantee [139, 193] is an approach to verifying concurrent (shared memory) programs based on the recognition that the behavior of a thread can be abstracted to a summary of its possible behavior. This addresses the inconvenient process of earlier work like the Owicki-Gries method [204] for concurrent program verification, which relied on first proving threads correct in isolation, then proving that the actions of each thread do not interfere with the proofs of other threads. That is, the Owicki-Gries method requires compatibility checks between threads to inspect each line of code for each thread, rather than an abstraction of each thread. Adding a single line of code to one thread executed in parallel incurs $O(m * n)$ additional verification checks when run in parallel with m other threads of n lines each, making it very computationally expensive to verify programs this way.

The main change rely-guarantee reasoning makes from Owicki-Gries is the addition of *rely* and *guarantee* relations — binary relations on program states — as thread summaries. Each thread is checked with a rely, which summarizes the possible behaviors of other threads. Assertions in the local proof that are *stable* with respect to the rely (i.e. preserved by the rely: $S \models P \wedge (S, S') \in R \Rightarrow S' \models P$) are valid assertions, because other threads should not invalidate them. Each action a thread takes locally must also be allowed by the guarantee relation, which summarizes the effects the local thread may have on the world. This then allows a safe concurrency rule (side conditions on variable modification elided):

$$\textsc{Par}\ \frac{R \vee G_2, G_1 \vdash \{P\} C_1 \{Q_1\} \qquad R \vee G_1, G_2 \vdash \{P\} C_2 \{Q_2\}}{R, G_1 \vee G_2 \vdash \{P\} C_1 || C_2 \{Q_1 \wedge Q_2\}}$$

This rule says that when forking new threads, the threads must assume the interference R of the spawning context, plus interference from the other thread's guarantee (its permitted

actions). This allows verifying each thread in isolation, with simple checks at points of parallel composition.

The original rely-guarantee logic, however, is global, and therefore difficult to use on incomplete programs. Abstracting program modules and hidden (module-private) state is also difficult because to ensure non-interference from other threads, threads must know about (and state non-interference on) other threads' private state. For this reason, a number of systems were developed to combine aspects of rely-guarantee reasoning (which excels at reasoning about interference on shared state) with separation logic (which excels at reasoning about private, unaliased state). The remainder of this subsection surveys key developments in making rely-guarantee reasoning more modular.

The first combinations of rely-guarantee reasoning with separation logic developed concurrently: RGSep [260] by Vafeiadis and Parkinson, and SAGL [83] by Feng, Ferreira, and Shao. There are subtle differences between them, but the key insights were to divide memory into shared and private heaps, and separate out assertions over shared memory from those over thread-private memory. Updates to private memory behave as in separation logic, while updates to shared memory are constrained by the guarantee and must occur in atomic sections to reason about ownership transfer. As in the original logic, parallel composition imposes certain restrictions on the rely and guarantee of composed threads. These logics are expressive enough to verify well-formedness invariants on concurrent structures, such as that a hand-over-hand locking implementation of a sorted list preserves shape and sorting.

RGSep and SAGL both suffered from issues with global assertions on shared data, making the resulting logics less modular than desired. To address this, Feng introduced Local Rely Guarantee (LRG) [82], where the key addition was a separating spatial conjunction on rely and guarantee relations. Each rely and guarantee relation is *fenced* by a precise invariant assertion[7] indicating which fragment of the state the relations govern, and relations on disjoint state may be conjoined into a larger relation on a larger piece of state. This

[7]The invariant must be precise for the same reasons that CSL resource invariants and module invariants in variants of SL with abstraction must be precise [258]. The check ensures that there is no ambiguity in which state is shared, and which state is governed by which rely and guarantee conditions.

separating conjunction on rely and guarantee relations allows greater modularity, including framing and abstraction rules for interference on shared resources. In particular, it permits an abstraction rule over resources shared by a subset of threads, so memory shared by one set of threads need not appear in unrelated threads' proofs. Rely and guarantee relations are specified in terms of transitions between separation logic assertions over the governed state, and interaction with shared state occurs through use of atomic blocks that may transfer between shared and local state, as in RGSep.

A shared limitation of RGSep, SAGL, and LRG is that they offer no new help reasoning about non-local effects on private assertions due to aliasing, and proving guarantee satisfaction over a shared region may still require relating local actions to region-global invariants. However, because all three systems include the frame rule and the rule of consequence, Hobor and Villard's new RAMIFY rule [128] (Section 2.2.1) is derivable in each system.

A weakness of the rely-guarantee approaches discussed thus far is that they all assume structured concurrency, so the rely and guarantee cannot change over time as they might with unstructured concurrency. Dodds et al. [73] introduce *deny-guarantee* reasoning to address this, where rights to operations on various state are represented as capabilities in the assertion context, either explicitly permitting or *denying* a specific action to the local or all remote threads. The permissions for each action form a partial commutative monoid structure, allowing reasoning about splitting, merging, and communicating permissions. In particular, when a thread is spawned, the handle for joining with that thread specifies which permissions are returned from the joined thread to the joiner. A full permission may be split into either some number of guarantee permissions (granting a holder permission for some action) or some some number of deny permissions (specifying that no thread may take a particular action). The advantage of having both the deny and guarantee permissions is the ability to synthesize *local* rely and guarantee relations based on permission flow: guarantee permissions add capabilities to the guarantee, and deny permissions (or full guarantee permissions) imply some action cannot be part of other threads' actions, and therefore refine or constrain the implied local rely. This is expressive enough to capture unstructured concurrency, and rely-guarantee proofs can be translated into a deny-guarantee proof. The main limitation of this approach is that the set of possible actions over state

must be decided upon a priori (or in a system with allocation, likely at allocation time), because permissions are based on splitting each of a finite set of declared capabilities. The type system we propose in Chapter 5 has some similarities with deny-guarantee reasoning, because capabilities follow data flow rather than control flow.

Concurrent Abstract Predicates [71] (CAP) is a clever approach for allowing multi-threaded client programs to reason serially about concurrent modules, while allowing module implementations to be proven correct using rely-guarantee style reasoning over named regions. Each data structure can be abstracted [206] inside a separate named region, where assertions over shared (module internal) regions must be stable with respect to a fixed (declared by the module definition) set of actions for the module. Because action availability may be contingent upon heap state, one action may enable or disable others, which in turn makes certain assertions stable or unstable based on the action capabilities they are paired with (reminiscent of deny-guarantee permissions). This allows some stateful reasoning about the shared resource even in the presence of other interference, and combined with the module support [206], allows module clients to reason sequentially. This allows verifying correctness of lock implementations and concurrent (lock-based) set implementations. Declaring the set of actions up front is a more natural specification here than in deny-guarantee reasoning, because an abstract module necessarily specifies all actions on a data type. However, adding new operations requires updated proofs of stability for all exported assertions. More recent developments have generalized CAP for impredicative quantification over specifications [242] (allowing clients to choose the level of interference they expect when instantiating a module, such as thread-locality or monotonicity properties), and support for higher order stores (first-class functions) [242].

Explicit stabilization [269] is another approach to modularizing interference summaries. It is a technique for making stability checks explicit in rely-guarantee-style program logics, which otherwise typically have implicit side conditions for stability on all rules. However, the original formulation makes each program module's proof highly context-dependent. It is difficult to reuse most rely-guarantee proofs in other programs without redoing the proof; while the guarantee relation is dictated by the local module's actions, the (global) rely depends on all the rest of the program, which changes when reusing a library in a new

program. Explicit stabilization addresses this by allowing proof rule assertions p to be parameterized over a given rely R by referring to the weakest assertion stronger than p and stable over R, and the strongest weaker assertion that is stable over R. In a concurrent setting, this allows making rely-guarantee more modular by making judgments parametric in R, depending only on certain implications of the rely and placing restrictions on how R is instantiated. They also apply this in a serial setting, adapting RGsep [260] to libraries with client-shared hidden state, proving that interleaved chains of calls to the UNIX v7 memory manager do not interfere with each other.

Fictional SL [137] is another restricted form of rely-guarantee reasoning. It enables fictional disjointness, where "disjoint" assertions may in fact govern the same physical state, but the fictionally-disjoint assertions are defined together with operations on that state, and exported axioms for the fictionally-disjoint assertions are coupled with proofs that the applicable operations preserve any assertions whose storage physically overlaps. This allows, for example, using a machine-atomic bitfield to represent multiple flags.

Recently Dinsdale-Young et al. have proposed the Views Framework [70], which unifies many ideas from concurrent program logics into a single system, to allow proofs of various program logics and type systems by embedding into the framework. The key insight is to explicitly identify the local assertions made by each thread as a "view" of the global program state, and to encode the compatibility of these views by requiring them to form a commutative semigroup. Interference is represented by including it in the definition of a view. For example, a view could be defined as a set of machine states *closed under an interference relation* $\mathcal{R}$ characterizing interference by other threads. Proving soundness for a type system of program logic using Views requires translating all assertions / type environments into views. Assertions (in the logic or type system being embedded in Views) whose satisfying states do not form a view (are unstable under interference) are therefore invalid; they cannot be translated into an appropriate view. The authors demonstrate Views by (re-)proving soundness for simple type systems, CSL [43], rely-guarantee [139], a version of Owicki-Gries [204], and CAP [71]. Views provide a powerful tool for proving soundness of verification systems, but do not directly address verification of specific properties. This dissertation uses the Views Framework to prove soundness for two systems (Chapters 3 and

5).

2.5.2 Alias Interference

Only recently have researchers begun investigating interference summaries for treating interference among aliases, regardless of threading. Some of this is degenerate application of concurrent program logics, as in Wickerson's use of explicit stabilization to verify an early version of the UNIX memory manager [269] (CAP [71] and similar systems should also be usable in this way, but Wickerson's work is the first we know of that treats aliasing via a concurrent program logic). Reference immutability, as discussed in Section 2.4, is incidentally a very coarse version of an alias interference summary system: immutable referents may not be changed through other aliases, other permissions provide no such assurance. But others have focused specifically on aliases, concurrently with our own work in Chapter 4 and later. Arguably some of the separation-based alias-centric analyses like object propositions [195, 196] and typestate views [171] (Section 2.3) are a form of alias interference control (degenerating to non-interference).

In Chapter 4, we adapt classic rely-guarantee specifications to govern aliases and interference between them, rather than governing threads. We call the new development *rely-guarantee references*. The intuitions and key elements (rely, guarantee, stable assertions, and compatibility checks) carry over directly, but because the relations are interpreted as governing *all state reachable from a reference*, there are additional technical challenges to solve with recursive pointer data structures. Because we preserve the full generality of relational specifications, rely-guarantee references are very expressive, at the cost of potentially requiring solutions to difficult verification obligations (no worse than in the original rely-guarantee reasoning). In Chapter 5 we extend this for concurrent programming with only modest changes.

Concurrently with our work described in Chapter 4, Militão et al. explored an explicit notion of rely and guarantee for typestate [172, 173]. A typestate rely is a typestate all other aliases are expected to leave objects in after use. A guarantee is a typestate a given alias must leave the object in after use through that reference. They ensure a given object is seen

consistently by adopting a serialized sharing model (Section 2.3). Focusing on an object with a rely-guarantee typestate requires a dynamic typestate check, because it is generally unknown whether the focused reference or another reference with a different guarantee was last used. This offers a programming model where references generally agree on a disjoint-sum-of-typestate, refined by runtime inspection. They also include a notion of a typestate refinement based on the rely and guarantee typestates, but this must be convertible to the guarantee, and it requires all aliases have the same guarantee, making its utility unclear (this is based on examples [173]; their description of refinement is vague). Concurrently with their work, we proposed *rely-guarantee references* [103], described in Chapter 4.

Militão et al. have recently studied a more foundational calculus of substructural rely-guarantee protocols for use of values, including state in the heap [174]. The type system specifies protocols as a transition system of linear capabilities, and compatibility when splitting a protocol-governed object is checked via simulation — essentially finite-state model checking — rather than logical implication. Each capability governs only a single cell of the heap, but the transition systems may govern multiple capabilities, giving some ability to state protocols over recursive pointer structures. Because it is a capability calculus, the expressive power is limited in comparison to relational rely-guarantee reasoning as in Chapter 4 (in particular, it cannot state logical invariants preserved by protocols).

2.5.3 Interference for Correctness of Fine-Grained Data Structures

An important application of interference summaries is in verifying concurrent data structures,[8] particularly those with fine-grained locking or lock-free implementations [121]. Interference plays a crucial role because correctness of operations on concurrent data structures depends as much on other threads' actions as the actions of a particular operation. Interference summaries express these constraints and assumptions in a straightforward manner. This section gives a brief overview of a few important developments in reasoning about concurrent data structures using rely-guarantee reasoning and derivatives.

[8]Technically it is access to the data structure that is concurrent, but we follow standard abuse of terminology by referring to data structures designed for concurrent access as *concurrent data structures*.

Linearizability from Interference

Vafeiadis et al. [259] showed how rely-guarantee with binary postconditions is useful in proving linearizability [122] — a correctness condition that intuitively states that the externally observable effects of a concurrent data structure appear to take effect instantaneously at some point during its execution, rather than allowing other threads to observe partial states. Vafeiadis et al. added a "lifting" rule to the logic, which allows lifting an abstract specification to a command when both the guarantee and rely imply that if some boolean expression b's truth flips from false to true then the abstract operation has completed, and the proof observes this flip. This essentially lifts the local effects of the linearization point [122] to the method specification. Vafeiadis's thesis [256] extends this approach with auxiliary (ghost) state in RGSep, useful for capturing abstract state changes at linearization points. This technique was originally proposed without a soundness proof, but Liang and Feng recently proved soundness for the approach [157].

This work also highlights an important limitation of rely-guarantee reasoning with stable assertions: because a stable assertion is necessarily preserved by all actions in other threads, temporally-varying (i.e., unstable) properties like membership of a particular element in a concurrent set are inexpressible. This applies to most of the concurrent interference approaches discussed thus far (e.g., RGSep, SAGL, LRG), with the partial exception of CAP and deny-guarantee[9] reasoning; the concurrent rely-guarantee reference system we describe in Chapter 5 shares this limitation, though we show how to compensate for it by reasoning about abstract execution traces. A further consequence of this is the inability to verify most functional correctness. Proving linearizability requires relating the concurrent implementation to a full specification of functionality.

O'Hearn et al. describe the *Hindsight Lemma* [201], which assuming certain state and temporal invariants hold about linked list nodes, allows inference that some global state exists in which an operation can be linearized. The identify 18 invariants and step restrictions, both local (to nodes) and global (to the list structure) which enable the Hindsight Lemma

[9]In these systems, permission to affect membership of a given element could be confined to a single thread, but this would require dedicating one of the explicitly enumerated root capabilities to that element (unwieldy for large domains like the integers).

for the linked list used in a lock-free linearizable set implementation. The step restrictions amount to rely and guarantee specifications, and the invariants are stable for (preserved by) the step restrictions. Our concurrent rely-guarantee references in Chapter 5 can express these specifications fairly directly, though the algorithms in O'Hearn et al.'s paper require stronger concurrency primitives than we treat: they use atomic blocks.

Observational Refinement from Interference

Filipović et al. [84] propose observational refinement as a more natural alternative to linearizability (and show they are equivalent under certain assumptions): that operations on a concurrent data structure refine the behavior of a sequential equivalent. Both observational refinement and linearizability are defined over histories (program traces), and proving either requires some reasoning about causal dependence between different threads' events and possible commutativity of different threads' actions (similar intuition to Lipton's theory of *movers* [158]). Turon et al. [255, 254] build on this. First, Turon and Wand [255] describe a program logic for proving refinement using proof rules that imply the transition traces of a concrete (concurrent) implementation are a subset of the traces allowed by an abstract (e.g., serial) implementation. Core to their approach is a notion of structure invariants (supporting isolation of private — not-yet-shared — state) and interference relations fenced (in the LRG [82] sense) by that invariant. Turon et al. [254] extend the first-order approach to a semantic model for a higher-order typed language. The model directly encodes a "life story" of each data structure node: essentially a protocol describing (possibly abstract) state transitions for each node, amounting to a description of both possible actions and possible interference on each node. Subsequently they built up a powerful proof theory [253] for this model, capable of proving — within the logic — that an implementation of a lock free data structure not only meets its specification but its behavior refines a coarse-grained implementation. Chapter 5 develops a method for proving a procedure refines a relational input-output specification, but does not make the same stronger restriction that observational refinement implies about other threads observing intermediate states.

Reasoning Abstractly about Other Threads

Ley-Wild and Nanevski propose *subjective auxiliary state* [156] for reasoning about concurrent data structures, where state for objects representable as a commutative monoid may be represented as a split of state contributions from other threads and from the local thread. This allows reasoning almost sequentially about a given thread's contributions to state (e.g., how many increments the local thread has performed) while assuming arbitrary (monoid-compatible) interference on the other fragment. Nanevski et al. recently proposed an extension of this idea [187], adding RGSep/LRG-style [260, 82] transitions over local and remote portions of the monoid state, along with rules for constructing larger state transition systems from smaller ones.

2.6 Implementing Verification Systems

Thus far we have avoided the question of how to implement the proof systems described in such a way as to actually verify programs! Fortunately, this is not wholly unexplored. There are a few main approaches to implementing these systems. First is the most obvious: implementation from scratch, over some language of choice or a custom language. Most of the work above that was implemented chose this approach, using algorithmic proof or type systems [275, 276, 68, 112] or generating verification conditions that are handed off to a theorem prover [21, 46, 72, 45, 191].

Much of the practical evaluation from the program logic community comes from custom implementations of frameworks. These often use formal proof rules to inspire a symbolic execution generating verification conditions as in the Smallfoot [21] implementation of SL and the SmallfootRG [46] implementation of RGSep [260] (where the verification conditions are separation logic entailments). Writing specifications for things like interference on shared state, however, can be onerous, leading Vafeiadis to implement action inference [257] for RGSep and SmallfootRG specifications, using symbolic execution to overapproximate interference on shared state.

Embedding in Type Theory Another successful approach to implementing imperative verification tools is to embed them in dependent type theories [62, 60, 213, 165], or more

practically in proof assistants [59, 22, 1, 194]. Deep embeddings — where programs in the modeled language are elements of some data type representing programs — such as those used to mechanically verify a proof system is sound (e.g., Nieto's verification or rely-guarantee soundness [193]), are theoretically usable for real program verification, but are usually quite unwieldy. Shallow embeddings — which essentially implement the target language as a library with dependent types implementing the appropriate proof rules — are much more usable due to integration with a more full-featured host language, and expose proof obligations more directly to any automation machinery available in the proof assistant. The combination of rich tactic languages and computation in types (for unfolding structure definitions) has proven particularly effective for separation logic [20, 50].

The most successful instance of this is Ynot [189, 52, 162, 190], which is a shallow embedding of Hoare Type Theory [188, 185, 212, 186, 190] as a monadic DSL inside Coq [59, 22]. It uses an indexed monadic type to simulate a Hoare triple, with appropriate bind (sequencing) operations to enforce that the postcondition of one statement matches the precondition in a sequential composition. It also leverages Coq's powerful code extraction mechanism to extract OCaml and Haskell code from Ynot programs. The crowning achievement of this work is the demonstration of how effectively proof obligations for HTT can be semi-automatically discharged using Coq's support for user-defined proof tactics [52], requiring user intervention typically only for unfolding logical representations of data structures, and choosing when to apply induction. The automation hinges on effective strategies for matching corresponding assertions in large separating conjunctions (though as Nanevski points out [190], the automation has weak support for separating implication).

This dissertation relies heavily on implementation by shallow embedding. Chapters 4 and 5 describe verifications performed using a shallow embedding into Coq. Chapter 6 relies on a shallow embedding into Liquid Haskell [261, 262, 263], an extension of Haskell with refinement types. Chapter 6 describes Liquid Haskell in detail.

2.7 Relating Sequential and Concurrent Correctness

One of the most elegant results in concurrent program verification is the fact that some proof systems allow correctness of concurrent programs to follow almost directly from sequential

correctness, along with a few additional restrictions imposed only when threads are created, destroyed, or communicate with each other. If any concurrency-specific rules exist, they employ the same verification principles used in sequential verification. Concurrent Separation Logic (CSL) is perhaps the canonical example of this approach to safe concurrency: as soon as separation logic was shown effective in the sequential case, the appeal of the concurrent case was immediately apparent, given that the small-footprint approach to SL assertions lends itself naturally to threads operating on disjoint heap portions. The only proof system considerations for concurrency are then when threads are spawned (splitting the assertion and state for the child threads), and when locks are acquired and released (transferring ownership of state satisfying an invariant between the local thread and the critical section). Essentially, the safe concurrency checks in many of the systems above can be abstracted to

$$\textsc{SafePar}\text{-}P_A\text{-}P_B\ \frac{\Gamma_1 \vdash C_1 \dashv \Gamma_1' \qquad \Gamma_2 \vdash C_2 \dashv \Gamma_2' \qquad P_A(\Gamma_1) \qquad P_B(\Gamma_2)}{\Gamma_1 * \Gamma_2 \vdash C_1 \parallel C_2 \dashv \Gamma_1', \Gamma_2'}$$

by treating Γ as interchangeable for program logic assertions or type environments. The rule relies on two predicates on environments / assertions, P_A and P_B (to allow some asymmetry between threads), which enforce restrictions on parallel composition particular to each system. For disjoint concurrent separation logic or the Views Framework, $P_A = P_B = \mathsf{True}$ (Γ_1 and Γ_2 are stable over any possible interference). For the concurrency rules in our work on reference immutability for safe concurrency [105] (Chapter 3), P_A and P_B either both prohibit writable references (data parallel + read sharing) or P_A only permits isolated (a variant of external uniqueness explained in Chapter 3 and immutable references and P_B permits anything (structured asynchronous block on separated data). This form essentially permits formulating the safe concurrency check as a combination of spatial separation ($*$) and additional predicates on potentially-overlapping resources. Verification of C_1 and C_2 proceed as for any other program, without explicit consideration for the environment.

2.8 Dependent Type Theory

Much of this thesis (Chapters 4, 5, and 6) builds on the foundation of dependent type theory. A full introduction to dependent type theory is well beyond the scope of this thesis, though

familiarity would be helpful. We give a brief overview here of the theory and relation to logic, and offer pointers to further reading that we have found useful.

Dependent type theory is an extension of the simple type theory [53] of the simply-typed lambda calculus to a full higher order intuitionistic predicate logic.[10] The main rules present in a dependent type theory that are absent or simpler in simpler systems are:

$$\Pi\text{-I}\quad \frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash M : B[x]}{\Gamma \vdash (\lambda x.\, M) : \Pi_{x:A} B[x]} \qquad \Pi\text{-E}\quad \frac{\Gamma \vdash M : \Pi_{x:A} B[x] \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\textsc{Conv}\quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \approx B}{\Gamma \vdash M : B}$$

The rules for introduction and elimination of dependent products (Π types) are important because they may project values into types. This is what permits dependent type theory to act as a predicate logic. The conversion rule is also important, as it embodies the notion that any two terms (or types) with the same normal form under β-reduction ($\approx$) are considered equivalent, making it easier to prove properties about the computational behavior of a term. For our purposes, one of the most significant reasons that dependent type theory is of interest is that the programming language and logic are part of the same term language, reducing the number of concepts to work with.

In terms of interpreting type theory as a logic, the general correspondence is given by the Curry-Howard correspondence [130], also known as the *propositions as types* principle, where a proposition is expressed using a type of a given form, and a proof is given by exhibiting a term of the corresponding type. Unprovable statements correspond to uninhabited types.

[10] Interpretations of classical logic, for example where the law of excluded middle corresponds to use of the control operator `call/cc`, exist but are relatively unused. Most intuitionistic type theories remain compatible with the addition of classical axioms, though these combinations are often delicate [217, 23].

The main correspondences are:

Predicate Logic	Type Theory
$\forall x.\, P$	$\Pi_{x:A} P$
$\exists x.\, P$	$\Sigma_{x:A} P$
True	$\top$
False	$\bot$
$A \wedge B$	$\Sigma_{x:A} B$ where x does not occur in B
$A \vee B$	$A + B$

For classic introductions, we refer the reader to the early work of Per Martin-Löf [165]; Coquand and Huet [62, 60]; Harper et al. [114, 113]; and later synthesis of these lines of work [15]. Some good expository texts covering some more advanced material include Thompson's *Type Theory and Functional Programming* [248], Nordstrom et al.'s *Programming in Martin-Löf's Type Theory*, and the more recent but less formal *Homotopy Type Theory* book [217] which covers recent developments such as univalence [2] and higher inductive types [217] based on homotopy theory [226]. Many extensions to base theories exist, including universes [115], inductive types [74, 210] (as well as the inductive-recursive [76] and inductive-inductive [94] types we use in Chapters 4 and 5), coinductive types [61, 168], and more.

2.8.1 Dependent Types and Mutable State

Some instances of combining dependent types and mutable state were already described with the style of reasoning they employ, but it is worth discussing these again from a type theoretical perspective. Integrating dependent types and mutable state is a long-standing research challenge. The heart of the challenge is similar to the general challenge of reasoning about imperative programs: how can we reconcile types whose interpretation depends on runtime values with features that can change the values influencing unrelated types? The primary approaches employed have largely followed two paths.

First, much work on dependent types takes the approach of applying refinement types [95, 243] to select acceptable subsets of values (for example, the type of positive integers $\{x :$

$\int |x > 0\}$), with the additional restriction that refinements may only mention the data structure (node) being refined, along with immutable data (i.e., immutable function arguments and constants). This is the approach taken by DML [272, 273], ATS [48], and X10's constrained types [198]. A common additional restriction is on the form of the refinement formulas to a theory that is effectively decidable by an SMT solver (as in Liquid Types [230]). This has obvious limitations on expressiveness (namely, only individual nodes may be refined, and refinements may not refer to other parts of the heap).

The second major approach is to use dependent types to perform a shallow embedding of a program logic, and apply the program logic to imperative state (which can store dependently-typed values). This is the approach taken by HTT [188, 185, 190], in particular in the form of its Ynot embedding in Coq [189, 52], and the Dijkstra monad used by F* [245]. This is more of a use of dependent types to express a program logic than actual support for dependently typed programming over state. In particular, HTT offers a low-level imperative programming model with pointer arithmetic, and a high-level functional programming language with dependent types — a peculiar combination. F* exposes proper reference types rather than raw locations, along with some support for enforcing and relying upon monotonicity properties on the heap, which permits some more flexible programming idioms.

Chapter 3

UNIQUENESS AND REFERENCE IMMUTABILITY FOR SAFE PARALLELISM

> My thesis is, that a helpful programming methodology should be closely tied to correctness concerns.
>
> Dijkstra, EWD 288

The results of this chapter were originally presented in the paper *Uniqueness and Reference Immutability for Safe Parallelism* [105] from OOPSLA 2012, and its extended technical report [106]. These results are the product of collaboration with Matthew Parkinson, Jared Parsons, Aleksandra Culver,[1] and Joe Duffy.

In a concurrent program, a side-effect in one thread can affect the behavior of another thread. This makes the program hard to understand as the programmer must consider the context in which their thread executes. In a setting with a relaxed memory-consistency model, even understanding the possible interactions is non-trivial [232].

One approach to restricting, or taming, these side effects for easier maintenance and understandability is to prohibit interference between threads, via non-interference and/or read-sharing. This chapter enforces non-interference and read-sharing between threads in a lightweight manner by extending *reference immutability* [28, 252, 275, 276], which uses permission type qualifiers to control object mutation. The advantage to this, as compared to other approaches to static data race freedom [85, 86, 87, 3, 35, 34] is that we obtain a method that not only prohibits data races, but is also very similar in flavor to its sequential variant: every construct in our language (beyond thread creation) is immediately useful in

[1] Under a previous name

sequential programs as well, making for a concurrent programming system that is a natural extension to serial programming.

We add to reference immutability a notion of isolation in the form of an extension to external uniqueness [112]. We support the natural use of isolation for object immutability (making objects permanently immutable through all references). But we also show a new use: to *recover isolation* or strengthen immutability assumptions without any alias tracking. To achieve this we give two novel typing rules, which allow recovering isolated or immutable references from arbitrary code checked in environments containing only isolated or immutable inputs.

We provide two forms of parallelism:

Symmetric Assuming that at most one thread may hold writable references to an object at a given point in time, then while all writable references in a context are temporarily forgotten (framed away, in the separation logic sense [199, 221]), it becomes safe to share all read-only or immutable references among multiple threads, in addition to partitioning externally-unique clusters between threads.

Asymmetric If all data accessible to a new thread is immutable or from externally-unique clusters which are made inaccessible to the spawning thread, then the new and old threads may run in parallel without interference.

We provide an extended version of the type system with polymorphism over reference immutability qualifiers. This maintains precision for instantiated uses even through rich patterns like iterators, which was not possible in previous work [275].

There are several aspects of this work which we are the first to do. We are the first to give a denotational meaning to reference immutability qualifiers. We are the first to formalize the use of reference immutability for safe parallelism. We are the first to describe industry experience with a reference immutability type system. We are also the first significant new development of a sound program verification approach developed on top of the Views Framework [70], which had previously only been used to reformulate proofs of existing systems.

3.1 Reference Immutability, Uniqueness, and Parallelism

While reference immutability was first introduced in Section 2.4.1, we give a more in depth overview here, tailored to our approach to data race freedom. Reference immutability is based on a set of permission-qualified types. Our system has four qualifiers:

writable: An "ordinary" object reference, which allows mutation of its referent.

readable: A read-only reference, which allows no mutation of its referent. Furthermore, no heap traversal through a read-only reference produces a writable reference (writable references to the same objects may exist and be reachable elsewhere, just not through a readable reference). A readable reference may also refer to an immutable object.

immutable: A read-only reference which additionally notes that its referent can never be mutated through any reference. Immutable references may be aliased by read-only or immutable references, but no other kind of reference. All objects reachable from an immutable reference are also immutable.

isolated: An external reference to an externally-unique object cluster. External uniqueness naturally captures thread locality of data. An externally-unique aggregate is a cluster of objects that freely reference each other, but for which only one external reference into the aggregate exists. We define isolation slightly differently from most work on external uniqueness because we also have immutable objects: all paths to *non-immutable* objects reachable from the isolated reference pass through the isolated reference. We allow references out of the externally-unique aggregate to immutable data because it adds flexibility without compromising our uses for isolation: converting clusters to immutable, and supporting non-interference among threads (see Figure 3.1). This change in definition does limit some traditional uses of externally-unique references that are not our focus, such as resource management tasks.

The most obvious use for reference immutability is to control where heap modification may occur in a program, similar to the owner-as-modifier discipline in ownership and universe

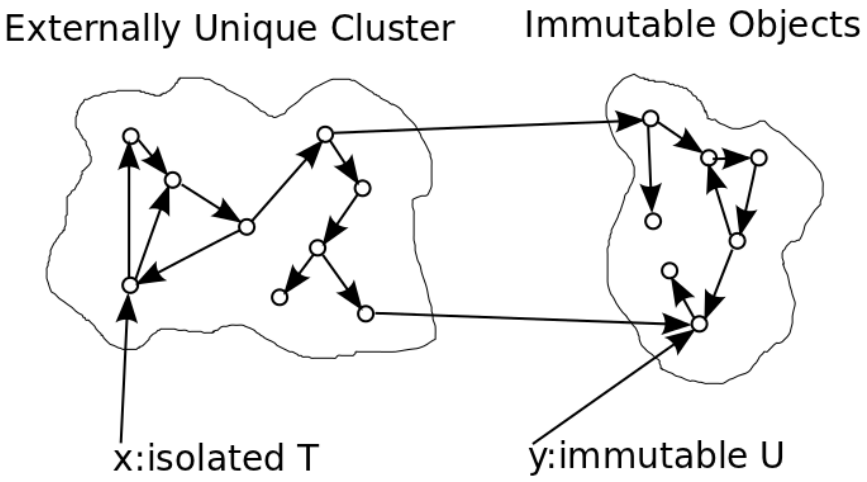


Figure 3.1: External uniqueness with immutable out-references.

type systems [68]. For example, a developer can be sure that a library call to a static method with the type signature

```
int countElements(readable ElementList lst);
```

will not modify the list or its elements (through the `lst` reference). Accessing any field of the argument `lst` through the readable reference passed will produce other readable (or immutable) results. For example, a developer could not implement `countElements` like so:

```
int countElements(readable ElementList lst)
{ lst.head = null; return 0; }
```

because the compiler would issue a type error. In fact, any attempt within `countElements()` to modify the list would result in a type error, because `lst` is deeply (transitively) read-only, and writes through read-only references are prohibited.

This type of caller-callee read-only behavior enforcement is the essential idea of all reference immutability techniques described in Section 2.4.1. The system presented in this chapter differs from those in two significant ways. First, this system is extended for strong isolation of groups of objects. This naturally supports thread-local data and ownership transfer. Second, the systems outlined in Section 2.4.1 all permit various ways to make the transitive read-only restriction stop at some point, in order to support certain useful

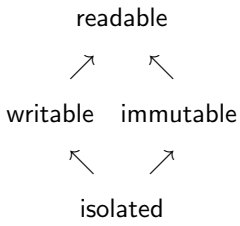


Figure 3.2: Qualifier conversion/subtyping lattice.

idioms. We must take a stronger stance on the foundations of reference immutability (truly transitive read-only references) than previous systems because we wish to reason about thread-interference solely in terms of mutability permissions, which would be significantly complicated if some data could be mutated through any of its aliases.

3.1.1 Conversion from Isolated

The isolated qualifier is atypical in reference immutability work, and is not truly a permission for (im)mutability in the purest sense. In fact, we require that isolated references be converted through subtyping to another permission before use, according to the type qualifier hierarchy in Figure 3.2.

isolated references are particularly important in our system for two reasons. First, they naturally support safe parallelism by partitioning mutable data amongst threads. The threads[2] in the following example cannot interfere with each other, because the object graphs they operate on and can mutate are disjoint:

```
isolated IntList l1 = ...;
isolated IntList l2 = ...;
{ l1.map(new Incrementor()); } || { l2.map(new Incrementor()); }
```

Second, the control of aliasing allows *conversion* of whole externally-unique object clusters. If there are no external references besides the isolated reference, then the whole object graph (up to immutable objects) can be converted at once. An isolated reference (and object graph)

[2]We use || for structured parallelism, and the formal system does not have dynamic thread creation.

can trivially be converted to writable, by essentially surrendering the aliasing information:

```
isolated IntList l = ...;
// implicitly update l's permission to writable
l.head = ...;
```

Or an isolated graph can be converted to immutable; as with any form of strong update, the decision to treat the whole object graph as immutable is localized:

```
isolated IntList l = ...;
// implicitly update l's permission to immutable
immutable IntList l2 = l;
l.head = ...; // Type Error!
```

The type system is flow sensitive, so although `l` was initially isolated after the assignment to `l2` it has been coerced to immutable and thus cannot be written to.

3.1.2 Recovering Isolation

A key insight of our approach is that converting an isolated reference to writable does not require *permanently* surrendering the aliasing information. In particular, if the input type context for an expression contains only isolated and immutable objects, then if the output context contains a single writable reference, we can convert that reference *back to isolated*. Consider the following method:

```
isolated IntBox increment(isolated IntBox b) {
  // implicitly convert b to writable
  b.value++;
  // convert b *back* to isolated
  return b;
}
```

The first conversion from isolated to writable occurs naturally by losing aliasing information. The second conversion is safe because if one writable reference is left when the initial context contained only isolated and immutable references, that reference must either refer to an object that was not referenced from elsewhere on entry, or was freshly allocated (our core language and prototype do not allow mutable global variables).

This flexibility is especially useful for algorithms that repeatedly map destructive operations over data in parallel. By keeping data elements as isolated, the map operations naturally parallelize, but each task thread can internally violate uniqueness, apply the updates, and recover an isolated reference for the spawning context for later parallelization (Section 3.1.5).

Recovering isolation is reminiscent of borrowing — allowing temporary aliases of a unique reference, often in a scope-delimited region of program text. The main advantage of recovery is that unlike all borrowing designs we are aware of, recovery requires no tracking or invalidation of specific references or capabilities as in other work [112, 42]. Of course this is a result of adding reference immutability, so recovery is not a stand-alone replacement for traditional borrowing; it is an additional benefit of reference immutability.

We also see two slight advantages to our recovery approach. First, a single use of recovery may subsume multiple uses of a scoped approach to borrowing [203], where external uniqueness is preserved by permitting access to only the interior of a particular aggregate within a lexically scoped region of code. Of course, scopeless approaches to borrowing exist with more complex tracking [112, 42]. Second, no special source construct is necessary beyond the reference immutability qualifiers already present for parallelism.

3.1.3 Recovering Immutability, and Cycles of Immutable Objects

Another advantage of using isolated references is that the decision to make data immutable can be deferred (arbitrarily). This makes constructing cycles of immutable objects easy and natural to support. The mechanism for converting an isolated reference to immutable is similar to recovering isolation, with the natural direct conversion being a special case. If the input context when checking an expression contains only isolated and immutable references, and the output context contains one readable reference (or in general, multiple readable references), then the readable referent must be either an already-immutable object or an object not aliased elsewhere that it is safe to now call immutable. The simplest case of this (equivalent to direct conversion) is to frame away all references but one, convert to readable, and then recover immutability:

```
immutable IntBox freeze(isolated IntBox b) {
  // implicitly convert b to readable
  // implicitly recover immutability;
  // the input context was all isolated
  return b;
}
```

Creating cycles of immutable objects is then simply a matter of restricting the input to a conversion to only isolated and immutable data, then recovering. This can even include recovering immutability from regular code:

```
// The default permission is writable
CircularListNode make2NodeList() {
  CircularListNode n1 = new CircularListNode();
  CircularListNode n2 = new CircularListNode();
  n1.next = n2; n1.prev = n2;
  n2.next = n1; n2.prev = n1;
  return n1;
}
...
immutable l = make2NodeList();
```

Here the method has no inputs and it returns a writable value, so at the call site anything it returns can be considered readable, then recovered to immutable (or directly recovered to isolated).

Prior reference immutability systems [275] required building immutable cyclic data structures in the constructor of one object, using extensions to pass a partially-initialized object during construction as (effectively) immutable to the constructor of another object. Our use of isolated with recovery means we do not need to explicitly model the initialization period of immutable structures.

While we have been using closed static method definitions to illustrate the recovery rules, our system includes a frame rule [199, 221], so these conversions may occur in localized sections of code in a larger context.

3.1.4 Safe Symmetric Parallelism

Fork-join concurrency is deterministic when neither forked thread interferes with the other by writing to shared memory. Intuitively, proving its safe use requires separating read and write effects, as in Deterministic Parallel Java (DPJ) [35]. With reference immutability, a simpler approach is available that does not require explicit region management, allowing much of the same expressiveness with simpler annotation (see Section 3.6).

If neither forked thread requires any writable reference inputs to type check, then it is safe to parallelize, even if the threads share a readable reference to an object that may be mutated later, and even if threads receive isolated references.

```
x = new Integer(); x.val = 3; y = x; z = x;
// y and z are readable aliases of x
a = new Integer(); b = new Integer();
// a and b are isolated
// frame away writable references (x)
a.val = y.val; || b.val = z.val;
// get back writable references (x)
x.val = 4;
```

After joining, `x` may be "unframed" and the code regains writable access to it. Safety for this style of parallelism is a natural result of reference immutability, but proving it sound (race free) requires careful handling of coexisting writable references to the temporarily-shared objects.

We require that each thread in the parallel composition receives disjoint portions of the stack, though richer treatments of variable sharing across threads exist [205, 219].

3.1.5 Safe Asymmetric Parallelism

C# has an `async` construct that may execute a block of code asynchronously via an interleaving state machine or on a new thread [26], and returns a handle for the block's result in the style of promises or futures. A common use case is asynchronously computing on separated state while the main computation continues. Our formal system models the asymmetric data sharing of this style of use on top of structured parallelism. The formal

system (Section 3.2) does not model the first-class join; in future work we intend to extend this rule to properly isolate `async` expressions.

A natural use for this style of parallelism is to have the asynchronous block process a limited data set in parallel with a "main" thread's execution. One definition of "limited" is to restrict the "worker" thread to isolated and immutable data, allowing the "main" thread to proceed in parallel while retaining writable references it may have.

```
writable Integer x = ...;
// Construct isolated list of isolated integers
y = new IsolatedIntegerList();
... // Populate list
f = new DoStuffFunc();
// Map in parallel with other work
y.map(f); || x.val = 3;
```

This code also demonstrates the flexibility of combining the rules for recovering isolated or immutable references with parallelism. In the left thread, `f` and `y` are both isolated on entry, and the rule for recovering an isolated reference can be applied to `y` at that thread's finish. Thus, when the threads join, `y` is again isolated, and suitable for further parallelization or full or partial conversion to immutable.

3.2 Types for Reference Immutability and Parallelism

We describe a simple core imperative, object-oriented language in Figure 3.3. Commands (statements) include standard field and variable assignments and reads, sequencing, loops, non-deterministic choice (to model conditional statements) and fork-join style parallelism. Our language also includes a destructive read, $x = \texttt{consume}(y.f)$, which reads the field, $y.f$, stores its value in x, and then updates the field to `null`. Our types include primitive types and permission-qualified class types. We include the four permissions from Section 3.1: readable, writable, isolated, and immutable. This section focuses on the language without methods, which are added in Section 3.2.3. Polymorphism, over both class types and permissions, is described in Section 3.4.

One of our primary goals for this core system is to understand the design space for source

languages with reference immutability and concurrency in terms of an intermediate-level target language. This approach permits understanding source-level proposals for typing higher level language features (such as closures) in terms of translation to a well-typed intermediate form (such as the function objects C# closures compile into), rather than independently reasoning about their source-level behavior.

The heart of reference immutability is that a reference's permission applies transitively. Any new references acquired through a reference with a given permission cannot allow modifications that the root reference disallows. We model this through a *permission combining* relation $\triangleright$, borrowing intuition and notation from universe types' "viewpoint adaptation" [68]. We define $\triangleright$ and lift it to combining with types in Figure 3.3.

Generally speaking, this relation propagates the weakest, or least permissive, permission. Notice that there are no permission-combining rules for isolated receivers and non-immutable fields; this reflects the requirement that accessing an isolated object graph generally requires upcasting variables first and accessing isolated fields requires destructive reads. Also notice that any combination involving immutable permissions produces an immutable permission; any object reachable from an immutable object is also immutable, regardless of a field's declared permission.

We use type environments Γ, and define subtyping on environments ($\vdash \Gamma \prec \Gamma$) in terms of subtyping for permissions ($\vdash p \prec p$), class types ($\vdash T \prec T$), and permission-qualified types ($\vdash t \prec t$) in Figure 3.4.

Figure 3.5 gives the core typing rules. These are mostly standard aside from the treatment of unique references. A destructive field read (T-FieldConsume) is fairly standard, and corresponds dynamically to a basic destructive read: as the command assigns `null` to the field, it is sound to return an isolated reference. Writes to isolated fields (T-FieldWrite) and method calls with unique arguments (T-Call) treat the isolated input references as affine resources, consumed by the operation. We use a metafunction RemIso() to drop "used" isolated references:

$$\mathsf{RemIso}() : \Gamma \to \Gamma$$
$$\mathsf{RemIso}(\Gamma) = \mathsf{filter}\ (\lambda x.\, x \neq \mathsf{isolated}\ \_)\ \Gamma$$

Metavariables

a	atoms
C	command (statement)
w, x, y, z	variables
t, u	types
T, U	class type
TD	class type declaration
cn	class name
p	permission
fld	field declaration
$meth$	method declaration
f, g	field names
m	method names
n, i, j	nat (indices)

Syntax

$$
\begin{array}{lll}
a & ::= & \\
 & & \mid x = y \\
 & & \mid x.f = y \\
 & & \mid x = y.f \\
 & & \mid x = \texttt{consume}(y.f) \\
 & & \mid x = y.m(z_1, ..., z_n) \\
 & & \mid x = \texttt{new}\ t() \\
 & & \mid \mathsf{return}\ x \\
C & ::= & a \mid \texttt{skip} \mid C; C \mid C + C \mid C \| C \mid C^* \\
p & ::= & \mathsf{readable} \mid \mathsf{writable} \mid \mathsf{immutable} \mid \mathsf{isolated} \\
T & ::= & cn \\
TD & ::= & \texttt{class}\ cn\ [<: T2]\ \{ field*\ \ meth* \} \\
fld & ::= & t\ fn \\
meth & ::= & t\ m(t_1\ x_1, ..., t_n\ x_n) p\{\ C; \mathsf{return}\ x;\ \} \\
t & ::= & \texttt{int} \mid \texttt{bool} \mid p\ T \\
\Gamma & ::= & \epsilon \mid \Gamma, x : t
\end{array}
$$

$\triangleright$: Permission $\rightarrow$ Permission $\rightarrow$ Permission

$$
\begin{array}{rcl}
\mathsf{immutable} \triangleright \_ & = & \mathsf{immutable} \\
\_ \triangleright \mathsf{immutable} & = & \mathsf{immutable} \\
\mathsf{readable} \triangleright \mathsf{writable} & = & \mathsf{readable} \\
\mathsf{readable} \triangleright \mathsf{readable} & = & \mathsf{readable} \\
\mathsf{writable} \triangleright \mathsf{readable} & = & \mathsf{readable} \\
\mathsf{writable} \triangleright \mathsf{writable} & = & \mathsf{writable}
\end{array}
$$

$$
\begin{array}{l}
p \triangleright \texttt{int} = \texttt{int} \\
p \triangleright \texttt{bool} = \texttt{bool} \\
p \triangleright (p'\ T) = (p \triangleright p')\ T
\end{array}
$$

Figure 3.3: Core language syntax.

$$\boxed{\vdash p \prec p'} \quad \frac{}{\vdash p \prec p} \quad \frac{}{\vdash p \prec \mathsf{readable}} \quad \frac{}{\vdash \mathsf{isolated} \prec p}$$

$$\boxed{\vdash T \prec T'} \quad \frac{\texttt{class}\ \ c\ <:\ d\ \{\overline{fld}\ \overline{meth}\ \} \in P}{\vdash c \prec d}\ \text{S-Decl}$$

$$\boxed{\vdash t_1 \prec t_2} \quad \frac{\vdash p \prec p'}{\vdash p\ T \prec p'\ T}\ \text{S-Perm} \quad \frac{\vdash T \prec T'}{\vdash p\ T \prec p\ T'}\ \text{S-Type}$$

$$\frac{}{\vdash t \prec t}\ \text{S-Reflexive} \quad \frac{\vdash t_1 \prec t_2 \quad \vdash t_2 \prec t_3}{\vdash t_1 \prec t_3}\ \text{S-Trans}$$

$$\boxed{\vdash \Gamma \prec \Gamma'} \quad \frac{}{\epsilon \prec \epsilon}\ \text{S-Empty} \quad \frac{\vdash \Gamma \prec \Gamma' \quad \vdash t \prec t'}{\vdash \Gamma, x : t \prec \Gamma', x : t'}\ \text{S-Cons} \quad \frac{\vdash \Gamma \prec \Gamma'}{\vdash \Gamma, x : t \prec \Gamma'}\ \text{S-Drop}$$

Figure 3.4: Subtyping rules

$$\boxed{\Gamma_1 \vdash C \dashv \Gamma_2}$$

$$\frac{t \neq \mathsf{isolated}\ \_}{x : \_, y : t \vdash x = y \dashv y : t, x : t}\ \text{T-AssignVar} \qquad \frac{}{\vdash x = \texttt{new}\ T() \dashv x : \mathsf{isolated}\ T}\ \text{T-New}$$

$$\frac{t'\ f \in T \quad p \neq \mathsf{isolated} \vee t' = \mathsf{immutable}\ \_ \quad t' \neq \mathsf{isolated}\ \_ \vee p = \mathsf{immutable}}{x : \_, y : p\ T \vdash x = y.f \dashv y : p\ T, x : p \rhd t'}\ \text{T-FieldRead}$$

$$\frac{t\ f \in T}{y : \mathsf{writable}\ T, x : t \vdash y.f = x \dashv y : \mathsf{writable}\ T, \mathsf{RemIso}(x : t)}\ \text{T-FieldWrite}$$

$$\frac{\mathsf{isolated}\ T_f\ f \in T}{y : \mathsf{writable}\ T \vdash x = \texttt{consume}(y.f) \dashv y : \mathsf{writable}\ T, x : \mathsf{isolated}\ T_f}\ \text{T-FieldConsume}$$

$$\frac{}{x : \_ \vdash x = n \dashv x : \texttt{int}}\ \text{T-Int} \quad \frac{}{x : \_ \vdash x = b \dashv x : \texttt{bool}}\ \text{T-Bool} \quad \frac{}{x : \_ \vdash x = \mathsf{null} \dashv x : p\ T}\ \text{T-Null}$$

$$\frac{\begin{array}{c} t'\ m(\overline{u'\ z'})\ p' \in T \qquad \vdash p \prec p' \qquad \overline{\vdash u \prec u'} \\ p = \mathsf{isolated} \implies t \neq \mathsf{readable}\ \_ \wedge t \neq \mathsf{writable}\ \_ \wedge \mathsf{IsoOrImm}(\overline{z : t}) \wedge p' \neq \mathsf{immutable} \end{array}}{y : p\ T, \overline{z : u} \vdash x = y.m(\overline{z}) \dashv y : p\ T, \mathsf{RemIso}(\overline{z : t}), x : t'}\ \text{T-Call}$$

$$\frac{\Gamma_1 \prec \Gamma_1' \quad \Gamma_1' \vdash C \dashv \Gamma_2' \quad \Gamma_2' \prec \Gamma_2}{\Gamma_1 \vdash C \dashv \Gamma_2}\ \text{T-SubEnv} \quad \frac{\Gamma_1 \vdash C \dashv \Gamma_2}{\Gamma, \Gamma_1 \vdash C \dashv \Gamma, \Gamma_2}\ \text{T-Frame} \quad \frac{\Gamma \vdash C \dashv \Gamma}{\Gamma \vdash C* \dashv \Gamma}\ \text{T-Loop}$$

$$\frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_2 \vdash C_2 \dashv \Gamma_3}{\Gamma_1 \vdash C_1; C_2 \dashv \Gamma_3}\ \text{T-Seq} \quad \frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_1 \vdash C_2 \dashv \Gamma_2}{\Gamma_1 \vdash C_1 + C_2 \dashv \Gamma_2}\ \text{T-Branch}$$

$$\frac{\Gamma, y : t', x : t, \Gamma' \vdash C \dashv \Gamma''}{\Gamma, x : t, y : t', \Gamma' \vdash C \dashv \Gamma''}\ \text{T-Shuffle}$$

Figure 3.5: Core typing rules.

$$\boxed{\vdash P} \quad \frac{\forall c \in \mathsf{Classes}(P).\ P \vdash c \quad \epsilon \vdash \mathsf{Expression}(P) \dashv \Gamma \quad \mathsf{ClassesOnce}(P)}{\vdash P} \text{ T-Program}$$

$$\boxed{P \vdash TD} \quad \frac{\mathsf{FldsOnce}(\overline{fld}) \quad \mathsf{MethsOnce}(\overline{meth}) \quad \forall fld \in \overline{fld}.\ P; TD \vdash fld \quad \forall meth \in \overline{meth}.\ P; TD \vdash meth}{P \vdash \texttt{class}\ cn\ [<: T2]\ \{\overline{fld}\ \overline{meth}\ \}} \text{ T-Class}$$

$$\boxed{P; TD \vdash fld} \quad \frac{TD = \texttt{class}\ cn\ [<: T2]\ \{\overline{fld}\ \overline{meth}\ \} \quad f \notin \mathsf{Fields}(\mathsf{ParentClasses}(T2))}{P; TD \vdash p\ T\ f} \text{ T-Field}$$

$$\boxed{P; TD \vdash meth}$$

$$\frac{\begin{array}{l} TD = \texttt{class}\ cn\ [<: T2]\ \{\overline{fld}\ \overline{meth}\ \} \qquad \forall t', \overline{x'}, \overline{t'}, p'.\ t'\ m(\overline{t'}\ \overline{x'})\ p' \notin T2 \\ p \neq \mathsf{isolated} \qquad \forall i \in [1 \ldots n].\ P \vdash t_i \qquad P \vdash t \\ \texttt{this} : p\ cn, \overline{t}\ \overline{x} \vdash C; \mathsf{return}\ x \dashv \texttt{result} : t \end{array}}{P; TD \vdash t\ m(\overline{t}\ \overline{x})\ p\ \{\ C; \mathsf{return}\ x;\ \}} \text{ T-Method1}$$

$$\frac{\begin{array}{l} TD = \texttt{class}\ cn\ [<: T2]\ \{\overline{fld}\ \overline{meth}\ \} \qquad t'\ m(\overline{t'}\ \overline{x'})\ p' \in T2 \qquad P \vdash t \prec t' \\ P \vdash p' \prec p \qquad P \vdash \overline{t'} \prec \overline{t} \qquad p \neq \mathsf{isolated} \qquad \forall i \in [1 \ldots n].\ P \vdash t_i \\ P \vdash t \qquad \texttt{this} : p\ cn, \overline{t}\ \overline{x} \vdash C; \mathsf{return}\ x \dashv \texttt{result} : t \end{array}}{P; TD \vdash t\ m(\overline{t}\ \overline{x})\ p\ \{\ C; \mathsf{return}\ x;\ \}} \text{ T-Method2}$$

Figure 3.6: Program typing

This is a slight inconvenience in the core language, but the implementation supports `consume` as a first class effectful expression. The method rule is otherwise straightforward aside from calls on isolated receivers (Section 3.2.3). We also provide structural rules to allow these rules to be used in more general contexts (last two rows of Figure 3.5). The definition of well-formed programs (Figure 3.6) is mostly routine, aside from requiring covariant method permissions for method overrides (T-Method2).

3.2.1 Recovery Rules

Figure 3.7 gives the two promotion rules from Sections 3.1.2 and 3.1.3 that are key to our system's flexibility: the rules for recovering isolated or immutable references, used for both precision and conversion. These rules restrict their input contexts to primitives, externally unique references, and immutable references. The rule T-RecovIso checks the variable in the premise x must either be null, or point into a freshly-allocated or previously present

$$\frac{\mathsf{IsoOrImm}(\Gamma) \quad \mathsf{IsoOrImm}(\Gamma') \quad \Gamma \vdash C \dashv \Gamma', x : \mathsf{writable}\ T}{\Gamma \vdash C \dashv \Gamma', x : \mathsf{isolated}\ T}\ \text{T-RecovIso}$$

$$\frac{\mathsf{IsoOrImm}(\Gamma) \quad \mathsf{IsoOrImm}(\Gamma') \quad \Gamma \vdash C \dashv \Gamma', x : \mathsf{readable}\ T}{\Gamma \vdash C \dashv \Gamma', x : \mathsf{immutable}\ T}\ \text{T-RecovImm}$$

where $\mathsf{IsoOrImm}(\Gamma) \stackrel{\text{def}}{=} \forall (x : p\,T) \in \Gamma.\ \vdash p \prec \mathsf{immutable}$

Figure 3.7: Recovery rules

(in Γ) object aggregate with no other references, and thus it is valid to consider it isolated. Similarly T-RecovImm checks sufficient properties to establish that it is safe to consider it immutable. In practice, using these relies on the frame rule (Figure 3.5).

Without reference immutability, such simple rules for recovery (sometimes called borrowing) would not be possible. In some sense, the information about permissions in the rules' input contexts gives us "permissions for free." We may essentially ignore particular permissions (isolation) for a block of commands, because knowledge of the input context ensures the writable or readable output in each premise is sufficiently separated to convert if necessary (taking advantage of our slight weakening of external uniqueness to admit references to shared immutable objects). Section 3.3.2 elaborates on the details of why we can prove this is sound. Additionally, the permission qualifications specify which references may safely interact with an externally-unique aggregate, and which must be prevented from interacting via the frame rule (readable and writable references). This distinction normally requires precise reasoning about aliases.

3.2.2 Safe Parallelism

Figure 3.8 gives the rules for safe parallelism. They ensure data race freedom, and therefore (for the concurrency primitives we provide) deterministic execution. T-Par corresponds to safe symmetric parallelism, when all writable references are framed out. The second rule T-Async corresponds to the safety criteria for asymmetric parallelism (named for C#'s async block). This rule obviously produces structured parallelism, not the unstructured

$$\frac{\mathsf{NoWrit}(\Gamma_1)\ \ \mathsf{NoWrit}(\Gamma_2)\ \ \Gamma_1 \vdash C_1 \dashv \Gamma_1'\ \ \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1', \Gamma_2'}\ \text{T-Par}$$

$$\frac{\mathsf{IsoOrImm}(\Gamma_1) \quad \Gamma_1 \vdash C_1 \dashv \Gamma_1' \quad \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1', \Gamma_2'}\ \text{T-Async}$$

where $\mathsf{NoWrit}(\Gamma) \stackrel{\text{def}}{=} \forall (x : p\,T) \in \Gamma.\, p \neq \mathsf{writable}$

Figure 3.8: Type rules for safe parallelism. IsoOrImm is defined in Figure 3.7

task-based concurrency present in C#. But it models the state separation required for safe task parallelism: all input to a task must be isolated or immutable. The implementation provides safe task parallelism of this form, as described in Section 3.5.1, as well as structured parallelism.

3.2.3 Methods

The type rule for a method call (T-Call) is shown in Figure 3.5. It is mostly standard (the method exists in the receiver type, actual arguments are subtypes of formal arguments), with a couple of complications. First, isolated actual arguments are forgotten by the typing context, in lieu of extending the method syntax for destructive reads.

Second, methods have required calling permissions, which restrict the side effects a method may have on the receiver. The permission on the receiver at the call site must be at least as permissive as the required permission (e.g., a program cannot call a writable method on a readable receiver). This is standard for reference immutability [252, 275, 276].

Finally, additional restrictions apply when the receiver is isolated. Intuitively, no isolated method may return an alias to an object inside its isolation bubble; alternatively, the restrictions ensure that an inlined method body is suitable for upcasting the isolated receiver's permission, executing, and finally applying `T-RecovIso`. Because no method is type checked with `this` : isolated T (by T-Method* in Figure 3.6, no method may require an isolated receiver permission), no method may leverage recovery rules (Figure 3.7) to recover an isolated or immutable reference to its receiver. Thus, any method returning primitives (`int` and `bool`) or isolated or immutable references is returning a value that does

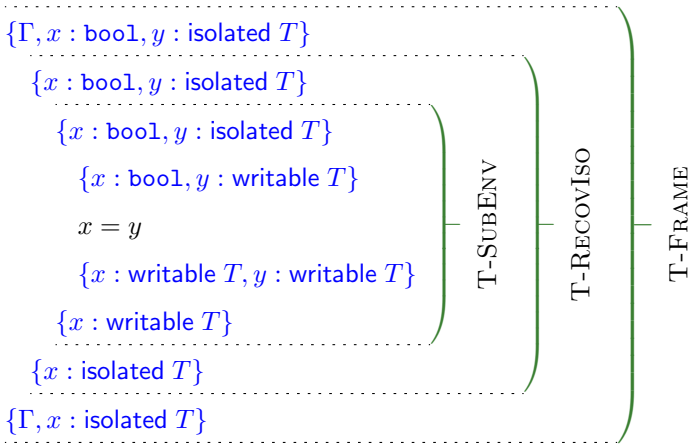
not violate external uniqueness for the receiver's bubble.

3.2.4 Examples

For brevity, because the type environment is flow sensitive, we write typing derivations in the same style as a proof in a program logic, with pre- and post-conditions of a statement in braces before and afterwards. Unmarked adjacent assertions represent use of the rule of implication (subtyping). Uses of other transformation rules are labeled. In Section 3.3, it will become clear how this style directly models the corresponding proof of type soundness in the program logic.

Assigning an Isolated Variable

Assigning an isolated variable consists of framing away outer context, upcasting the isolated reference to writable, assigning normally, weakening to drop the source variable, and an application of T-RecovIso to recover the isolation property on the destination variable. It is possible to add an admissible rule for the direct consumption. It is also possible to preserve access to the source variable by also overwriting it with a primitive value such as null, which is equivalent to an encoding of a traditional destructive read on a variable.



Temporarily Violating Isolation

Figure 3.9 shows the type derivation for a simple use of the T-RecovIso rule, adding a node to an isolated list. The inner portion of the derivation is not notable, simply natural

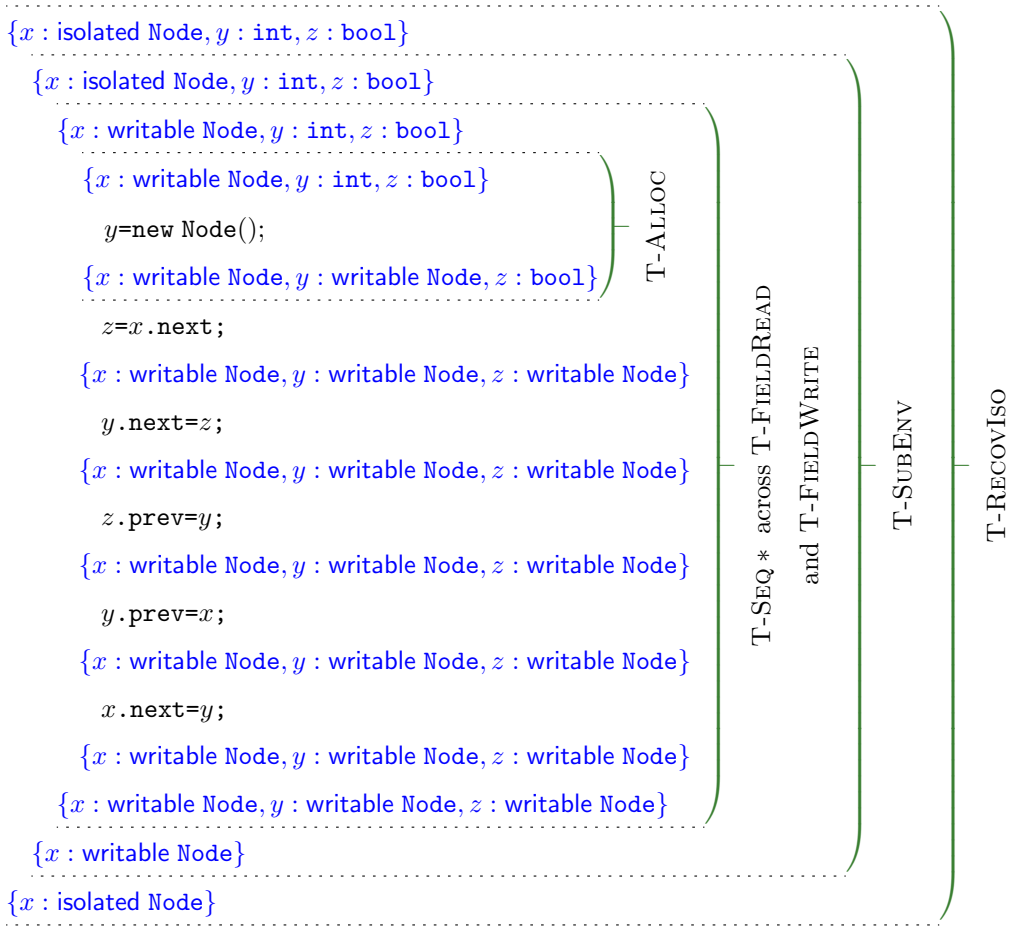


Figure 3.9: Typing derivation for adding a node to an isolated doubly-linked list.

use of sequencing, allocation, and field write rules. But that inner portion is wrapped by a use of subtyping, followed by recovering an isolated reference. Using T-RecovImm to recover an immutable reference would be similar, using a readable reference to the list after the updates.

3.3 Type Soundness

We present our proof of soundness in three stages. First, we present the language without methods. Towards the end of this section, we extend the soundness proof to include method dispatch. Finally, in the next section (Section 3.4), we extend the language with parametric

polymorphism over both base types and permissions, then extend the soundness proof for those additional features.

To prove soundness, we must define the dynamic language semantics and relate the typing rules. The dynamic semantics for commands C are standard small step operational semantics over the states we define below, so we omit them here. The operational rule for reducing an atom a appeals to a denotational semantics of atoms, which is defined in an entirely standard way and therefore also omitted (method calls have some subtlety, but conform to standard intuition of evaluating method calls by inlining method bodies, fully detailed in Section 3.3.4). We relate the type rules to the semantics by defining a denotation of type environments in terms of an extended machine state.

We define abstract machine states as:

$$\mathcal{S} \stackrel{\text{def}}{=} \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{TypeMap}$$

where $\mathsf{Stack} \stackrel{\text{def}}{=} \mathsf{Var} \rightharpoonup \mathsf{Val}$ and is ranged over by s, $\mathsf{Heap} \stackrel{\text{def}}{=} \mathsf{OID} \times \mathsf{Field} \rightharpoonup \mathsf{Val}$ and is ranged over by h, and $\mathsf{TypeMap} \stackrel{\text{def}}{=} \mathsf{OID} \rightharpoonup \mathsf{Class}$ and is ranged over by t.

We only consider well-typed states. To define well-typed states, we assume a function that gives the type and permission for each field of each class, reflects inheritance of fields, and in Section 3.4 handles instantiating field types of polymorphic types:

$$\mathsf{FType} : \mathsf{Class} \times \mathsf{Field} \to \mathsf{Type}$$

We can describe a state (s, h, t) as well-typed iff

$$\begin{aligned}
&\mathsf{WellTyped}(s,h,t) = \forall o, f. \\
&\quad (\exists v.\, h(o,f) = v) \\
&\qquad\qquad \iff (\exists ft.\, \mathsf{FType}(t(o), f) = ft) \\
&\quad \wedge \forall ft.\, \mathsf{FType}(t(o), f) = ft \implies \\
&\qquad \begin{pmatrix} ft = p\ c' \wedge h(o,f) \neq \mathsf{null} \implies \vdash t(h(o,f)) \prec c' \\ \wedge\ ft = \texttt{bool} \implies \exists b.\, h(o,f) = b \\ \wedge\ ft = \texttt{int} \implies \exists n.\, h(o,f) = n \end{pmatrix}
\end{aligned}$$

The first conjunct requires that the type map contains a type for every object in the heap, and vice-versa; it limits the type map to the heap contents. The second conjunct simply enforces that each field holds well-typed contents.

Reasoning about which objects are immutable and the permissions of various references is somewhat difficult for such a basic state space, so we define an *instrumented state* with additional metadata: a partitioning of objects among regions, and permission to each region (important for safe parallelism).

We map each object to a region

$$r : \mathsf{RegionMap} = \mathsf{OID} \rightharpoonup \mathsf{Region}$$

We have three forms of region:

- $\mathsf{Root}(\rho)$ is a root region with abstract root ρ
- $\mathsf{Field}(o, f)$ means the region is only accessible through the isolated field f of object o.
- $\mathsf{Immutable}$ means immutable

We associate two permissions with each root region:

$$\pi : \mathsf{RegionPerm} = \mathsf{Root} \rightharpoonup \mathsf{Update}[0, 1] \times \mathsf{Reference}(0, 1]$$

where

- Update: Is used to indicate if objects in the region can be modified. Full (1) means this is the case. An update permission will be split for the period of a parallel composition, as a fractional permission [41].
- Reference: Is used to indicate whether there is a framed-out reference to this region (< 1). This prevents the conversion of a region to isolated or immutable when there are framed-out readable or writable references to it. Note that 0 reference permission is not allowed; states with no permission at all to a region do not have that permission in their permission map.

For both permissions, the total permission available to the program for any given region is 1. These two permission types capture two interference concepts. You can interfere with

$$
\begin{aligned}
&\mathsf{WellRegioned}(s,h,t,r,\pi) = \\
&\quad \mathsf{CompleteRegionInfo}(s,h,t,r,\pi) \wedge \\
&\quad \left(\begin{array}{l}
\forall o,f,v,p.\, h(o,f) = v \wedge v \in \mathit{OID} \wedge \mathsf{FType}(t(o),f) = p \_ \\
\implies \\
\left(\begin{array}{l}
(r(o) = r(v) \implies \\
\qquad r(o) = \mathsf{Immutable} \vee p \in \{\mathsf{readable}, \mathsf{writable}\}) \\
\wedge\, (r(o) \neq r(v) \implies \mathsf{ValidXRegionRef}(r,o,f,p,v))
\end{array}\right)
\end{array}\right) \\
\wedge \quad & (\forall \rho.\, \mathsf{Root}\ \rho \in \mathit{Img}(r) \implies \pi(\rho)(\geq, >)(0,0)) \\
\wedge \quad & (\forall o,f.\, \mathsf{Field}(o,f) \in \mathit{Img}(r) \implies (o,f) \in \mathit{dom}(h))
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathsf{ValidXRegionRef}(r,o,f,p,v) = \\
&\quad \left(\begin{array}{l}
(r(o) \neq \mathsf{Immutable}) \\
\wedge\, (r(v) = \mathsf{Immutable} \implies p \in \{\mathsf{immutable}, \mathsf{readable}\}) \\
\wedge\, (r(v) = \mathsf{Field}(o,f) \implies p = \mathsf{isolated}) \\
\wedge\, (r(v) = \mathsf{Root}(\_) \implies p = \mathsf{readable} \wedge r(o) = \mathsf{Root}(\_))
\end{array}\right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{CompleteRegionInfo}(s,h,t,r,\pi) = \\
&\quad \left(\begin{array}{l}
\forall o,f.\, h(o,f)\ \text{defined} \implies \\
\qquad t(o)\ \text{defined} \wedge r(o)\ \text{defined} \wedge \\
\qquad\quad (\forall \rho.\, r(o) = \mathsf{Root}(\rho) \implies \pi(\rho)\ \text{defined})
\end{array}\right) \\
&\quad \wedge (\forall o, r(o)\ \text{defined} \implies \exists f, h(o,f)\ \text{defined})
\end{aligned}
$$

Figure 3.10: Definition of Well-Regioned

yourself; and you cannot interfere with other threads. Interference with other threads is prevented by the update permission, only one thread can ever have an update permission to a region.

These states also satisfy two well-formedness predicates. We require instrumented states to be *well-regioned*: e.g. an immutable reference points to an object in region Immutable, no readable or writable reference outside a given externally unique aggregate points to an object in an isolated region, etc. We define well-regioned given in Figure 3.10. The first conjunct ensures full region information for the heap's objects. The second, largest conjunct enforces

restrictions on references between regions. Intra-region pointers must be either within the Immutable region, or following readable or writable fields. Cross-region pointers must not be pointing out of Immutable (which is closed under field dereference), and must either point into Immutable from fields with appropriate permissions, an isolated field pointing into an appropriate Field region, or a readable reference between root regions. The next conjunct requires permissions on any root region, and the final conjunct limits the region map's Fields to those whose entry points are present in the heap.

We can thus define an instrumented state as:

$$\mathcal{M} = \left\{ \begin{array}{l} m \in \mathcal{S} \times \mathsf{RegionMap} \times \mathsf{RegionPerm} \\ \qquad \mid \mathsf{WellRegioned}(m) \wedge \mathsf{WellTyped}(\lfloor m \rfloor) \end{array} \right\}$$

where we define an erasure $\lfloor \cdot \rfloor : \mathcal{M} \to \mathcal{S}$ that projects instrumented states to the common components with $\mathcal{S}$. We use $m.s$, $m.h$, $m.t$, $m.r$, and $m.\pi$ to access the stack, heap, type map, region map and region permission map, respectively.

We view type environments denotationally, in terms of the set of instrumented states permitted by a given environment. Figure 3.11 defines the type environment denotation $[\![\Gamma]\!]_\pi$. Isolated and immutable denotations are mostly straightforward, though they rely on a notion of partial separating conjunction $*$ of instrumented states. To define this separation, we must first define composition of instrumented states $\bullet$:

$$\bullet = (\bullet_{\rightharpoonup}, \bullet_{\cup}, \bullet_{\cup}, \bullet_{\cup}, \bullet_{\pi})$$

where

$$\begin{aligned} s \in (s_1 \bullet_{\rightharpoonup} s_2) &\stackrel{\text{def}}{=} dom(s_1) \cap dom(s_2) = \emptyset \wedge s = s_1 \cup s_2 \\ x \in (x_1 \bullet_{\cup} x_2) &\stackrel{\text{def}}{=} x = x_1 \cup x_2 \\ \pi \in (\pi_1 \bullet_{\pi} \pi_2) &\stackrel{\text{def}}{=} \forall \rho.\, \pi(\rho) = \pi_1(\rho)(+,+)\pi_2(\rho) \end{aligned}$$

Partial separating conjunction then simply requires the existence of two states that compose:

$$m \in P * Q \stackrel{\text{def}}{=} \exists m'.\, \exists m''.\, m' \in P \wedge m'' \in Q \wedge m \in m' \bullet m''$$

This partial separation makes denotation of immutable or isolated references mostly independent of other state. For example, an isolated reference in the environment must be

the only reference to some root region, and it must be possible to split that full permission away from the state described by the rest of the environment without invaliding other parts of the context. We cannot define the meaning of readable and writable individually, because we need an externally visible bound on the regions involved in denoting a readable or writable reference when proving conversions (T-RecovIso and T-RecovImm) sound. We give the meaning of a typing context with respect to some local permission map π, which the denotations of readable and writable references refer to, in addition to checking permissions in the concrete state. Because this π bounds the set of regions supporting an environment, when π contains only full permissions we can prove that certain region-changing operations will not interfere with other threads. It also enables proving parallel composition is race free, as our proof of safe composition gives full update permission on a shared region to neither thread, meaning neither thread may denote a writable reference to a shared object (as in T-Par).

Section 3.3.1 briefly describes specifics of how we interact with an existing program logic [70] to prove soundness. Even without reading Section 3.3.1, the actual soundness proof in Section 3.3.2 should be understandable enough to build an intuition for soundness with only intuition for $*$. The proofs are based around a relation $\sqsubseteq$, which can be viewed as saying what changes to the π and r components of instrumented states are allowed, such that other threads can preserve their view of the typing of the state.

3.3.1 Views Framework

Our soundness proof builds upon an early version of the Views Framework [70], which is in some sense a generalization of the ideas behind separation logic and rely-guarantee reasoning. The definitions we gave in the previous section, $\mathcal{S}$, $\mathcal{M}$ and $\bullet$, happen to coincide with definitions required by this framework. Given a few operations and relations over $\mathcal{M}$, the framework gives a natural structure to the soundness proof as an embedding of type derivations into the view's program logic. To do this we must define:

- An operation $\bullet : \mathcal{M} \to \mathcal{M} \to \mathcal{M}$ that is commutative and associative.
- A preorder interference relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ that defines permissible interference on

$$[\![x : \mathsf{isolated}\ T]\!] = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.\pi(m.r(m.s(x))) = (1,1) \\ \wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \\ \wedge \mathsf{RootClosed}(m.r(m.s(x)), m) \\ \vee m.s(x) = \mathsf{null} \end{array} \right\}$$

$$[\![x : \mathsf{immutable}\ T]\!] = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.r(m.s(x)) = \mathsf{Immutable} \\ \wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \\ \vee m.s(x) = \mathsf{null} \end{array} \right\}$$

$$[\![x : \mathsf{readable}\ T]\!]_\pi = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.s(x) = \mathsf{null} \vee \\ ((\exists \rho.\, \mathsf{Up}(\pi(\rho)) > 0 \wedge \mathsf{Up}(m.\pi(\rho)) > 0 \\ \qquad \wedge\, m.r(m.s(x)) = \mathsf{Root}\ \rho) \\ \vee m.r(m.s(x)) = \mathsf{Immutable}) \\ \wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \end{array} \right\}$$

$$[\![x : \mathsf{writable}\ T]\!]_\pi = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.s(x) = \mathsf{null} \vee \\ (\exists \rho.\, \pi(\rho) = (1, \_) \wedge m.\pi(\rho) = (1, \_) \\ \wedge m.r(m.s(x)) = \mathsf{Root}\ \rho) \\ \wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \end{array} \right\}$$

$$[\![\Gamma, x : \mathsf{isolated}\ T]\!]_\pi = [\![x : \mathsf{isolated}\ T]\!] * [\![\Gamma]\!]_\pi$$

$$[\![\Gamma, x : \mathsf{immutable}\ T]\!]_\pi = [\![x : \mathsf{immutable}\ T]\!] * [\![\Gamma]\!]_\pi$$

$$[\![\Gamma, x : \mathsf{readable}\ T]\!]_\pi = [\![x : \mathsf{readable}\ T]\!]_\pi \cap [\![\Gamma]\!]_\pi$$

$$[\![\Gamma, x : \mathsf{writable}\ T]\!]_\pi = [\![x : \mathsf{writable}\ T]\!]_\pi \cap [\![\Gamma]\!]_\pi$$

$$[\![\epsilon]\!]_\pi = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.\pi \geq \pi \\ \wedge\, (\forall \rho \in dom(\pi).\, \mathsf{Up}(\pi(\rho)) > 0) \\ \wedge\, \forall o, f, o'.\, m.r(o) \in dom(\pi) \wedge m.h(o,f) = o' \\ \qquad \implies \begin{pmatrix} m.r(o') \in dom(\pi) \vee \\ m.r(o') = \mathsf{Immutable} \vee \\ m.r(o') = \mathsf{Field}(o,f) \end{pmatrix} \end{array} \right\}$$

where $\mathsf{RootClosed}(\rho, m) \stackrel{\text{def}}{=} \left(\begin{array}{c} \forall o, f, o'.\, m.r(o) = \mathsf{Root}(\rho) \wedge m.h(o,f) = o' \implies \\ (m.r(o') = m.r(o) \vee m.r(o') = \mathsf{Immutable} \vee \\ m.r(o') = \mathsf{Field}(o,f)) \end{array} \right).$

Figure 3.11: Denoting types and type environments.

an instrumented state. The relation must distribute over composition:

$$\forall m_1, m_2, m.\, (m_1 \bullet m_2)\mathcal{R}m \implies \\ \exists m_1', m_2'.\, m_1\mathcal{R}m_1' \wedge m_2\mathcal{R}m_2' \wedge m \in m_1' \bullet m_2'$$

- A (left and right) unit to $\bullet$ that is closed with respect to to $\mathcal{R}$ (in our case, an instrumented state where all the components are empty maps).

- A denotation of static assertions (in our case, types) in terms of instrumented states: $[\![\Gamma]\!]_\pi$ as in Figure 3.11.

Soundness follows from proving that the denotation of a typing derivation ($[\![\Gamma \vdash C \dashv \Gamma]\!]$) respects some lifting of the operational semantics to instrumented states, by embedding the typing derivations into a program logic. The advantage of choosing this approach over a more traditional technique like syntactic type soundness is that after proving a few lemmas about how type environment denotations behave with respect to composition and interference, a number of typically distinct concepts (including forking and joining threads, frame rules, and safety of environment reordering) become straightforward applications of simpler lemmas.

We define the interference permitted by a single action of another thread (heap modifications) $\mathcal{R}_0$ in Figure 3.3.1. The interference relation for individual actions allows relatively little to change. The stack and region permissions must remain constant. The types and region map must remain constant, aside from objects initially in Field regions disappearing (as in another view performing a destructive read), new objects appearing in any region the current view has no update permission for, and moving objects between root regions with no update permission (intuitively, another view merging two root region contents). The heap has similar restrictions, though it additionally permits field changes in root regions with 0 update permissions. Note that WellTyped and WellRegioned constrain the domains of the region and type maps, and the object-only projection of the heap domain, to all be equal. So if an object appears in the region map, it appears in the type map and heap as well, and so on. We define the final interference relation $\mathcal{R}$ as the reflexive transitive closure of $\mathcal{R}_0$(a direct definition is also possible, but harder to read).

$$
\begin{array}{l}
(s,h,t,r,\pi)\mathcal{R}_0(s',h',t',r',\pi') \stackrel{\text{def}}{=} \\
\quad s = s' \wedge \pi = \pi' \\
\wedge (\forall o, f.\, h(o,f) \neq h'(o,f) \implies \pi(r(o)) = (0,\_)) \\
\wedge \left(\begin{array}{l}
\text{let } O = dom(t) \text{ in} \\
\text{let } O' = dom(t') \text{ in} \\
\quad (\forall o.\, o \in O \cap O' \implies \\
\qquad (r(o) \neq r'(o) \implies \\
\qquad\quad \pi(r(o)) = (0,\_) \wedge \pi(r'(o)) = (0,\_))) \\
\qquad \wedge\, t(o) = t'(o) \\
\quad \wedge (\forall o.\, o \in O \setminus O' \implies r(o) = \mathsf{Field}(\_,\_)) \\
\quad \wedge (\forall o.\, o \in O' \setminus O \wedge r'(o) \in dom(\pi') \\
\qquad \implies \pi'(r'(o)) = (0,\_))
\end{array} \right)
\end{array}
$$

Figure 3.12: The thread interference relation $\mathcal{R}_0$.

Given the specific definitions of composition and interference, the Views Framework defines a number of useful concepts to help structure and simplify the soundness proof. First, it defines a *view* as the subset of instrumented states that are *stable* under interference:

$$\mathsf{View} \stackrel{\text{def}}{=} \{M \in \mathcal{P}(\mathcal{M}) \mid \mathcal{R}(M) \subseteq M\}$$

The program logic is proven sound with respect to views [70], and our denotation of type environments is a valid view (stable with respect to $\mathcal{R}$). The framework also describes a useful concept called the *view shift* operator $\sqsubseteq$, that describes a way to reinterpret a set of instrumented states as a new set of instrumented states with the same erasures to $\mathcal{S}$, accounting for any requirement of other views. It requires that:

$$p \sqsubseteq q \stackrel{def}{\iff} \forall m \in \mathcal{M}.\, \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$$

This specifies how the region information can be changed soundly. That is, we can only change the region information such that all other possible threads can maintain compatible views. This corresponds to precisely what subtyping must satisfy in a concurrent setting and underlies the majority of encoding the complex typing rules into the Views Framework.

3.3.2 Soundness Proof

As mentioned earlier, soundness of a type system in the Views Framework proceeds by embedding the types' denotation into a sound program logic [70]. The logic itself contains judgments of the form $\{p\}C\{q\}$ for views p and q and commands C, and the logic's soundness criteria, subject to our definitions of composition, interference, etc. satisfying the required properties, is

Theorem 1 (Views Logic Soundness [70])**.** *If $\{p\}C\{q\}$ is derived in the logic, then for all $s \in \lfloor p \rfloor$, and $s \in \mathcal{S}$, if $(C, s) \longrightarrow^* (\mathsf{skip}, s')$ then $s' \in \lfloor q \rfloor$.*

Thus because our definitions of $\mathcal{S}$, $\mathcal{M}$, $\bullet$ and $\mathcal{R}$ satisfy the required properties, if every type derivation denotes a valid derivation in the Views Framework's logic, then the type system is sound. We can define the denotation of a typing judgment as:

$$[\![\Gamma_1 \vdash C \dashv \Gamma_2]\!] \stackrel{\text{def}}{=} \forall \pi.\, (\exists \rho.\, \pi(\rho) = (1, \_)) \Rightarrow \{[\![\Gamma_1]\!]_\pi\} C \{[\![\Gamma_2]\!]_\pi\}$$

We map each judgement onto a collection of judgements in the Views Framework. This allows us to encode the rules for recovery, as the logic does not directly support them. Specifically, closing over π allows us to prove that permissions are preserved. Thus, if a block of code is encoded with a set of initially full permissions, it will finish with full permissions, allowing conversion back to isolated if necessary.

We always require there to be at least one local region that is writable: $(\exists \rho.\, \pi(\rho) = (1, \_))$. This is required to prove soundness for the subtyping rule, to allow us to cast isolated to writable.

Here we describe the major lemmas supporting the soundness proof, and omit natural but uninteresting lemmas, such as proving that the denotation of a type environment is stable under interference. We also omit methods here. To prove soundness for method calls, we extended the Views Framework with support for method calls. The semantics are mostly intuitive (reducing a call statement to an inlined method body with freshly bound locals), and both the semantics and proof extensions are described in detail in Section 3.3.4.

The most important lemmas are those for recovering isolated or immutable references, which prove the soundness of the type rules T-RecovIso and T-RecovImm:

Lemma 1 (Recovering Isolation)**.**

$$\mathit{IsoOrImm}(\Gamma) \implies \mathit{FullPermsOnly}(\pi) \implies [\![\Gamma, x : \mathit{writable}]\!]_\pi \sqsubseteq [\![\Gamma, x : \mathit{isolated}]\!]_\emptyset$$

Proof. By induction on Γ. The base case appeals directly to the denotations of the two permissions and leveraging the fact that all permissions in π are $(1, 1)$, so those roots may be remapped into a single root region. The inductive case depends on the fact that denoting isolated and immutable permissions is unaffected by π. □

Lemma 2 (Recovering Immutability)**.**

$$\mathit{IsoOrImm}(\Gamma) \implies \mathit{FullPermsOnly}(\pi) \implies [\![\Gamma, x : \mathit{readable}]\!]_\pi \sqsubseteq [\![\Gamma, x : \mathit{immutable}]\!]_\emptyset$$

Proof. Similar to the proof of Lemma 1, though additionally moving all objects in π's regions and all objects transitively contained in Field regions of those regions' objects into the Immutable region. □

Both Lemmas 1 and 2 rely on the fact that readable and writable references into root regions refer only to regions in π. Without that restriction, and the fact that the denotation of type environments separates isolated and immutable references from regions in π, recovering isolation or immutability would not be possible. Another important factor for these lemmas is our slight weakening of external uniqueness, to allow references out of an aggregate into immutable data; without this, recovering isolation would not be possible with immutable references in Γ.

Applying Lemmas 1 and 2 requires being able to frame away other permissions and non-isolated non-immutable references in order to introduce a fresh singleton π with full permission to a fresh region, making the recovery lemmas applicable:

Lemma 3 (Isolated-Immutable Decomposition)**.**

$$\mathit{IsoOrImm}(\Gamma) \implies [\![\Gamma]\!]_\pi \sqsubseteq [\![\Gamma]\!]_\emptyset * [\![\epsilon]\!]_\pi$$

Proof. By induction on Γ; the goal is a natural result of the way type environment denotation is defined. □

A helpful fact for many lemmas is the fact that environment denotation lifts through separation:

Lemma 4 (Environment Lifting)**.**

$$m \in P * [\![\Gamma]\!]_\pi \implies m \in [\![\Gamma]\!]_\pi$$

Proof. By induction on Γ. □

It is also necessary to prove soundness of environment subtyping, which is one of the most difficult lemmas.

Lemma 5 (Subtyping Denotation)**.**

$$(\exists \rho.\, \pi(\rho) = (1, \_)) \ \wedge \ \vdash \Gamma_1 \prec \Gamma_2 \Longrightarrow [\![\Gamma_1]\!]_\pi \sqsubseteq [\![\Gamma_2]\!]_\pi$$

Proof. This is actually proved by strengthening the lemma to also be polymorphic in views P and Q, and strengthening the conclusion to:

$$Q * ([\![\Gamma_1]\!]_\pi \cap P) \sqsubseteq Q * ([\![\Gamma_2]\!]_\pi \cap P)$$

The proof then proceeds by induction on the subtyping derivation, then inducting on the type of the first entry in the environment in the non-empty environment case. The strengthened induction hypothesis helps manage the fact that type environment denotation alternates between intersections and separations for different type qualifiers. Most of the subtyping implications are obvious (e.g., that isolated implies writable, etc.), though we leverage Lemma 2 to prove that isolated can be converted to immutable.

This strengthened lemma also makes use of supporting lemmas that prove that type environment denotations are *precisely splittable* with respect to isolated or immutable variables, and requires that P is precisely splittable with respect to both. For example, in the immutable case, a predicate P is precisely splittable with respect to an immutable variable x if for all Q and m:

$$m \in ([\![x : \mathit{immutable}\ c]\!] * Q) \to m \in P \to$$
$$\quad m \in ([\![x : \mathit{immutable}\ c]\!] * (Q \cap P))$$

This is obviously true for P being membership in the denotation of any type environment

$$[\![x : \mathsf{readable}\ T]\!]^{\omega}_{\pi} = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.s(x) = \mathsf{null} \vee \\ ((\exists \rho.\, \mathsf{Up}(\pi(\rho)) > 0 \wedge \mathsf{Up}(m.\pi(\rho)) = 0 \\ \quad \wedge m.r(m.s(x)) = \mathsf{Root}\ \rho) \\ \vee m.r(m.s(x)) = \mathsf{Immutable}) \\ \wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \end{array} \right\}$$

$$[\![x : \mathsf{writable}]\!]^{\omega}_{\pi} = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.s(x) = \mathsf{null} \vee \\ (\exists \rho.\, \pi(\rho) = (1, \_) \wedge m.\pi(\rho) = (0, \_) \\ \wedge m.r(m.s(x)) = \mathsf{Root}\ \rho) \\ \wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \end{array} \right\}$$

$$[\![\Gamma, x : \mathsf{isolated}\ T]\!]^{\omega}_{\pi} = [\![x : \mathsf{isolated}\ T]\!] * [\![\Gamma]\!]^{\omega}_{\pi}$$
$$[\![\Gamma, x : \mathsf{immutable}\ T]\!]^{\omega}_{\pi} = [\![x : \mathsf{immutable}\ T]\!] * [\![\Gamma]\!]^{\omega}_{\pi}$$
$$[\![\Gamma, x : \mathsf{readable}\ T]\!]^{\omega}_{\pi} = [\![x : \mathsf{readable}\ T]\!]^{\omega}_{\pi} \cap [\![\Gamma]\!]^{\omega}_{\pi}$$
$$[\![\Gamma, x : \mathsf{writable}\ T]\!]^{\omega}_{\pi} = [\![x : \mathsf{writable}\ T]\!]^{\omega}_{\pi} \cap [\![\Gamma]\!]^{\omega}_{\pi}$$
$$[\![\epsilon]\!]^{\omega}_{\pi} = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} (\forall \rho.\, \rho \in dom(\pi) \\ \qquad \implies \mathsf{Ref}(m.\pi(\rho)) \geq \mathsf{Ref}(\pi(\rho))) \\ \wedge\ dom(m.\pi) = dom(\pi) \end{array} \right\}$$

Figure 3.13: Weak type environment denotation, for framed-out environments. The differences from Figure 3.11 are the permissions required by readable and writable references, and the way π bounds the state's permissions.

that does not include x. Similar proofs apply for precise splitting over an isolated variable, though those proofs must additionally qualify that the type environment does not depend on the isolated region of x or any of its (transitively owned) Field regions. To prove the actual (unstrengthened) goal, we instantiate P with the set of all views and Q with the empty view, then take advantage of the fact that the empty view is the unit with respect to $\bullet$. □

The lemmas for framing (and unframing) parts of the type environment require defining a weakened type denotation $[\![\Gamma]\!]^{\omega}_{\pi}$, shown in Figure 3.13. This denotation is mostly the same as the regular denotation but requires only a non-zero update permission in π, with

0 update permission in the state, but checking reference permission against a π matching the "unframed" state. This makes the environment unusable for executing commands but retaining enough information to restore the environment later. We also use a transformation function on π to frame out a reference permission, preventing the recovery rules from being applied in cases where a readable or writable reference to some region is framed out: $\mathsf{frame\_out\_perm}\ (u, r) := (u, r/2)$.

Lemma 6 (Type Framing)**.**

$$[\![\Gamma, \Gamma']\!]_\pi \sqsubseteq [\![\Gamma]\!]^\omega_{(\mathsf{map\ frame\_out\_perm}\ \pi)} * [\![\Gamma']\!]_{(\mathsf{map\ frame\_out\_perm}\ \pi)}$$

Proof. By induction on Γ and the denotation of Γ. Because for any Γ and π, $[\![\Gamma]\!]_\pi$ and $[\![\Gamma]\!]^\omega_\pi$ have such similar structure, the only slight change is in weakening the denotation of readable and writable references. □

Note that Lemma 6 is actually a significant deviation from the typical approach to proving environment weakening in a type system; rather than inducting over the structure of a derivation to show that typing is preserved by adding bindings, we simply apply the frame rule.

Lemma 7 (Type Unframing)**.**

$$[\![\Gamma]\!]^\omega_{(\mathsf{map\ frame\_out\_perm}\ \pi)} * [\![\Gamma']\!]_{(\mathsf{map\ frame\_out\_perm}\ \pi)} \sqsubseteq [\![\Gamma, \Gamma']\!]_\pi$$

Proof. By induction on Γ, using the facts that the weak denotation only enforces bounds the reference permissions in π and that if all root region pointers remain among regions in the active environment's π, they will still be among the regions in the restored π. □

Fork-join safety:

Lemma 8 (Symmetric Decomposition)**.**

$$\mathit{NoWrit}(\Gamma) \implies [\![\Gamma, \Gamma']\!]_\pi \sqsubseteq [\![\Gamma]\!]_{(\mathsf{map\ halve\_perm}\ \pi)} * [\![\Gamma']\!]_{(\mathsf{map\ halve\_perm}\ \pi)}$$

Proof. By induction on Γ, similar to Lemma 6, using the fact that Γ contains no writable references to sidestep issues splitting full update permission. □

$$\alpha ::= \ldots \mid \mathsf{return}\ z \mid \mathsf{Bind}(\overline{x};\overline{z}) \mid \mathsf{y} = \mathsf{result} \qquad \phi ::= y = x.m(\overline{z}) \qquad \phi \xrightarrow{\mathsf{id}} \mathsf{assume}(\mathsf{pre}\ \phi\ C); C$$

$$\mathsf{pre}(y = x.m(\overline{z}))C = \left\{ s,h,t \;\middle|\; \begin{array}{l} s(x) = \mathsf{null} \vee \\ \left(\begin{array}{l} \mathsf{mbody}(t(s(x)), m) = (\overline{x}, C') \\ \wedge C = \mathsf{Bind}(\overline{w};\overline{z},x); C'[\overline{w}/\overline{x}, \mathtt{this}]; \mathsf{y} = \mathsf{result} \end{array} \right) \end{array} \right\}$$

$$\begin{array}{rcl} [\![\mathsf{Bind}(\overline{w};\overline{z})]\!](s,h,t) & = & \{s[\overline{w} \mapsto s(\overline{z}), h, t \mid \overline{w} \cap dom(s) = \emptyset\} \\ [\![\mathsf{return}\ z]\!](s,h,t) & = & \{s[\mathsf{result} \mapsto z], h, t\} \\ [\![\mathsf{y} = \mathsf{result}]\!](s,h,t) & = & \{s[y \mapsto s(\mathsf{result})], h, t\} \end{array}$$

$$\lceil S \rceil = \{m \mid \lfloor m \rfloor \cap S \neq \emptyset\}$$

$$[\![\mathsf{assume}(S)]\!](s) = \left\{ \begin{array}{ll} \{s\} & \text{If } s \in S \\ \emptyset & \text{Otherwise} \end{array} \right\}$$

$$\frac{}{\Xi, \{p\}\phi\{q\} \vdash \{p\}\phi\{q\}} \qquad \frac{p \cap \lceil S \rceil \sqsubseteq q}{\Xi \vdash \{p\}\mathsf{assume}(S)\{q\}} \qquad \frac{\Xi \Vdash \Xi' \quad \Xi, \Xi' \vdash \{p\}C\{q\}}{\Xi \vdash \{p\}C\{q\}}$$

$$\Xi \Vdash \Xi' \iff \forall \{p\}\phi\{q\} \in \Xi'.\ \forall C.\ \Xi, \Xi' \vdash \{p\}\mathsf{assume}(\mathsf{pre}\ \phi\ C); C\{q\}$$

Figure 3.14: Method call extensions to the language and program logic.

Lemma 9 (Join)**.** $[\![\Gamma]\!]_\pi * [\![\Gamma']\!]_{\pi'} \sqsubseteq [\![\Gamma, \Gamma']\!]_{\pi \bullet \pi'}$

Proof. By induction on Γ, similar to Lemma 7 (this lemma is invoked when π and π' are slightly extended versions of Lemma 8's results). □

Asymmetric parallelism uses environments of a slightly different shape:

Lemma 10 (Asymmetric Decomposition)**.**

$$\mathit{IsoOrImm}(\Gamma) \Longrightarrow [\![\Gamma, \Gamma']\!]_\pi \sqsubseteq [\![\Gamma]\!]_\emptyset * [\![\Gamma']\!]_\pi$$

Proof. By induction on Γ. □

Theorem 2 (Type Soundness)**.**

$$\Gamma_1 \vdash C \dashv \Gamma_2 \Longrightarrow [\![\Gamma_1 \vdash C \dashv \Gamma_2]\!]$$

Proof. By induction on the derivation $\Gamma_1 \vdash C \dashv \Gamma_2$. □

3.3.3 Extending Views for Methods

The core Views program logic does not include methods, so we must extend the language and program logic slightly according to Figure 3.14. First, we add four new atoms, for

returning a variable's contents, for retrieving a returned value, for binding a set of variable values to fresh variables, and for asserting that execution is proceeding from within a certain set of states (binding and assertions may not appear in source programs). We also define a shorthand ϕ for method invocations, and add a new identity transition that reduces method calls to assertions that the method's preconditions hold, followed by the method body. The assertion of method preconditions takes the form of the new atom $\mathsf{assume}(S)$, which has no effect on state but requires the concrete state when executed to be in the set S. The method precondition itself is generated by a metafunction

$$\mathsf{pre} : \phi \to C \to \mathcal{P}(\mathcal{S})$$

It may be more natural to think of a function of type $\phi \to \mathcal{S} \to C$ for looking up the method body for a given function invocation, but we are more concerned with satisfying preconditions by set membership than with finding the method body. pre is defined in Figure 3.14, only for commands that match the correct dynamic dispatch target for the runtime type of the receiver, wrapped with binding of local variables and storing the method's return value in the specified local. The dispatch commands rely on a function mbody

$$\mathsf{mbody} : \mathsf{Class} \times \mathsf{Method} \to \mathsf{Variable\ List} \times C$$

which returns method bodies for a given type, including those inherited from supertypes. Thus the reduction of method call statements only steps to the appropriate method body.

We extend the program logic form to include a context Ξ, a set of triples $\{p\}\phi\{q\}$ that summarize method invocations ϕ as in Figure 3.14:

$$\Xi \vdash \{p\}C\{q\}$$

Most rules of the logic remain the same, merely extended with an unused Ξ context. Three new rules are added as well: a rule for using a method summary in the context, a rule for assuming a state is in a given set of states, and a context strengthening rule for extending a context with sound method summaries, shown in Figure 3.14. The context lookup rule is straightforward. The rule for proving assumptions is relatively straightforward; it uses a lifting of states sets to instrumented state sets ($\lceil S \rceil$), and requires the predecessor state to

be a member of the lifted set. The context extension rule is more subtle. It allows lookup in an arbitrary *valid* extension of the method summary context. Figure 3.14 defines what it means for a context Ξ' to be a valid extension of a context Ξ: $\Xi \Vdash \Xi'$. Intuitively, it requires the extension to be able to prove soundness of the result of the reduction from a ϕ: the target method body wrapped with local binding and value return.

3.3.4 Soundness with Methods

Soundness for methods requires a couple additional lemmas.

First, we must define the context Ξ_P containing verification summaries for all methods in the program, and prove the summaries sound. We define Ξ_P as the context containing the summary schema:

$$\{[\![y : \_, x : p\ T, \overline{z : t}]\!]_\pi\} y = x.m(\overline{z})\{[\![y : t', x : p\ T, \mathsf{RemIso}(\overline{z : t})]\!]_\pi\}$$

for every method m such that:

$$P; \texttt{class}\ T\ [<:\ T2]\ \{\ldots\} \vdash t'\ m(\overline{a : t})p\{\ C; \mathsf{return}\ w\ \}$$

for every calling context that would be permitted by T-Call (allowing subtypes of declared formal arguments, subpermissions of the required receiver permission, subject to the isolated receiver restrictions) for the receiver class at the type of the method declaration. Also add duplicates (all variants) for subtypes T' such that $\vdash T' \prec T$ for any methods present in superclasses that are not overridden in T'. All summaries maintain the declared return type (covariant return types will be handled at call sites). This should include all methods in the program if the program type checks ($\vdash P$). This means that for each method in Ξ_P we can invert on a use of T-Method* to derive important constraints later for typing method bodies and handling method overrides. First we must prove that Ξ_P holds sound method summaries:

Theorem 3 (Method Context)**.** $\emptyset_\Xi \Vdash \Xi_P$

Proof. By simultaneous induction with Theorem 2. By the if-and-only-if definition in Figure

3.14, for each method m in or inherited by a class T we must prove:

$$\begin{array}{rl} & \{[\![y : \_, x : p_{sub}\ T, \overline{z : t_{sub}}]\!]_\pi\} \\ \Xi_P \vdash & \mathsf{assume}(\mathsf{pre}(y = x.m(\overline{z}))\ C);\ C \\ & \{[\![y : t, x : p_{sub}\ T, \mathsf{RemIso}(\overline{z : t_{sub}})]\!]_\pi\} \end{array}$$

For

$$C = \mathsf{Bind}(\overline{w}; \overline{z}, x); C'[\overline{w}/\overline{a}, \mathtt{this}]; \mathsf{y} = \mathsf{result}$$

where

$$\mathsf{mbody}(T, m) = (\overline{x}, C')$$

($\mathsf{mbody}(T, m)$ returns superclass implementations if T inherits an implementation) and by construction of Ξ_P:

$$\overline{\vdash t_{sub} \prec t}$$

We know by inversion on T-Method* that

$$\mathtt{this} : p\ cn, \overline{a : t} \vdash C' \dashv \mathtt{result} : t$$

for formal arguments $\overline{a : t}$ and declaring class cn, which is a (possibly reflexive) supertype of T. Which by α-renaming implies

$$(\mathtt{this} : p\ cn, \overline{a : t})[\overline{w}/\overline{a}, \mathtt{this}] \vdash C'[\overline{w}/\overline{a}, \mathtt{this}] \dashv \mathtt{result} : t$$

And by Theorem 2, with an appropriate π:

$$\{[\![(\mathtt{this} : p\ cn, \overline{a : t})[\overline{w}/\overline{a}, \mathtt{this}]]\!]_\pi\} C'[\overline{w}/\overline{a}, \mathtt{this}] \{[\![\mathtt{result} : t]\!]_\pi\}$$

Which then by the semantics of Bind; weakening to drop the isolated arguments; framing of the non-isolated actual arguments; environment subtyping on the bound arguments to upcast the actual arguments, receiver permission, and upcast the receiver class to the class that defines the dynamically dispatched version of m; the body typing above; weakening to drop locals; and the embedding of simple assignment to recover the result, followed by unframing the original non-isolated arguments, allows us to complete the derivation. When the summary is for an isolated receiver, apply T-RecovIso around the use of argument and

receiver subtyping, recovering isolation after weakening drops all bound locals (this works because by construction of Ξ_P, for a summary with an isolated receiver to be present, all arguments to the method must have been isolated, immutable, or primitive). Section 3.3.5 gives the full tree derivation for the non-isolated receiver case (the isolated case is nearly identical, additionally using T-RecovIso). □

Lemma 11. *Method Call Embedding*

$$\Gamma \vdash x = y.m(\overline{z}) \dashv \Gamma' \Longrightarrow [\![\Gamma \vdash x = y.m(\overline{z}) \dashv \Gamma']\!]$$

Proof. By induction on the argument list and runtime receiver type, which must be some (possibly reflexive) subtype of that in the type environment. Then by T-Method*, if the runtime receiver type is a subclass of the static receiver type that overrides m, then all arguments can be upcast to match the contravariant formal argument types in the override. Then leverage the lookup in Ξ_P (which by construction contains summaries of all methods in all valid calling contexts of the proper direct receiver type), and if necessary apply subtyping again to treat covariant return types and to recover the original call site static class of the receiver. □

The overall soundness for the rest of the system proceeds as before.

3.3.5 Proof of Ξ_P Validity

Section 3.3.4 outlines the proof that $\emptyset_\Xi \Vdash \Xi_P$. Figure 3.15 gives the proof tree for the non-isolated case of Ξ_P validity. The treatment of isolated receivers is similar, additionally using T-RecovIso.

3.4 Polymorphism

Any practical application of this sort of system naturally requires support for polymorphism over type qualifiers. Otherwise code must be duplicated, for example, for each possible permission of a collection and each possible permission for the objects contained within a collection. Of course, polymorphism over unique and non-unique references with mutable state

All with context $\emptyset_\Xi, \Xi_P$:

1	$\{[\![y : \_, x : p_{sub}\ T, \overline{z : t_{sub}}]\!]_\pi\}$		
2	$\mathsf{assume}(\mathsf{pre}(y = x.m(\overline{z}))\ C)$; // for C as binding, body, saving result		
3	$\{[\![y : \_, x : p_{sub}\ T, \overline{z : t_{sub}}]\!]_\pi\}$ (Line 1 and mbody satisfies the requirements for this logic rule.)		
4	$\mathsf{Bind}(\overline{w}; \overline{z}, x)$;		
5	$\{[\![y : \_, x : p_{sub}\ T, \mathsf{RemIso}(\overline{z : t_{sub}}), \overline{w_{arg} : t_{sub}}, w_{\mathtt{this}} : p_{sub}\ T]\!]_\pi\}$ (Manual)		
6	$\{[\![x : p_{sub}\ T, \mathsf{RemIso}(\overline{z : t_{sub}})]\!]_{\pi -}\}$	$*$	$\{[\![y : \_, \overline{w_{arg} : t_{sub}}, w_{\mathtt{this}} : p_{sub}\ T]\!]_{\square\square}\}$
7			$\{[\![y : \_, \overline{w_{arg} : t_{decl}}, w_{\mathtt{this}} : p_{decl}\ T]\!]_{\square\square}\}$ (Subtyping)
8			$\{[\![y : \_]\!]_{\square\_}\}$ $*$ $\{[\![\overline{w_{arg} : t_{decl}}, w_{\mathtt{this}} : p_{decl}\ T]\!]_{\square\square}\}$
9			C'
10			$\{[\![\mathtt{result} : t]\!]_{\square\square}\}$ (T-Method* / def. of Ξ_P)
11			$\{[\![y : \_, \mathtt{result} : t]\!]_{\square\square}\}$
12			$y = \mathtt{result}$;
13			$\{[\![y : t, \mathsf{RemIso}(\mathtt{result} : t)]\!]_{\square\square}\}$ (Assignment or Section 3.2.4)
14			$\{[\![y : t]\!]_{\square\square}\}$
15	$\{[\![x : p_{sub}\ T, \mathsf{RemIso}(\overline{z : t_{sub}}), y : t]\!]_\pi\}$		

Figure 3.15: The proof tree for validity of method summaries when the summary is not for an isolated receiver. Environment reordering steps are not shown explicitly. Parentheticals on the right hand side indicate the method by which the assertion on the same line was proved from the previous assertion and/or statement when it is not an obvious derivation like framing.

W, X, Y, Z	type variables
P, Q, R	permission variables
T, U, V	$::= cn\langle\overline{T}\rangle\langle\overline{p}\rangle \mid X$
TD	$::= \texttt{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ [<: T2]\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\overline{field}\ \overline{meth}\ \}$
$meth$	$::= t\ m\langle\overline{X}\rangle\langle\overline{P}\rangle(t_1\ x_1, ..., t_n\ x_n)\ p\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\ C; \texttt{return}\ x;\ \}$
p, q	$::= \ldots \mid P \mid p \rightsquigarrow p$
Δ	$::= \epsilon \mid \Delta, X <: T \mid \Delta, P <: p$

Figure 3.16: Grammar extensions for the polymorphic system.

still lacks a clean solution due to the presence of destructive reads (using a destructively-reading collection for non-unique elements would significantly alter semantics, though in the pure setting some clever techniques exist [167]).

To that end we also develop a variant of the system with both type and method polymorphism, over class types and permissions. As in C#, we allow a sort of dependent kinding for type parameters, allowing type and permission parameters to bound each other (without creating a cycle of bounding). We separate permission parameters from class parameters for simplicity and expressivity. A source-level variant may wish to take a single sort of parameter that quantifies over a permission-qualified type as in IGJ [275]. There is a straightforward embedding from those constraints to the more primitive constraints we use, and our separation makes our language suitable as an intermediate language that can be targeted by variants of a source language that may change over time.

Figure 3.16 gives the grammar extensions for the polymorphic system. Our language permits bounding a polymorphic permission by any other permission, including other permission parameters, and by concrete permissions that produce parameters with only a single valid instantiation (such as immutable). This allows for future extensions, for example distinguishing multiple immutable data sources. We add a context Δ containing type bounds to the typing and subtyping judgements. We extend the previous typing and subtyping rules in the obvious way. Δ is invariant for the duration of checking a piece of code, and the main soundness theorem, restated below, relies on an executing program having an empty Δ (a program instantiates all type and permission parameters to concrete classes

$$\frac{\mathsf{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ \mathsf{where}\ \ldots\{\overline{field}\ t\ f\ \overline{field}\ \overline{method}\} \in P}{t[\overline{P/p}, \overline{X/T}]\ f \in cn\langle\overline{T}\rangle\langle\overline{p}\rangle}$$

$$\frac{\mathsf{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ \mathsf{where}\ \ldots\ \{\overline{field}\ \overline{method}\} \in P \quad m \in \overline{method} \quad m = t'\ m\langle\overline{Y}\rangle\langle\overline{Q}\rangle(\overline{u'\ z'})\ p'\ \texttt{where}\ \overline{Y <: V}, \overline{Q <: q} \ldots}{m[\overline{P/p}, \overline{X/T}] \in cn\langle\overline{T}\rangle\langle\overline{p}\rangle}$$

$$\mathsf{IsoOrImm}_\Delta(\Gamma) \stackrel{\text{def}}{=} \forall(x : p\,T) \in \Gamma.\ \Delta \vdash p \prec \mathsf{immutable}$$

$$\begin{aligned}
\rhd_\Delta : \mathsf{Permission} \to \mathsf{Permission} \to \mathsf{Permission}& \\
\mathsf{immutable} \rhd_\Delta _ &= \mathsf{immutable} \\
_ \rhd_\Delta \mathsf{immutable} &= \mathsf{immutable} \\
\mathsf{readable} \rhd_\Delta \mathsf{writable} &= \mathsf{readable} \\
\mathsf{readable} \rhd_\Delta \mathsf{readable} &= \mathsf{readable} \\
\mathsf{writable} \rhd_\Delta \mathsf{readable} &= \mathsf{readable} \\
\mathsf{writable} \rhd_\Delta \mathsf{writable} &= \mathsf{writable} \\
\mathsf{readable} \rhd_\Delta Q &= \mathsf{readable} \\
P \rhd_\Delta \mathsf{readable} &= \mathsf{readable} \\
\mathsf{writable} \rhd_\Delta Q &= Q \\
P \rhd_\Delta \mathsf{writable} &= P \\
P \rhd_\Delta Q &= \begin{cases} Q & \Delta \vdash P \prec Q \\ P & \Delta \vdash Q \prec P \\ P \rightsquigarrow Q & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 3.17: Auxiliary judgements and metafunctions for the polymorphic system.

and permissions). Concretely, type judgements and subtyping judgements now take the forms:

$$\Delta \mid \Gamma \vdash C \dashv \Gamma \qquad \Delta \vdash t \prec t$$

Figure 3.17 gives auxiliary judgements and metafunctions used in the polymorphic system. Most of the extensions are straightforward, leveraging bounds in Δ to type check interactions with generic permissions and class types. We discuss a couple of the most interesting rules here, presenting the full type system in Section 3.4.1.

One of the most interesting rules is the field read:

$$\frac{t'\ f \in T \qquad p \neq \mathsf{isolated} \vee t' = \mathsf{immutable}\ \_ \qquad t' \neq \mathsf{isolated}\ \_ \vee p = \mathsf{immutable}}{\Delta \mid x : \_, y : p\ T \vdash x = y.f \dashv y : p\ T, x : p \rhd_\Delta t'}$$

It uses a variant of permission combining parameterized by Δ, given in Figure 3.17, and lifted to types as before. When reading the definition of $\rhd_\Delta$, bear in mind that no permission variable may ever be instantiated to isolated. The final case of $\rhd_\Delta$ produces a *deferred permission combination* ($p \rightsquigarrow p$). The two cases previous to it that combine uninstantiated permission parameters leverage the bounding relation in Δ to give a sound answer that might produce writable or immutable results that can be used locally (though in the case that P is instantiated to immutable, this can lose some precision compared to instantiated uses). In the unrelated case, there is always an answer to give: readable. But this is too imprecise for uses such as container classes. There is always a more precise answer to give, but it cannot be known until all parameters are instantiated. To this end, we also change the type and permission substitution to reduce $p \rightsquigarrow q$ to $p \rhd_\Delta q$ if p and q are both concrete permissions after substitution. Note that these deferred permissions are effectively equivalent to readable in terms of what actions generic code using them may perform. This deferred combination plays a pivotal role in supporting highly polymorphic collection classes, as Section 3.5.3 describes.

We also support covariant subtyping on readable and immutable references, as in IGJ [275].

$$\frac{\Delta \vdash p\ c\langle \overline{T}^{i-1}, T_i, \overline{T}^{m-i}\rangle\langle \overline{p}\rangle \quad \Delta \vdash p\ c\langle \overline{T}^{i-1}, T_i', \overline{T}^{m-i}\rangle\langle \overline{p}\rangle \quad p = \mathsf{readable} \vee p = \mathsf{immutable} \quad \Delta \vdash T_i \prec T_i'}{\Delta \vdash p\ c\langle \overline{T}^{i-1}, T_i, \overline{T}^{m-i}\rangle\langle \overline{p}\rangle \prec p\ c\langle \overline{T}^{i-1}, T_i', \overline{T}^{m-i}\rangle\langle \overline{p}\rangle}$$

There is another rule for safe covariant permission subtyping as well.

3.4.1 Soundness for Generics

At a high level, the soundness proof for the polymorphic system is similar to the monomorphic system, because we only need to embed fully-instantiated programs (the top level program expression is type checked with an empty type bound context). The definition for type maps in the concrete machine states and views are redefined to have a range of only fully-instantiated types, making type environment denotations defined only over fully-instantiated types.

Several auxiliary lemmas are required such as that substituting any valid permission or type instantiations into a generic derivation yields a consistent derivation. Additionally, the denotation of writable and isolated references must use a strengthened subtyping bound on their referents, to ensure they are viewed at a type that does not change any field types (thus preventing the classic reference subtyping problem while allowing covariant subtyping of read-only references).

Figures 3.18 and 3.19 present the full polymorphic type system, aside from the natural extensions of rules from Figures 3.5, 3.7, and 3.8 for judgement forms using Δ. It uses supporting judgements and metafunctions already shown in Figure 3.17.

First, we must slightly refine the denotation of type judgements:

$$[\![\epsilon \mid \Gamma_1 \vdash C \dashv \Gamma_2]\!] = \left(\begin{array}{l} \forall \pi.\, (\exists \rho.\, \pi(\rho) = (1, \_)) \Rightarrow \\ \qquad \Xi_\emptyset \vdash \{[\![\Gamma_1]\!]_\pi\} C \{[\![\Gamma_2]\!]_\pi\} \end{array} \right)$$

We define the denotation only over empty type bounds because the only judgements that need to embed into the program logic are those for running programs, for which all type parameters must be instantiated.

We must prove that any instantiation of permission parameters is consistent with a monomorphic version of the code:

Lemma 12. *Consistent Adaptation*

$$\begin{array}{l} \Delta = \Delta_P, P <: P', \Delta'_P \implies \\ \quad \Delta = \Delta_Q, Q <: Q', \Delta'_Q \implies \\ \qquad (\Delta_P[Q/q]) \setminus (q <: Q') \vdash p \prec P' \implies \\ \qquad\quad (\Delta_Q[P/p]) \setminus (p <: P') \vdash q \prec Q' \implies \\ \qquad\qquad P \rhd_\Delta Q = r \implies \\ (\Delta[P/p, Q/q] \setminus (p <: P', q <: Q')) \vdash (p \rhd_\Delta q) \prec r[P/p, Q/q] \end{array}$$

For any valid instantiation of P and Q under Δ, the instantiation (and reduction) of the result of combining P and Q is always a supertype (possibly reflexively) of combining the concrete instantiations.

Proof. By induction on $P \rhd_\Delta Q = r$ followed by induction on valid instantiations of P and Q according to case. In general, instantiating the generic combination yields the same

$$\boxed{\Delta \vdash p \prec p'} \quad \frac{}{\Delta, P <: p \vdash P \prec p}\ \text{S-PVar} \quad \frac{}{\Delta \vdash p \prec p}\ \text{S-PRefl} \quad \frac{}{\Delta \vdash p \prec \mathsf{readable}} \quad \frac{}{\Delta \vdash \mathsf{isolated} \prec p}$$

$$\boxed{\Delta \vdash T \prec T'} \quad \frac{}{\Delta, X <: T \vdash X \prec T}\ \text{S-CVar} \quad \frac{}{\Delta \vdash T \prec T}\ \text{S-CRefl}$$

$$\frac{P \vdash \texttt{class}\ c\langle \overline{X} \rangle \langle \overline{P} \rangle <: d\langle \overline{U} \rangle \langle \overline{q} \rangle \ldots \quad \Delta \vdash c\langle \overline{T} \rangle \langle \overline{p} \rangle \quad \Delta \vdash d\langle \overline{U} \rangle \langle \overline{q} \rangle [\overline{X}/\overline{T}][\overline{P}/\overline{p}]}{\Delta \vdash c\langle \overline{T} \rangle \langle \overline{p} \rangle \prec d\langle \overline{U} \rangle \langle \overline{q} \rangle [\overline{X}/\overline{T}][\overline{P}/\overline{p}]}\ \text{S-CDecl}$$

$$\boxed{\Delta \vdash t \prec t'} \quad \frac{\Delta \vdash p \prec p'}{\Delta \vdash p\,T \prec p'\,T}\ \text{S-Perm} \quad \frac{\Delta \vdash T \prec T'}{\Delta \vdash p\,T \prec p\,T'}\ \text{S-Class} \quad \frac{}{\Delta \vdash t \prec t}\ \text{S-Refl} \quad \frac{\Delta \vdash t \prec t' \quad \Delta \vdash t' \prec t''}{\Delta \vdash t \prec t''}\ \text{S-Trans}$$

$$\frac{\Delta \vdash p\,c\langle \overline{T}^{i-1}, T_i, \overline{T}^{m-i} \rangle \langle \overline{p} \rangle \quad \Delta \vdash p\,c\langle \overline{T}^{i-1}, T_i', \overline{T}^{m-i} \rangle \langle \overline{p} \rangle \quad p = \mathsf{readable} \vee p = \mathsf{immutable} \quad \Delta \vdash T_i \prec T_i'}{\Delta \vdash p\,c\langle \overline{T}^{i-1}, T_i, \overline{T}^{m-i} \rangle \langle \overline{p} \rangle \prec p\,c\langle \overline{T}^{i-1}, T_i', \overline{T}^{m-i} \rangle \langle \overline{p} \rangle}\ \text{S-CParam}$$

$$\frac{\Delta \vdash p\,c\langle \overline{T} \rangle \langle \overline{p}^{i-1}, p_i, \overline{p}^{m-i} \rangle \quad \Delta \vdash p\,c\langle \overline{T} \rangle \langle \overline{p}^{i-1}, p_i', \overline{p}^{m-i} \rangle \quad p = \mathsf{readable} \vee p = \mathsf{immutable} \quad \Delta \vdash p_i \prec p_i'}{\Delta \vdash p\,c\langle \overline{T} \rangle \langle \overline{p}^{i-1}, p_i, \overline{p}^{m-i} \rangle \prec p\,c\langle \overline{T} \rangle \langle \overline{p}, p_i, \overline{p}^{m-i} \rangle}\ \text{S-PParam}$$

$$\boxed{\Delta \mid \Gamma_1 \vdash C \dashv \Gamma_2} \quad \frac{t \neq \mathsf{isolated}\ \_}{\Delta \mid x : \_, y : t \vdash x = y \dashv y : t, x : t}\ \text{T-AssignVar} \quad \frac{\Delta \vdash T}{\Delta \mid x : \_ \vdash x = \texttt{new}\ T() \dashv x : \mathsf{isolated}\ T}\ \text{T-New}$$

$$\frac{t'\ f \in T \qquad p \neq \mathsf{isolated} \vee t' = \mathsf{immutable}\ \_ \qquad t' \neq \mathsf{isolated}\ \_ \vee p = \mathsf{immutable}}{\Delta \mid x : \_, y : p\,T \vdash x = y.f \dashv y : p\,T, x : p \rhd_\Delta t'}\ \text{T-FieldRead}$$

$$\frac{\Delta \vdash p \prec \mathsf{writable} \quad p \neq \mathsf{isolated} \quad t\ f \in T}{\Delta \mid y : p\,T, x : t \vdash y.f = x \dashv y : p\,T, \mathsf{RemIso}(x : t)}\ \text{T-FieldWrite}$$

$$\frac{\mathsf{isolated}\ T_f\ f \in T}{y : \mathsf{writable}\ T \vdash x = \texttt{consume}(y.f) \dashv y : \mathsf{writable}\ T, x : \mathsf{isolated}\ T_f}\ \text{T-FieldConsume}$$

$$\frac{\begin{array}{c} t'\ m\langle \overline{Y} \rangle \langle \overline{Q} \rangle (\overline{u'\ z'})\ p'\ \texttt{where}\ \overline{Y <: V}, \overline{Q <: q} \ldots \in T \\ \forall i \in 1..|\overline{r}|.\, \Delta \vdash r_i \prec q_i[\overline{Q/r}^{i-1}] \quad \forall i \in 1..|\overline{U}|.\, \Delta \vdash U_i \prec V_i[\overline{Q/r}, \overline{Y/U}^{i-1}] \\ \Delta \vdash p \prec p' \qquad \overline{\Delta \vdash u \prec u'[\overline{Q/r}, \overline{Y/U}]} \qquad |\overline{r}| = |\overline{Q}| \qquad |\overline{U}| = |\overline{Y}| \qquad \mathsf{isolated} \notin \overline{r} \\ p = \mathsf{isolated} \implies \mathsf{ConcretePermission}(p') \wedge t \neq \mathsf{readable}\ \_ \wedge t \neq \mathsf{writable}\ \_ \wedge \mathsf{IsoOrImm}(\overline{z : t}) \wedge p' \neq \mathsf{immutable} \end{array}}{\Delta \mid y : p\,T, \overline{z : u} \vdash x = y.m\langle \overline{U} \rangle \langle \overline{r} \rangle (\overline{z}) \dashv y : p\,T, \mathsf{RemIso}(\overline{z : t}), x : t'}\ \text{T-Call}$$

$$\boxed{\Delta \vdash T} \quad \frac{\begin{array}{c} P \vdash \texttt{class}\ c\langle \overline{X}^m \rangle \langle \overline{P}^n \rangle : d\langle \ldots \rangle \langle \ldots \rangle\ \texttt{where}\ \overline{X <: T}^m, \overline{P <: p}^n \ldots \quad \forall i \in 1..n.\Delta \vdash q_i \quad \forall i \in 1..m.\Delta \vdash U_i \\ \forall i \in 1..n.q_i \neq \mathsf{isolated} \quad \forall i \in 1..m.\Delta \vdash U_i \prec T_i[\overline{p/q}, \overline{X/U}^{i-1}] \quad \forall i \in 1..n.\Delta \vdash q_i \prec p_i[\overline{P/q}^{i-1}] \end{array}}{\Delta \vdash c\langle \overline{U} \rangle \langle \overline{q} \rangle}\ \text{WF-T}$$

$$\boxed{\Delta \vdash p} \quad \frac{p \in \{\mathsf{isolated}, \mathsf{writable}, \mathsf{immutable}, \mathsf{readable}\}}{\Delta \vdash p}\ \text{WF-BasicPerm} \quad \frac{P <: p \in \Delta}{\Delta \vdash P}\ \text{WF-BoundPerm}$$

$$\boxed{\Delta \vdash t} \quad \frac{\Delta \vdash p \quad \Delta \vdash T}{\Delta \vdash p\,T}\ \text{WF-QualTy} \quad \frac{t \in \{\texttt{int}, \texttt{bool}\}}{\Delta \vdash t}\ \text{WF-PrimTy} \quad \boxed{\Delta \vdash \Gamma} \quad \frac{}{\Delta \vdash \epsilon}\ \text{WF-Empty}$$

$$\frac{\Delta \vdash \Gamma \quad \Delta \vdash t \quad x \notin \Gamma}{\Delta \vdash \Gamma, x : t}\ \text{WF-Type} \quad \boxed{\vdash \Delta} \quad \frac{\vdash \Delta \quad \Delta \vdash p \quad P \notin \Delta}{\vdash \Delta, P <: p}\ \text{WF-PBound} \quad \frac{\vdash \Delta \quad \Delta \vdash T \quad X \notin \Delta}{\vdash \Delta, X <: T}\ \text{WF-CBound}$$

Figure 3.18: Generics typing rules. All structural, recovery, and parallelism rules remain as in Figures 3.5, 3.7, and 3.8, extended with Δ in conclusions and appropriate hypotheses.

$$\boxed{\vdash P} \quad \frac{\forall c \in \mathsf{Classes}(P).\ P \vdash c \quad \epsilon \vdash \Gamma \quad \epsilon \mid \Gamma \vdash \mathsf{Expression}(P) \dashv \Gamma' \quad \mathsf{ClassesOnce}(P)}{\vdash P}\ \text{T-Program}$$

$$\boxed{P \vdash TD}$$

$$\frac{\begin{array}{c} TD = \texttt{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ [<: T2]\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\overline{fld}\ \overline{meth}\ \} \\ |\overline{X}| = |\overline{X <: T}| \quad |\overline{P}| = |\overline{P <: p}| \\ \Delta = \epsilon, \overline{P <: p}, \overline{X <: T} \quad \vdash \Delta \quad \Delta \vdash T2 \quad \mathsf{FldsOnce}(\overline{field}) \\ \mathsf{MethsOnce}(\overline{meth}) \quad \overline{P; TD \vdash field} \quad \overline{P; TD \vdash meth} \end{array}}{P \vdash TD}\ \text{T-Class}$$

$$\boxed{P; TD \vdash field} \quad \frac{\begin{array}{c} TD = \texttt{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ [<: T2]\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\overline{field}\ \overline{meth}\ \} \\ t\ f \in \overline{field} \quad f \notin \mathsf{Fields}(\mathsf{ParentClasses}(T2)) \quad \epsilon, \overline{P <: p}, \overline{X <: T} \vdash t \end{array}}{P; TD \vdash p\ T\ f}\ \text{T-Field}$$

$$\boxed{P; TD \vdash meth}$$

$$\frac{\begin{array}{c} TD = \texttt{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ [<: T2]\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\overline{field}\ \overline{meth}\ \} \\ meth = t\ m\langle\overline{Y}\rangle\langle\overline{Q}\rangle(t_1\ x_1, ..., t_n\ x_n)\ p\ \texttt{where}\ \overline{Y <: U}, \overline{Q <: q}\ \{\ C; \mathsf{return}\ x;\ \} \\ meth \in \overline{meth} \quad p \neq \mathsf{isolated} \quad |\overline{Y}| = |\overline{Y <: U}| \quad |\overline{Q}| = |\overline{Q <: q}| \\ \Delta = \epsilon, \overline{P <: p}, \overline{X <: T}, [p <: \mathsf{readable}], \overline{Q <: q}, \overline{Y <: U} \\ \vdash \Delta \quad \forall i \in 1 \ldots n.\ \Delta \vdash t_i \quad \Delta \vdash t \\ \Delta \mid \texttt{this} : p\ cn, x_1 : t_1, \ldots, x_n : t_n \vdash C; \mathsf{return}\ x \dashv \texttt{result} : t \\ \forall t', \bar{t}, \bar{x}, p'.\ t'\ m(t'_1\ x'_1, ..., t'_n\ x'_n)\ p' \notin T2 \end{array}}{P; TD \vdash meth}\ \text{T-Method1}$$

$$\frac{\begin{array}{c} TD = \texttt{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ [<: T2]\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\overline{field}\ \overline{meth}\ \} \\ meth = t\ m\langle\overline{Y}\rangle\langle\overline{Q}\rangle(t_1\ x_1, ..., t_n\ x_n)\ p\ \texttt{where}\ \overline{Y <: U}, \overline{Q <: q}\ \{\ C; \mathsf{return}\ x;\ \} \\ meth \in \overline{meth} \quad p \neq \mathsf{isolated} \quad |\overline{Y}| = |\overline{Y <: U}| \quad |\overline{Q}| = |\overline{Q <: q}| \\ \Delta = \epsilon, \overline{P <: p}, \overline{X <: T}, [p <: \mathsf{readable}], \overline{Q <: q}, \overline{Y <: U} \\ \vdash \Delta \quad \forall i \in 1 \ldots n.\ \Delta \vdash t_i \quad \Delta \vdash t \\ \Delta \mid \texttt{this} : p\ cn, x_1 : t_1, \ldots, x_n : t_n \vdash C; \mathsf{return}\ x \dashv \texttt{result} : t \\ t'\ m(t'_1\ x'_1, ..., t'_n\ x'_n)\ p'\ \mathsf{where}\ \overline{W <: V}, \overline{R <: q'} \in T2 \\ \Delta \vdash t \prec t' \quad \Delta \vdash p' \prec p \vee \exists R.\ p = R \quad \forall i \in [1...n].\Delta \vdash t'_i \prec t_i \\ \overline{Y <: U} \subseteq \overline{W <: V} \quad \overline{Q <: q} \subseteq \overline{R <: q'} \end{array}}{P; TD \vdash meth}\ \text{T-Method2}$$

Figure 3.19: Well formed polymorphic programs. In T-Method*, the bound on the method permission p is present if and only if p is a permission variable.

permission as combining the instantiations directly, the only exception being $P \rhd_\Delta Q$ when $\Delta \vdash P \prec Q$ and P is instantiated as immutable and Q is instantiated as readable. In that case, $P \rhd_\Delta Q = Q$, which instantiates to readable, while immutable$\rhd_\Delta$readable = immutable, which is a subtype of the instantiated generic result (readable). Note that the complete substitution of $(P \rightsquigarrow Q)[P/p, Q/q]$ is defined as $p \rhd_\Delta q$. □

We must also prove that the typing derivations introduce no ill-formed contexts (unbound parameters):

Lemma 13. *Well-Formed Environments*

$$\vdash \Delta \implies \Delta \vdash \Gamma \implies \Delta \mid \Gamma \vdash C \dashv \Gamma' \implies \Delta \vdash \Gamma'$$

Proof. By induction on the assumed typing derivation. □

And the most important polymorphism-specific lemma is the proof that the typing derivation agrees with any instantiation of parameters.

Lemma 14. *Generics Instantiation*

$$\begin{aligned}&\Delta, X <: T \mid \Gamma \vdash C \dashv \Gamma' \implies \\ &\quad \Delta \vdash T' <: T \implies \\ &\qquad \Delta \mid \Gamma[X/T'] \vdash C[X/T'] \dashv \Gamma'[X/T']\end{aligned}$$

and

$$\begin{aligned}&\Delta, P <: p \mid \Gamma \vdash C \dashv \Gamma' \implies \\ &\quad \Delta \vdash p' <: p \implies \\ &\qquad \Delta \mid \Gamma[P/p'] \vdash C[P/p'] \dashv \Gamma'[P/p']\end{aligned}$$

Proof. By simultaneous induction on the assumed typing derivations. Since all the type and permission parameters have bounds and the instantiations must satisfy those bounds, we can easily satisfy the subtyping rule. Most other rules are straightforward, though to maintain expressiveness we tweak the field write rule in Figure 3.18 to simply require that the permission is bounded by writable and not isolated. □

$$[\![x : \mathsf{isolated}\ T]\!] = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.\pi(m.r(m.s(x))) = (1,1) \\ \wedge m.t(m.s(x)) = T' \wedge \epsilon \vdash T' \prec_{=f} T \\ \wedge \mathsf{RootClosed}(m.r(m.s(x)), m) \\ \vee m.s(x) = \mathsf{null} \end{array} \right\}$$

$$[\![x : \mathsf{writable}\ T]\!]_\pi = \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l} m.s(x) = \mathsf{null} \vee \\ (\exists \rho.\, \pi(\rho) = (1, \_) \wedge m.\pi(\rho) = (1, \_) \\ \wedge m.r(m.s(x)) = \mathsf{Root}\ \rho) \\ \wedge m.t(m.s(x)) = T' \wedge \epsilon \vdash T' \prec_{=f} T \end{array} \right\}$$

where

$$\frac{\Delta \vdash T \prec T' \quad \forall t, f.\, t\ f \in T' \implies t\ f \in T}{\Delta \vdash T \prec_{=f} T'}$$

Figure 3.20: Denoting types and type environments in the polymorphic system, differences from Figure 3.11. The main change is to the subclass clause of the writable and isolated cases, which require the runtime type's field types to exactly match the supertype's.

Intuitively, repeated application of Lemma 14 allows proving that any permission or class instantiation of a method body is consistent with some monomorphic version. Now we will describe other notable new cases in the soundness proof.

First, subtyping in the parametric system is much richer, particularly because of the rules S-CParam and S-PParam, which allow covariant subtyping of type parameters when the polymorphic object reference does not allow mutation (as in IGJ [275]).[3] Consequently, the type system must avoid problems with reference subtyping. We do this by slightly modifying the denotation of type environments, as in Figure 3.20. In addition to updating the subtype relation form to the polymorphic version ($\Delta \vdash T \prec T$), the isolated and writable cases use a strengthened subtyping relationship that constrains fields present in both superclass and subclass to have the same type given either class type. Thus, attempts to prove the parameter subtyping rules sound when applied to writable or isolated methods fail, and they continue to provide enough information to prove that any permitted field writes preserve well-typedness of the heap. Thus, we use two additional lemmas when reproving the subtyping denotation lemma (Lemma 5) for the polymorphic system:

[3]The monomorphic system requires no special consideration of mutable fields because no subtyping in that system can change the type of a field lookup.

Lemma 15 (Class Parameter Subtyping)**.**

$$
\begin{array}{l}
(\exists \rho.\, \pi(\rho) = (1, \_)) \implies \\
\quad \epsilon \vdash p\ c\langle \ldots, T, \ldots \rangle \langle \ldots \rangle \prec p\ c\langle \ldots, T', \ldots \rangle \langle \ldots \rangle \implies \\
[\![x : p\ c\langle \ldots, T, \ldots \rangle \langle \ldots \rangle, \Gamma]\!]_\pi \sqsubseteq [\![x : p\ c\langle \ldots, T', \ldots \rangle \langle \ldots \rangle, \Gamma]\!]_\pi
\end{array}
$$

Proof. By inversion on the subtyping assumption (S-CPARAM is the only possibility) there are only two possible base permissions. In each case, the result is trivial if x refers to null. Otherwise, the region results and subtyping goals are straightforward. □

For Lemma 15, it may be informative to consider why the lemma would fail if the base permission were not restricted to readable or immutable, which is due to the use of $\Delta \vdash T \prec_{=f} T'$ in Figure 3.20. We prove a similar lemma for S-PPARAM:

Lemma 16 (Permission Parameter Subtyping)**.**

$$
\begin{array}{l}
(\exists \rho.\, \pi(\rho) = (1, \_)) \implies \\
\quad \epsilon \vdash p\ c\langle \ldots \rangle \langle \ldots, q, \ldots \rangle \prec p\ c\langle \ldots \rangle \langle \ldots, q', \ldots \rangle \implies \\
[\![x : p\ c\langle \ldots \rangle \langle \ldots, q, \ldots \rangle, \Gamma]\!]_\pi \sqsubseteq [\![x : p\ c\langle \ldots \rangle \langle \ldots, q', \ldots \rangle, \Gamma]\!]_\pi
\end{array}
$$

Proof. Similar to proof of Lemma 15. □

Method soundness is similar to the monomorphic system; the only change of note is to ensure the method summary context for the program logic contains summaries for all concrete instantiations of polymorphic method types. Details appear in Section 3.4.2.

Other lemmas from Section 3.3.2 must be reproven with empty type bound contexts. Beyond the changes above, the lemmas are not significantly different. Finally, we must prove the (restated) soundness theorem:

Theorem 4 (Polymorphic Type Soundness)**.**

$$\epsilon \mid \Gamma_1 \vdash C \dashv \Gamma_2 \implies [\![\epsilon \mid \Gamma_1 \vdash C \dashv \Gamma_2]\!]$$

which when applied to the main program expression, guarantees the execution of that program will not violate type safety.

3.4.2 Polymorphic Methods

Polymorphic method support also required a few minor adjustments, primarily ensuring that the method summary context contains summaries for all valid instantiations of polymorphic method types.

Definition 1 (Polymorphic Program Method Context)**.** $\langle \Xi_P \rangle$ *is the summary context containing:*

$$\{[\![\epsilon \mid y : \_, x : p\ T, \overline{z : t}]\!]_\pi\}$$
$$y = x.m\langle \overline{T} \rangle \langle \overline{q} \rangle (\overline{z})$$
$$\{[\![\epsilon \mid y : t', x : p\ T, \mathsf{RemIso}(\overline{z : t})]\!]_\pi\}$$

for every method in the program in every concrete calling context that would be permitted by the polymorphic version of T-Call *in Figure 3.18 at each receiver type that declares or inherits a particular override of each method, as for* Ξ_P*.*

Note that Definition 1 includes all concrete instantiations of applicable method and class types.

Theorem 5 (Polymorphic Method Context)**.** $\emptyset_\Xi \Vdash \langle \Xi_P \rangle$

Proof. Similar to Theorem 3, but leveraging Lemma 14 when reasoning about the soundness of method bodies. □

3.5 Evaluation

A source-level variant of this system, as an extension to C#, is in use by a large project at Microsoft, as their primary programming language. The group has written several million lines of code, including: core libraries (including collections with polymorphism over element permissions and data-parallel operations when safe), a webserver, a high level optimizing compiler, and an MPEG decoder. These and other applications written in the source language are performance-competitive with established implementations on standard benchmarks; we mention this not because our language design is focused on performance, but merely to point out that heavy use of reference immutability, including removing mutable static/global state, has not come at the cost of performance in the experience of the

Microsoft team. In fact, the prototype compiler exploits reference immutability information for a number of otherwise-unavailable compiler optimizations.

3.5.1 Differences from Formal System

The implementation differs from the formal system described earlier in a number of ways, mostly small. The most important difference is that the implementation supports proper task parallelism, with a first-class (unstructured) join. Task parallelism is supported via library calls that accept isolated delegates (closures, which must therefore capture only isolated or immutable state, in correspondence with T-Async) and return isolated promises, thus interacting nicely with recovery and framing, since the asynchronous task's mutable memory is disjoint from the main computation's. `async` blocks are not currently checked according to T-Async, mostly because we restrict `async` block task execution to single-threaded cooperative behavior, multiplexing `async` block tasks on a single CLR thread,[4] which already reduces concurrency errors from its use, so the team has not yet decided to undertake the maintenance task of turning on such checking. This permits some unchecked concurrency, but single-threaded (avoiding at least memory model issues) and with only explicit points of interference (an `await` expression basically acts as a yield statement; essentially cooperative concurrency). The team plans to eventually enable checking of `async` blocks as well. T-Par is not used for asynchronous tasks because it is unsound: recovery (T-RecovIso and T-RecovImm) is not valid if a shared readable reference to mutable data can live arbitrarily long after the "recovery block" in an asynchronous task. Thus T-Par is used only for applicable static constructs such as parallel for loops.

There are also a few source-level conveniences added as compared to the system here. The most notable is immutable classes. Immutable classes are simply classes whose constructors are required to have a final type environment with `this` : immutable rather than isolated. This allows the constructor to internally treat the self pointer as writable or isolated, before the type system conceptually uses T-RecovImm. Thus, writable and readable constructor arguments are permitted; they simply cannot be stored directly into the object.

[4] In C#'s implementation, `async` blocks may run on other threads [26], but the team decided prior to adding reference immutability that such behavior was too error prone.

The disadvantage of this is that it is not possible, without unsafe casts, to create a cycle of objects of immutable *classes* (cycles of immutable objects in general remain possible as in Section 3.1.3).

The source variation also includes an *unstrict* block, where permission checking is disabled. The eventual goal is for this to be used only in trusted code (whose .NET assemblies are marked as such), for optimizations like lazy initialization of immutable data when an accessor is called; the core libraries offer a set of standard abstractions to encapsulate these unsafe actions (Section 3.5.7). Finally, the source language uses only a single type parameter list, where each argument may be instantiated with a single permission or full permission-qualified type.

C# also permits compound expressions, and expressions with side-effects, which our core language disallows. `consume` is an expression in the source language, which performs a destructive read on its l-value argument and returns the result. This makes using isolated method arguments more convenient than in our core language, which allows statement consumption of fields, but treats isolated variables as affine resources when passed to methods.

A common focus for safe data-parallelism systems is handling of arrays. At the time of formalization, the implementation lacked direct support for arrays, instead using trusted library abstractions for safe data parallelism. Since formalizing this design, the Microsoft team has continued work to support a notion of a sub-array, using a combination of isolation and immutability to allow safe array partitioning for in-place updates, as well as functional data-parallelism.

3.5.2 Differences from C#

Beyond adding the obvious immutability-related extensions and restricting `async` block task execution to a single-threaded model (augmented with a true task parallelism library) as already discussed, the only additional difference from C# is that all static (global) state must be immutable. This is necessary for safe parallelism and for the recovery rules to avoid capturing shared mutable state. This restriction does lead to some different coding patterns, and required introducing several internally-unsafe but externally-safe library abstractions

for things like global caches, which we will discuss shortly.

3.5.3 Source-Level Examples

Generic Collections Collection libraries are a standard benchmark for any form of generics. The source variant of our system includes a full collections library, including support for polymorphism over permissions of the collection itself and elements of the collection. An illustrative example is the following simplified collections interface (using a lifting of our notation to a source language with interfaces, retaining our separation of permission and class parameters):

```
public interface ICollection<Elem><PElem> {
    public void add(PElem Elem e) writable;
    public writable Enumerator<Elem><P,PElem> getEnumerator() P;
}
public interface IEnumerator<Elem><PColl,PElem> {
    public bool hasNext() readable;
    public PColl⇝PElem Elem getNext() writable;
}
```

This collection interface is parameterized over a class type for elements and a permission for the elements (which may not be instantiated to isolated). The `add` method is natural, but the interesting case is `getEnumerator`. This method returns a writable enumerator, but the enumerator manages two permissions: the permission with which `getEnumerator` is called (which governs the permission the enumerator will hold on the collection) and the permission the collection has for the elements.

These separate permissions come into play in the type of the enumerator's `getNext` method, which uses deferred permission composition ($p \rightsquigarrow p$, Section 3.4) to return elements with as precise a permission as possible. Simply specifying a single permission for the elements returned requires either specifying a different enumerator variant for every possible permission on the collection, or losing precision. For example, given a writable collection of immutable elements, it is reasonable to expect an iterator to return elements with an immutable permission. This is straightforward with a `getEnumerator` variant specific to writable collections, but difficult using polymorphism for code reuse. Returning (using the

enumerator definition's parameters) `PElem` elements is in general not possible with a generic `PColl` permission on the collection because we cannot predict at the method definition site the result of combining the two permissions when the enumerator accesses the collection; it would be sound for a writable collection of immutable objects, but not for an immutable collection of writable objects since immutable $\triangleright_{\Delta}$ writable = immutable, not writable. It also preserves precision, as any element from enumerating an immutable collection of readable references should ideally return immutable elements rather than the sound but less precise readable.

Consider a linked list as in Figure 3.21. The heart of the iterator's flexibility is in the type checking of the first assignment in `LLEnum.getNext`. There the code has a `PColl` permissioned reference `next` to a linked list node that contains a `PElem` permissioned reference field `item` to an element. Thus the result type of `next.item` is `PColl`$\rightsquigarrow$`PElem PE` by T-FieldRead and $\triangleright_{\Delta}$. When the linked list type is instantiated, and `getEnumerator` is called with a certain permission, the enumerator type becomes fully instantiated and the deferred combination is reduced to a concrete permission. For example:

```
writable LinkedList<IntBox><writable> ll =
    new LinkedList<IntBox><writable>();
writable IEnumerator<IntBox><writable,writable> e =
    ll.getEnumerator(); // P instantiated as writable
writable IntBox b = e.getNext();

writable LinkedList<IntBox><readable> llr =
    new LinkedList<IntBox><readable>();
writable IEnumerator<IntBox><writable,readable> e =
    llr.getEnumerator(); // P instantiated as readable
writable IntBox b = e.getNext(); // Type Error!
// e.getNext() returns readable, since w⇝r=r
readable IntBox b = e.getNext(); // OK
```

A slightly richer variant of this enumerator design underlies the prototype's `foreach` construct, and is used widely in the Microsoft team's code.

Data Parallelism Reference immutability gives our language the ability to offer unified specializations of data structures for safe concurrency patterns. Other systems, such as the

```
public class LinkedList<Elem><PElem>
  implements ICollection<Elem><PElem> {
    protected writable Node<Elem><PElem> head;
    protected class Node<Elem><PElem> {
        public PElem Elem item;
        public writable Node<Elem><PElem> next;
    }
    protected class LLEnum<E><PColl,PE>
      implements IEnumerator<E><PColl,PE> {
        private PColl Node<Elem><PE> next;
        public LLEnum(PColl LinkedList<E><PE> coll) {
            next = coll.head;
        }
        public bool hasNext() readable { return next == null; }
        public PColl⇝PElem E getNext() writable {
            if (next != null) {
                PColl⇝PElem E nextElem = next.item;
                next = next.next;
                return nextElem;
            }
            return null;
        }
    }
    public LinkedList() { head = null; }
    public void add(PElem Elem e) writable {
        writable Node<Elem><PElem> n = new Node<Elem><PElem>();
        n.item = e;
        n.next = head;
        head = n;
    }
    public writable Enumerator<Elem><P,PElem> getEnumerator() P {
        return new LLEnum<Elem><P,PElem>(this);
    }
}
```

Figure 3.21: A simplified collection with a polymorphic enumerator.

collections libraries for C# or Scala separate concurrency-safe (immutable) collections from mutable collections by separate (but related) trees in the class hierarchy.[5]

A fully polymorphic version of a `map` method for a collection can coexist with a parallelized version `pmap` specialized for immutable or readable collections. Consider the types and extension methods [225] (intuitively similar to mixins on .NET/CLR, though the differences are non-trivial) in Figure 3.22, adding parallel map to a `LinkedList` class for a singly-linked list (assuming the list object itself acts as a list node for this example). Each maps a function[6] across the list, but if the function requires only readable permission to its arguments, `pmap` may be used to do so in parallel. Note that the parallelized version can still be used with writable collections through subtyping and framing as long as the mapped operation is pure; no duplication or creation of an additional collection just for concurrency is needed. With the eventual addition of static method overloading by permissions (as in Javari [252]), these methods could share the same name, and the compiler could automatically select the parallelized version whenever possible.

3.5.4 Optimizations

Reference immutability enables some new optimizations in the compiler and runtime system. For example, the concurrent GC can use weaker read barriers for immutable data. The compiler can perform more code motion and caching, and an MSIL-to-native pass can freeze immutable data into the binary.

A common concern with destructive reads is the additional memory writes a naïve implementation (such as ours) might incur. These have not been an issue for us: many null writes are overwritten before flushing from the cache; the compiler's MSIL is later processed by one of two optimizing compilers (.NET JIT or an ahead-of-time MSIL-to-native compiler) that often optimize away shadowed null writes; and in many cases the manual treatment of uniqueness would still require storing null.

---

[5] C# and Scala have practical reasons for this beyond simply being unable to check safety of parallelism: they lack the temporary immutability of our system due to the presence of unstructured parallelism.

[6] `Func1` is the intermediate-language encoding of a higher-order procedure. C# has proper types for these, called *delegate types* [225], each compiled to an abstract class with an `invoke` method with the appropriate arity and types. We restrict our examples to the underlying object representation for clarity.

```
public abstract class Func1<In,Out><Pin,Pout,Prun> {
  public abstract Pout Out invoke(Pin In in) Prun;
}
public static class LinkedListExtensions {
  // A parallel map
  public static readable LinkedList<readable><X>
    pmap<X>(
      this readable LinkedList<readable><X>,
      immutable Func1<X,X><readable,readable,readable> fun)
    readable {
      readable LinkedList<readable><X> rest = null;
      isolated LinkedList<readable><X> head = null;
      head =                          || if (list.next != null)
        new LinkedList<readable><X>;  ||   rest =
      head.elem = fun(list.elem);     ||     list.next.map<X>(fun);
      head.next = rest;
      return head;
  }
  // A polymorphic map
  public static writable LinkedList<PL⇝PE><X>
    map<X><PE>(
      this PL LinkedList<PE><X> list,
      immutable Func1<X,X><PL⇝PE,PL⇝PE,readable> fun) PL {
      writable LinkedList<PL⇝PE><X> result =
          new LinkedList<PL⇝PE><X>;
      result.elem = fun(list.elem);
      writable LinkedList<PL⇝PE><X> newCurr = result;
      PL LinkedList<PE><X> oldCurr = list;
      while (oldCurr.next != null) {
        newCurr.next = new LinkedList<PL⇝PE><X>;
        newCurr = newCurr.next;
        oldCurr = oldCurr.next;
        newCurr.elem = fun(oldCurr.elem);
      }
      return result;
  }
}
```

Figure 3.22: Extension methods to add regular and parallel map to a linked list.

3.5.5 Contracts

We have also integrated design-by-contract [169] in the style of .NET Code Contracts [81] into the language. Reference immutability is used to enforce external purity of these clauses by type-checking each clause in an environment where any writable or isolated reference is treated as readable, which therefore prohibits the modification of any checked state. This use of reference immutability has revealed additional bugs in code ported into this system, cases where a Code Contracts clause did incorrectly modify state.

3.5.6 Evolving a Type System

This type system grew naturally from a series of efforts at safe parallelism. The initial plans included no new language features, only compiler plugins, and language extensions were added over time for better support. The earliest version was simply copying Spec#'s `[Pure]` method attribute [18], along with a set of carefully designed task- and data-parallelism libraries. To handle rough edges with this approach and ease checking, readable and immutable were added, followed by library abstractions for isolated and immutable. After some time using unstrict blocks to implement those abstractions, we gradually saw a way to integrate them into the type system. With all four permissions, the team was much more eager to use reference immutability. After seeing some benefit, users eagerly added readable and immutable permissions.

Generics were the most difficult part of the design, but many iterations on the design of generic collections produced the design shown here. The one aspect we still struggle with is the occasional need for *shallow* permissions, such as for a collection with immutable membership, but mutable elements. This is the source of some unstrict blocks.

The entire design process was guided by user feedback about what was difficult. Picking the right defaults had a large impact on the users' happiness and willingness to use the language: writable is the default annotation, so any single-threaded C# that does not access global state also compiles with the prototype. This also made converting existing code much easier.

Since this work was first presented [105], several important extensions have been made.

First, actor concurrency was added using isolated and immutable permissions for safe message passing [124, 111]. At the time of initial presentation, the large system implemented in this language was approximately 90% checked for safe concurrency. Since then the team has completed conversion, and the only unsafe concurrency in the system is in the relatively small runtime system and a handful of trusted core library primitives.

3.5.7 User Experience

Overall, the Microsoft team has been satisfied with the additional safety they gain from not only the general software engineering advantages of reference immutability [252, 275, 276] but particularly the safe parallelism.

Anecdotally, they claim that the further they push reference immutability through their code base, the more bugs they find from spurious mutations. The main classes of bugs found are cases where a developer provided an object intended for read-only access, but a callee incorrectly mutated it; accidental mutations of structures that should be immutable; and data races where data should have been immutable or thread local (i.e. isolated, and one thread kept and used a stale reference).

Annotation burden has been low. There is roughly 1 annotation (permission or consume) per 63 lines of code. These are roughly 55% readable, 16.8% `consume`, 16.5% immutable, 4.7% writable, 4.1% isolated, and 2.8% generic permissions. This is partly due to the writable default, as well as C#'s local type inference (e.g. `var x = ...;`). Thus, most annotations appear in method signatures. Note that because users added additional qualifiers for stricter behavior checking, this is not the minimal annotation burden to type check, but reflects heavy use of the system.

The type system does make some common tasks difficult. We were initially concerned that immutable-only global state would be too restrictive, but has been mitigated by features of the platform the Microsoft team develops on top of. The platform includes pervasive support for capability-based resource access for resources such as files. Global caches are treated as capabilities, which must be passed explicity through the source (essentially writable references). This requires some careful design, but has not been onerous. Making

the caches global per process adds some plumbing effort, but allows better unified resource management.

Another point of initial concern was whether isolation would be too restrictive. In practice it also adds some design work, but our borrowing / recovery features avoid viral linearity annotations, so it has not been troublesome. It has also revealed subtle aliasing and concurrency bugs, and it enables many affine reference design patterns, such as checking linear hand-off in pipeline designs.

The standard library also provides trusted internally-unstrict abstractions for common idioms that would otherwise require wider use of unstrict blocks. Examples include lazy initialization and general memoization for otherwise immutable data structures, caches, and diagnostic logging. There are relatively few unstrict blocks, of varying sizes (a count does not give an accurate estimate of unchecked code). Most of these are in safe (trusted) standard library abstractions and interactions with the runtime system (GC, allocator, etc., which are already not memory-safe). Over the course of development, unstrict blocks have also been useful for the Microsoft team to make forward progress even while relying on effectively nightly builds of the compiler. They have been used to temporarily work around unimplemented features or compiler bugs, with such blocks being marked, and removed once the compiler is updated.

The Microsoft team was surprisingly receptive to using explicit destructive reads, as opposed to richer flow-sensitive analyses [40, 183] (which also have non-trivial interaction with exceptions [250]). They value the simplicity and predictability of destructive reads, and like that it makes the transfer of unique references explicit and easy to find. In general, the team preferred explicit source representation for type system interactions (e.g. `consume`, permission conversion).

The team has also naturally developed their own design patterns for working in this environment. One of the most popular is informally called the "builder pattern" (as in building a collection) to create frozen collections:

```
isolated List<Foo> list = new List<Foo>();
foreach (var cur in someOtherCollection) {
  isolated Foo f = new Foo();
```

```
  f.Name = cur.Name;
  // etc ...
  list.Add(consume f);
}
immutable List<Foo> immList = consume list;
```

This pattern can be further abstracted for elements with a deep clone method returning an isolated reference.

The safe concurrency features described here have handled most of the team's needs. Natural partitioning of tasks, such as in the H.264 and JPEG decoders (both verified for safe concurrency) is "surprisingly common," and well-supported by these abstractions. Sometimes breaking an algorithm into Map-Reduce-style phases helps fit problems into these abstractions. The main difficulties using the model come in two forms. The first form is where partitioning is dynamic rather than structural. This is difficult to express efficiently, and the team continues work on a framework to compute a partitioning blueprint dynamically. Second, sometimes communication among tasks is not required for correctness, but offers substantial performance benefits: for example, in a parallelized search algorithm, broadcasting the best-known result thus far can help all threads prune the search space. At the time this design was formalized, unstrict code was used for a few instances of this, motivating subsequent work work to add actors [124, 111] to the language.

3.6 Related Work

This section relates our work on reference immutability for data race freedom to other work in more detail than in Chapter 2. There we focused on the relationship between high level approaches, but here we compare technical details to related systems.

As described in Section 2.4.1, reference immutability [28, 252, 275, 276] is a family of work characterized by the ability to make an object graph effectively immutable to a region of code by passing read-only references to objects that may be mutated later, where the read-only effect of a reference applies transitively to all references obtained through a read-only reference. Common extensions include support for class immutability (classes where all instances are permanently immutable after allocation) and object immutability (making an

individual instance of a class permanently immutable). Surprisingly, despite safe parallelism being cited as a natural application of reference immutability, we are the first to formalize such a use.

Immutability Generic Java (IGJ) [275] is the most similar reference immutability work to ours, though it does not address concurrency. IGJ uses Java generics support to embed reference immutability into Java syntax (it still requires a custom compiler to handle permission subtyping). Thus reference permissions are specified by special classes as the first type parameter of a generic type. IGJ's support for object immutability is also based on the permission passed to a constructor, rather than conversion, so object immutability is enforced at allocation, and may not be deferred as our T-RecovImm rule allows. This means that creating a cycle of immutable objects requires a self-passing constructor idiom, where a constructor for cyclic immutable data structures must pass its `this` pointer into another constructor call as Immutable. Haack and Poll relax this restriction by lexically scoping the modification lifetime of an immutable instance [110].

Subsequent to the initial publication of this work [105], Servetto et al. [234] extended a similar core calculus with explicit borrowing of unique references using `@Lent` references to the root of a *balloon* [5] (roughly, externally unique cluster), and implemented it as a refinement of Java's type system. Their solution is similar to the treatment of taking references to isolated references in the Microsoft prototype. Servetto et al.'s goals are slightly different, however: their goal is that given the appropriate `@Balloon` (isolated) annotations, the compiler should infer where it is safe to place fork-join blocks — automatic parallelization. Our goals took a slightly different focus, of offering additional tools to expressing where mutation was expected, and using those to check that explicit parallelism were safe. One of the initial goals of the Microsoft prototype, however, was to support implicit parallelism when safe: adding permission-based method overloading would permit, for example, implicitly parallelizing a list traversal for a readable list and suitable traversal operation by giving appropriate overloads to a `map` method.

Ownership [39, 38, 65, 54] and Universe [68] types describe a notion of some objects "owning" others as a method of structuring heap references and mutation. The "owner-as-modifier" interpretation of ownership resembles reference immutability: any modification

to an object o must be done through a reference to o's owner. These systems still permit references across ownership domain, but such references (`any` references in Universe types) are deeply read-only. Universe types specify a "viewpoint adaptation" relation used for adapting type declarations to their use in a given context, which directly inspired our permission adaptation relation. Leino, Müller, and Wallenburg [153] boost the owner-as-modifier restriction to object immutability by adding a freeze operation that transfers ownership of an object to a hidden owner unavailable to the program source. Since the owner cannot be mentioned, no modifications may be made to frozen objects. In general, ownership transfer, (as in Leino et al.'s system or UTT [180]) relies on uniqueness and treats ownership domains as regions to merge.

The most similar treatment of polymorphism over mutability-related qualifiers to our work is Generic Universe Types (GUT). GUT provides polymorphism over permission-qualified types as in our prototype source language, rather than separating qualifiers and class types as our core language does. GUT's viewpoint adaptation (roughly equivalent to our permission combining relation $\triangleright_\Delta$) deals immediately with concrete qualifier combinations, and preserves some precision when combining a concrete ownership modifier with a generic one. But when combining two generic ownership modifiers, the result is always `any`, roughly equivalent to readable in our system. In practice, this is sufficient precision for GUT, because passing a generic type across ownership domains typically converts to `any` anyways. Our use cases require additional precision, which we retain by using deferred permission combination ($p \rightsquigarrow p$) to postpone the combination until all type parameters are instantiated. Without this, the generic enumerator in Section 3.5.3 could only return readable elements, even for a writable collection of immutable elements. IGJ [275] does not discuss this type of permission combining, but appears to have a similar loss of precision: i.e. accessing a field with generic permission through a reference with generic permission always yields a readable reference. OIGJ [276] can express generic iterators, but mostly because reference immutability in OIGJ is not transitive: a read-only iterator over a mutable list is permitted to mutate the list, and an iterator over an immutable list of mutable elements can return mutable references to those elements.

Ownership type systems have been used to achieve data race freedom [39, 38, 65], es-

sentially by using the ownership hierarchy to associate data with locks and beyond that enforcing data race freedom in the standard way [85, 86, 87]. Clarke et al. [55] use ownership to preserve thread locality, but allow immutable instances and "safe" references (which permit access only to immutable, e.g. Java-style `final`, portions of the referent) to be shared freely across threads, and add transfer of externally unique references.

Östlund et al. [203] present an ownership type and effect system for a language Joe_3 with the explicit aim of supporting reference immutability idioms by embedding into an ownership type system. Owner polymorphic methods declare the effects they may have on each ownership domain, treating ownership domains as regions [246, 249]. Joe_3 uses a similar technique to ours for delayed initialization of immutable instances, as it has (externally) unique references, and writing a unique reference to an immutable variable or field converts the externally unique cluster into a cluster of immutable objects. While Joe_3 has blocks for borrowing unique references, our T-RecovImm rule is more general, combining borrowing and conversion. Their borrowing mechanism also creates local owners to preserve encapsulation, requiring explicit ownership transfer to merge aggregates. Our type system also permits invoking some methods directly on unique references (as opposed to the required source-level borrowing in Joe_3) because our frame rule makes it easy to prove the invocation preserves uniqueness with the additional argument restrictions (Figure 3.5).

Our T-RecovIso rule is in some ways a simplification of existing techniques for borrowing unique references, given the presence of reference immutability qualifiers. The closest work on borrowing to ours in terms of simplicity and expressiveness is Haller and Odersky's work [112] using capabilities that guard access to regions containing externally-unique clusters. Regions of code that return an input capability have only borrowed from a region, and have not violated its external uniqueness. Local variable types $\rho \triangleright \tau$ are references to an object of type τ in region ρ. When a reference to some object in an externally unique aggregate is written to a heap location, that aggregate's capability is consumed in a flow-sensitive fashion, and all local variables guarded by that capability become inaccessible. Our recovery rule requires no invalidation, though its use may require environment weakening. We believe the expressiveness of the two approaches to be equal for code without method calls. For method calls, Haller and Odersky track borrowing of individual arguments by

what permissions are returned. Our system would require returning multiple isolated references through the heap, though our recovery rules would allow inferring some method returns as isolated in the proper contexts, without cooperation of the method called. We also add some flexibility by allowing arbitrary paths to immutable state reachable from an externally-unique aggregate.

Another interesting variant of borrowing is *adoption* [80, 42], where one piece of state is logically embedded into another piece, which provides a way to manage unique references without destructive reads.

Boyland proposed fractional permissions [41] to reason statically about interference among threads in a language with fork-join concurrency. One of the main encumbrances of his original system was the appearance of explicit fractions in source programs. Since then, many verification tools have inherited this feature [151], though recent work alleviates developers of this particular nuisance [123]. We use a twist on fractional permissions in the denotation of our types, including to denote uniqueness, though the fractions do not appear in the source program, type system, or compiler implementation. The Plaid language uses fractional permissions to manage aliasing and updates to objects when checking typestate [24]. Recent work by Naden et al. to simplify Plaid [183], like our system, does not require any fractions or explicit permission terms to appear in source code, though unlike us an implementation of their type system must reason about fractional permissions (our fractional permissions appear only in our meta-proofs). Like us they use type qualifiers to specify access permissions to each object, though their permissions do not apply transitively (a distinction driven largely by our differing motivations). Naden's language supports externally-unique and immutable references, and more fine-grained control than our system over how permissions for each argument to a method are changed (e.g. preserving uniqueness as an indication of borrowing), though his language does not address concurrency.

Deterministic Parallel Java (DPJ) [35] uses effect typing and nested regions [246, 249] to enable data parallelism and deterministic execution of parallel programs. An expression may have a read or write effect on each of some number of regions, and expressions with non-conflicting effects (one thread with write effect and none with read effects, or multiple threads with read effects and no write effects, on each region) may safely be parallelized.

This ensures not only the absence of data races, but determinism as well (later revisions add controlled nondeterminism [34]). Our system is similar in spirit, but requires no mention of regions in the source, only mention of the permissions required for each object a method accesses, not where they reside; our regions exist only in the denotation of permissions. This means, for example, that region polymorphism is implicit in our system; methods need not bind to a specific number of regions, while DPJ requires methods and classes to declare fixed numbers of regions over which they operate (it is possible to instantiate multiple region arguments with the same region). The advantage to the explicit treatment of regions in DPJ is that non-interfering writes may be parallelized without requiring any sort of reference uniqueness (the type system must still be able to prove two regions are distinct). DPJ also treats data parallelism over arrays, whereas we do not.

Westbrook et al. [268] describe Habanero Java with Permissions (HJp), a language with parallelism structure between our formal and source languages: async(e) begins an asynchronous task and returns unit; finish(e) waits for all tasks spawned within e, rather than allowing joins with individual tasks. They use qualifiers to distinguish thread-local read, thread-local write, and thread-shared read access to objects (the latter is mutually exclusive with the first two). They must distinguish between thread local and shared read-only access because they cannot guarantee the inaccessibility of writable references to objects for the duration of an async; doing so would require a flow-sensitive set of variables inaccessible until the enclosing finish() because the end of an async is not statically scoped, and async blocks may capture any shareable state, not only unique or immutable state. Their treatment of storing (totally) unique references in unique fields and embedding the referent's permissions is more flexible for concurrency than our isolated fields. Their embedding allows natural read-only access to unique field referents if the containing object is shared read-only, while isolated fields of shared readable objects are inaccessible until recovery or conversion. Reads from non-unique fields in HJp have no static permissions; dereferencing such fields requires dynamically acquiring permissions. We treat permission polymorphism, while they do not.

3.7 Conclusions

We have used reference immutability to ensure safe (interference-free) parallelism, in part by combining reference immutability with external uniqueness. Applying our approach to an intermediate-level language led us to derive recovery rules for recovering isolation or immutability in certain contexts, which offers a natural approach to borrowing for languages with reference immutability. Our type system models a reference immutability system in active use in industry, and we have described their experiences with it.

Taking a step back, the similarities between this concurrency-aware reference immutability system and its sequential bretheren [28, 252, 275, 276, 132] speak to the flexibility of taking non-interference in the form of read-sharing as a central concept in a language design. The presence of read-sharing primitives gives a natural form to support read-sharing among concurrently-executed threads. Given that non-interference embodied here and described earlier in Chapter 2.4 is a special case of interference summaries (Chapter 2.5), the question naturally arises as to whether this gentle step from sequential to concurrent reasoning remains gradual for the more general case where interference is permitted but bounded. This is the question the remainder of this thesis answers, in the affirmative.

Chapter 4

RELY-GUARANTEE REFERENCES

> Only gradually, my resistance to that kind of formula manipulation faded, until eventually, I began to love it.
>
> Dijkstra, EWD 1298

This chapter is largely a reworking and expansion of the contents of the PLDI'13 paper *Rely-Guarantee References for Refinement Types Over Aliased Mutable Data* [103] and its associated technical appendix [104]. This is joint work with Michael Ernst and Dan Grossman.

4.1 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 [252, 275, 276, 132, 105] (Chapter 3), but also in ownership [68] and region-based type systems [35]. 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 previous chapter illustrated that per-reference read-sharing support in a language permitted both (1) a gradual transition from sequential to concurrent programming, and (2) a high degree of expressiveness, based on several years of usage in an industrial context. Given that read-sharing is a special case of the more general notion of interference [223], it is natural to ask whether the more general phenomenon may be treated in a similar manner. This chapter explores one approach to treating interference more generally in a per-reference approach to controlling mutation. Here we discuss only sequential programming again; concurrency is deferred until Chapter 5.

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 [204], 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 [139]. 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 [252, 275, 276, 105] (Chapter 3) to finer-grained control over interference through aliases. The traditional reference immutability qualifiers correspond to simple rely and guarantee conditions. For $\mathsf{ref}\ \tau$[R,G] as a reference to data of type τ with rely R and guarantee G:

- $\mathsf{readable}\ \tau \equiv \mathsf{ref}\ \tau$[any interference, no writes]
- $\mathsf{writable}\ \tau \equiv \mathsf{ref}\ \tau$[any interference, any writes]

- immutable $\tau \equiv$ ref τ[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.)

4.1.1 Contributions

Refinement Types Over Mutable Data Rely-guarantee references permit refinement types [95] 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 this chapter, 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.

4.2 Rely-Guarantee References

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

- A refinement predicate P over the τ and a heap h that can enforce local properties and/or data-structure well-formedness.

- A guarantee relation G over pairs of τs 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 τ 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 4.4.

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

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

where any is the trivial (always true) refinement, and increasing (Section 4.3.1) 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 [36, 151] 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 [139, 260, 82], but only coarsely [269] 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 [126]).

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

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

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.

4.2.1 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

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

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 4.4).

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 [105], to capture the transitive meaning of 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 4.3.2). Furthermore, if one reference points to another, how should the rely of the "inner" reference be related to the outer one? It is un-

sound 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.

4.3 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 4.4), we present them using a slight simplification of our shallow embedding in COQ [59]. The embedding is largely in the style of YNOT [189, 52], 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

[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.

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 [240] (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.

4.3.1 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 4.5), 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

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

which is easily solved, with little effort beyond what is required to verify a pure-functional increment function (see Section 4.5.3). 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 [215] as a challenging goal for program verification. Unlike their solution and another in fictional separation logic [137], 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 this chapter's design merely ensures that increment is the only permitted action. Chapter 5 extends the system for proving functional correctness.

4.3.2 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 4.1 shows the latter.[3]

We first define `hpred` and `hrel`, type-level functions that allow shorter type declarations. We use them throughout this chapter. Next we define a linked list structure, with restricted references to the tail. `list_imm` constrains the tail to be immutable. For the reader unfamiliar with Coq, `list_imm` is a GADT [271] 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_container`s 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 ($\cap$) 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.

---

[3] The most natural design uses inductive-inductive types [6, 94] (simultaneously-defined 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 4.5.

```
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 4.1: RGREF code for a prepend-only linked list.

Not shown in Figure 4.1 are implicit obligations such as $\forall h.\, P$ `nil` h in `Nil`. Other such obligations include:

- $\forall tl, h.\, P\ tl\ h \implies P'$ (`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 4.1 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 component-wise 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.

4.3.3 Reference Immutability as a Special Case

Reference immutability [28, 252, 275, 276, 105, 132] (Chapter 3) 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 4.3.2) 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 4.2). 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 4.2. 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.

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

Figure 4.2: Combining reference immutability permissions, from [105]. Using a p reference to read a $q\ T$ field produces a $(p \triangleright q)\ T$.

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

```
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 4.3.1. This offers a natural path for gradually adding stronger verification guarantees to code using reference immutability.

4.3.4 RCC/Java with Reference Immutability

The core of RCC/Java [87] is also implementable as a small library using our Coq DSL, and we present a translation of an early version [86]. 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 ref{lockwitness | locked}[empty, locked-->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:

```
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.

4.3.5 Function Memoization

We have implemented a simple imperative memoization for dependently typed functions. We use a reference to a function, constrained using RGref's facilities to only change to observationally equivalent (extensionally equal) functions.

For natural numbers — which have decidable equality — we introduce a function `prefernat`:

```
Definition prefernat
  {P:nat->Set}(g:forall x:nat, P x)(a:nat)
      : forall x:nat, P x.
  refine(let v := g a in
          (fun x => if (* x == a *) eq_nat_dec x a then _ else g x)).
  subst. exact v.
Defined.
```

The `_` is filled in with the elimination of the equality between `x` and `a`, producing (under call-by-value) the memoized value `v` as an object of type `P x` rather than its original `P a`.

We can then define an imperative RGref function to accept a reference to a function, and update it in an observationally equivalent way to prefer a certain value:

```
Program Definition prioritize {Γ}
(r:ref{forall x:nat, B x|any}[obs_equiv,obs_equiv]) (n:nat)
: rgref Γ unit Γ :=
  [r]:= prefernat (!r) n.
```

This construction could trivially be extended to any domain type with a decidable equality.

4.4 A Type System for Rely-Guarantee References

Figure 4.3 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 [62, 14]. We include a few basic datatypes (natural numbers, booleans, pairs, unit), a type for propositional equality, and their standard recursors [129]. 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 [12]. 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 4.5).

Figures 4.4 and 4.5 present 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 $\Gamma \vdash \tau \prec \tau \circledast \tau$ to split a type (V-$\circledast$). 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 (Π-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 (Π-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

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

Figure 4.3: 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 [14] pseudoterms.

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 4.2.1 has non-trivial splitting behavior, mediated by Ref-⁕:

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

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,

$$\begin{aligned}\epsilon \vdash \mathsf{ref}\{\mathtt{nat} \mid \mathsf{any}\}[\mathsf{decreasing}, \mathsf{increasing}] \\ \prec \mathsf{ref}\{\mathtt{nat} \mid \mathsf{any}\}[\mathsf{havoc}, \mathsf{increasing}] \\ \text{⁕} \; \mathsf{ref}\{\mathtt{nat} \mid \mathsf{any}\}[\mathsf{havoc}, \mathsf{increasing}]\end{aligned}$$

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-⁕.

The conversion relation $\Gamma \vdash \tau \rightsquigarrow \tau$ is a directed call-by-value β-conversion (so for example β-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-⇒ weakens the predicate; R-⊂ assumes more interference may occur; and G-⊂ 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,

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

[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.

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 4.7) 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 [210, 75] 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 4.7 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 [22] (CIC) beyond richer treatment of folding for the full spectrum of inductive types (Section 4.5.2); remember, however, that this pertains only to specification expressiveness, as the pure fragment is not normalizing. 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 ($\Gamma \vdash \tau \prec \tau \mathbin{⅋} \tau$). In principle this could be weakened, but this design avoids the need for linear function types. We build upon CC for simpler presentation.

4.4.1 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 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.

To make this proof easier, the dynamic semantics for RGref have a few quirks:

- Variable binding occurs by stack usage in impure code (making it easier to prove substructural behavior), and by substitution in pure code (justified by the requirement that any variable used in pure code is self-splitting).

- Variables captured by either a pure or impure closure are required to be self-splitting, but it would be unsound to allow those captured uses to read the stack location if the closure were executed after impure code split the stack reference with different permissions. To prevent this, closures are only values when they have no free variables, and there are reduction rules that perform the reads for free variables.

- References are represented as "fat pointers" of the underlying pointer, rely, guarantee, and predicate

- Casts/subsumption and substructural consumption/splitting of variables are explicit — including explicit mention of result predicates and relations — so the semantics can appropriately modify underlying reference values. The translation from the source language presented here to the more explicit language is straightforward by induction on the typing derivation.

- We assume the heap behaves according to the axiomatization used in rely/guarantee conditions and predicates.

- Folding has a runtime representation. Reads produce the heap value wrapped in a deferred folding construct which lazily pushes the guarantee restriction through a

data structure as components are evaluated. This is necessary for dependent type constructors like propositional equality, where types appear in values of the type.

- The semantics occasionally reduce multiple dereferences of the same location simultaneously, if they are constrained by types to be definitionally equal. We have yet to conceive of desirable *computational* code that observes this behavior; see Sections 4.4.4 and 4.5.1.

4.4.2 Operational Semantics

This section describes the operational semantics of the core language from Section 4.4. Figure 4.9 gives the most interesting rules, while Figure 4.10 gives the remaining rules that effectively define contexts and evaluation order. Recursors (not shown) follow the standard reduction rules.

The most unusual part of the semantics is the imperative reduction "evaluation context" rule for reducing pure expressions in an impure context, mentioned earlier. This rule sometimes reduces multiple dereferences of the same location in one step, including inside unevaluated closures.

The semantics reducing multiple dereference expressions is very unusual, and could hypothetically lead to unexpected results. Implementations could impose stronger analyses to prevent writing terms that would observe the change in behavior (thus obviating the unusual semantics). However, we have been unable to conceive of a desirable term that observes this atypical reduction; recall that we use the pure fragment for two different purposes: computation and specification. Observing the multiple-dereference behavior requires using dereference expressions in types.

The simplest example we can think of and a few reduction steps are shown in Figure 4.8. Assuming ℓ is a reference to a natural number (3 in the current heap), the type of this term is $\mathsf{unit} \to \mathtt{nat}$. The first reduction shown substitutes the reference, and the second performs the first actual dereference. The result of the second reduction (the last line) only type-checks because of the untagging conversion rule, which says $\langle !\ell \rangle 3 \rightsquigarrow !\ell$. If the

tag were stripped from the term without further ado and reduction proceeded naïvely, the recursor's result terms would be a proof that $3 = 3$ and a proof that $!\ell = !\ell$. While this is typeable, it would require introducing additional dependencies into the type (not strict preservation) and we don't know if even that would be possible in general. And while one could conceivably coerce the new term to type-checking using the heap for unrestricted definitional equality, this would be unsound: this problematic term could be reduced inside an impure computation which subsequently stored 4 to ℓ's heap cell. For this reason, our semantics would produce the final term in the figure, deduce that immediately reducing the inner dereference would ensure the term is well-typed in *any* heap, and perform that additional dereference.

Note that to observe the unusual semantics, a term must:

- Dereference a location in a context that introduces the dereference expression into a type (such as the redex location of a reflexivity proof), in a part of the term that is reduced before returning to the impure context

- Return a closure to the surrounding impure context, which also dereferences the same location in a context that injects dereference into a type

- Relate (via types) the "outer" and "inner" terms whose types depend on dereference of the same location.

If a pure embedding's type is any "flat" type (not containing a closure), then the multiple-dereference semantics are irrelevant; all dereference expressions in the pure context will be reduced in the same heap (or thrown away, for example from an unused branch of a recursor). The heap_indep step also imposes restrictions on reasoning about pure terms; see Section 4.5.1.

4.4.3 Static Semantics of Dynamic State

Figure 4.11 gives the typing judgments for dynamic program states.

4.4.4 Soundness

The proof of soundness follows the sketch in Section 4.4.1. The dynamic semantics are mostly standard for an ML-like language with references, with a couple small changes:

- Binding: Variable binding in the imperative core is stack-based, not substitution-based, to handle the substructural splitting behavior in the imperative language. Binding in the pure language is substitution based (essentially pre-splitting and inlining read results).[5]

- Variable capture: Variables captured by either a pure or impure closure are required to be self-splitting, but it would be unsound to allow those captured uses to read the stack location if the closure were executed after impure code split the stack reference with different permissions. To prevent this, closures are only values when they have no free variables, and there are reduction rules that perform the reads for free variables.

- Locations: Locations are represented by a tuple of not only the heap "index" (the traditional basic location) but also with the heap storage type, predicate and rely/guarantee relations explicitly tagged as part of the value. The tags are not required for functionality, but their presence simplifies the soundness argument.

- Subsumption: We actually prove soundness for a slight translation of the calculus to one with explicit reference conversion casts placed wherever expressions were typed using the rules P-$\Rightarrow$, R-$\subset$, and G-$\subset$.

- Substructural behavior: Variable splitting and dropping are explicitly identified in the source, to allow the semantics to modify the stack appropriately.

---

[5] In an implementation, substitution is fine for both bindings as the relations attached to arguments serve no operational purpose: they are present here only for proving that resource semantics are respected. This would also simplify interaction between the pure and impure languages, allowing self-splitting variable bindings to be captured between sublanguages. In our implementation embedded in Coq, variables of self-splitting types may be captured by either abstraction and used only purely; substructural variables are encapsulated in a monad.

- Step Granularity: Small-step reduction semantics are used, but the pure computation's semantics include only a single input heap, and no output, since the pure terms cannot modify the heap.

- Folding: Folding has a runtime representation. Reads produce the heap value wrapped in a deferred folding construct which lazily pushes the guarantee restriction through a data structure as components are evaluated. This is necessary for dependent type constructors like propositional equality, where types appear in values of the type.

- As mentioned in Section 4.4.2, it is sometimes necessary to reduce multiple heap dereferences at once. We have yet to imagine an example of desirable *computational* code that observes this difference.

While considering the role of predicate and guarantee obligations from the typing derivations, note that the type judgments themselves do not require proof terms to prove predicates and guarantee obligations. Those predicates and relations are *specified* using propositions-as-types, but the actual proof method is up to the implementation. Section 4.5.1 discusses the subtleties of using proofs-as-programs to actually prove various obligations.

Execution must preserve two critical invariants beyond standard invariants for well-typed heaps:

- 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 $\emptyset \subset G \subseteq R'$

Soundness proceeds as a type preservation proof, by induction on the evaluation step taken.

Lemma 17 (Pure Preservation)**.** *If* $H; \Sigma; \Gamma \vdash e : \tau$ *and* $H; e \to e'$*, then there exists some* τ' *such that* $H; \Sigma; \Gamma \vdash e' : \tau'$ *and* $H; \Gamma \vdash \tau' \rightsquigarrow \tau$*.*

The pure cases are mostly straightforward (recall that progress and preservation — subject reduction — hold for CC with β-conversion [14, 99]), so we focus discussion on the cases for impure rules. The one notable point in the pure cases is reduction of a dereference expression — assume $!\,r$ — which applies relation folding ($[R, G] \gg \tau$, Figure 4.7). In this case note that the folding — a form of weakening guarantees of read results — ensures that the guarantees in the result of $!\,r$ allow no more heap changes than r's guarantee. This is required to preserve the compatible alias guarantee, because some alias of r may have a rely equal to r's guarantee, so if a read result allowed too strong a guarantee, that result might allow actions that would violate the alias's rely, potentially invalidating refinements.

Evaluating $!\,r$ also produces labeled expressions, $\langle !\,r \rangle a$, where a is the folded result of the heap lookup, and is convertible to $!\,r$ itself to aid type preservation. The labeled expression is a witness of a heap dependency. It is the reduction of these heap reads that requires the type before and after be related by $H; \Gamma \vdash \tau' \rightsquigarrow \tau$, which gains an additional runtime rule that a dereference-tagged expression converts to the dereference expression. This is useful in structural rules like the inductive cases of pure function application, where for example a reduction of the function term could introduce a tagged value into the argument type. For the application to continue to check, appearances of the same (dereferenced) value in the type of the argument position must be convertible to the argument type.

All heap witnesses are removed (reduced to a) by another reduction rule (from the surrounding imperative context) before producing a value that flows back into imperative computation, taking advantage of the fact that replacing $\langle !\,r \rangle a$ by a preserves typing, up to replacing the tagged expression with a in the type. This is done in two steps by the context rule for embedding pure expressions within impure contexts, under the meta-function heap_indep. The runtime typing rule for the pure-in-impure embedding requires the pure expression typecheck without using the untagging conversion rule, so heap_indep rewrites the pure evaluation result to remove the use of untagging. This involves changing (as few as possible) instances of $!l$ to $\langle !l \rangle a$ in types when previously a conversion was necessary, and at constructors occasionally changing a term-level dereference as well. Thus, this is the reduction of multiple dereferences at once mentioned earlier. Then a substitution is performed using the following lemma:

Lemma 18 (Untagging)**.** *For all references ℓ and tagged expressions $\langle !\ell \rangle a$ such that $H; \Sigma; \epsilon \vdash \langle !\ell \rangle a : \tau'$, if $H; \Sigma; \Gamma \vdash e : \tau$ without using the untagging conversion, then $H; \Sigma; \Gamma \vdash e[\langle !\ell \rangle a/a] : \tau[\langle !\ell \rangle a/a]$.*

Proof. By inversion on the typing of the tagged value, $H; \Sigma; \epsilon \vdash a : \tau'$. Proof follows by induction on the typing derivation for e. The Conv case proceeds by induction on the type conversion, where the untagging case holds vacuously. □

Because only closed result types are permitted to flow from pure subterms back into imperative computation (Sub and Π-E), this extra "unlabelling" step preserves the type, so all values that persist across pure evaluations are heap-independent.

With these lemmas in hand, impure preservation is reasonably straightforward.

Lemma 19 (Impure Preservation)**.** *If $\vdash S; H; e : \Gamma; \Sigma; \tau \Rightarrow \Gamma'$, and $S; H; e \to S'; H'; e'$, then there exists a Γ'' and Σ' such that $\vdash S'; H'; e' : \Gamma''; \Sigma'; \tau \Rightarrow \Gamma'$.*

Proof. Note that the initial state (well-typed expression and the empty heap and stack) satisfies all invariants.

- Call: Reduction of the procedure or argument is sound by induction. In the case where the procedure is actually applied, the only affected parts of the state are the stack (which gains a fresh variable with the argument as its value, with the obvious type), and the expression (now the body with the fresh variable substituted for the source variable). Stack typing follows naturally, expression typing follows from α-renaming and a lemma that replacing well-typed subexpressions preserves typing, and heap typing is unchanged.

- Assign: Stack and expression typing is straightforward. Basic heap typing is straightforward. Establishing that the predicate holds in the new heap is straightforward, using inversion on the heap write type rule (including strong updates to predicates). This rule may create a new alias of some reference, but preserving non-conflicting relies and guarantees among aliases is straightforward. The more subtle part of this case is proving that all other references' predicates still have proofs in the new heap.

For direct aliases of the write target, by the compatible R/G condition, stability of all predicates over their respective relies, and the guarantee satisfaction (by inversion on the typing rule), those predicates are preserved. For references from which the modified cell is reachable, a similar reasoning applies using the containment requirements of well-formed reference types. For references from which the modified cell is not reachable, preservation is by the precision requirements on predicates.

- Swap: Similar to the assignment case.

- Allocate: Stack soundness is preserved, expression soundness is straightforward. Basic heap soundness is straightforward, leaving the RGref-specific heap invariants as remaining proof obligations. By inversion on the typing rule for allocation, there is a proof of the predicate on the allocated value in all heaps, therefore one exists for the new heap. The new object is (initially) unaliased, so all aliases' rely and guarantee imply each other. For previously-existing references, the allocation is not reachable from any existing allocation, so proofs are preserved by the fact that all existing references' predicates are precise (insensitive to changes outside their reachable heap).

- Drop-Var: Stack and heap typing are straightforward, as is expression typing. The main invariant that could be violated is that aliases' rely and guarantee conditions might conflict; this invariant is preserved because the operation moves an existing reference, it does not create an additional alias.

- Split-Var: Similar to the Drop-Var case, except the value is actually split according to the elaborated syntax for splitting variable reads, which is only well-typed if values of the type split according to $\Gamma \vdash \tau \prec \tau' \divideontimes \tau''$, which preserves compatible rely and guarantee conditions.

- Pure: Justified by soundness for the pure sublanguage, plus restrictions on pure/impure interactions (specifically that runtime typing requires that the pure expression must typecheck without using the heap as a source of definitional equality — i.e., without the untagging conversion).

□

4.5 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 4.3'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 [189, 52], 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 [240]. 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 [51] 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.[6] The implementation also relaxes some restrictions present in the formal language, such as allowing values to be projected into predicates (as in Section 4.3.2); predicates, rely, and guarantee relations must simply abstain from dereference.

We made compromises to fit into COQ. Notably, COQ lacks support for inductive-

[6]This axiom is valid only for proving guarantee satisfaction in the absence of recursive types or when applied to a reference that does not reach the modified location. This is because naive application of this axiom could assume a predicate holds of the modified location while proving that predicate holds; this corresponds to degenerate corecursion. The examples we have verified thus far are acyclic. The axiom remains unconditionally sound when discharging predicate obligations at allocation, or when proving stability. In general, guarded recursion [184, 29, 135, 30, 11, 182] could be used to expose this axiom simply in all contexts.

inductive types [6, 94] (e.g., a datatype simultaneously defined with an inductive predicate on that type). Our original implementation adapted a standard encoding [47] of induction-recursion [77] to support examples like the list in Section 4.3.2. Any use of this encoding somewhat complicates generated proof obligations and data structure designs, so we presently axiomatize the constructors and eliminators by hand. This carries its own costs, but proof obligations are generally simpler than with the adapted encoding. 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.

Our implementation focuses on self-splitting types; not all primitives for handling substructural data monadically have been implemented (or necessary) yet.

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 could generally be synthesized automatically for simple types.

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/`.

4.5.1 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 [14]. 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.

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 in the pure fragment, 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. Thus 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 prototype currently conservatively prohibits returning terms with deferred dereference expressions to the imperative context

Recursion Through the Heap

One important aspect the proof above ignores is the potential to recur through the heap, as this term would:

```
Program Example heap_recursion :=
    (* Allocate a unit->nat on the heap *)
    (* Assume predicate any, rely/guarantee havoc *)
    fn <- alloc (fun _:unit => 3);
    (* Close fn2 over fn *)
    fn2<- alloc (fun u:unit => (!fn) u);
    (* Point fn2's function reference to itself! *)
    fn <- fn2;
    return (!fn2) tt.
```

We do not consider this fundamentally problematic; it simply requires our implementation to prevent creating proof terms relying on such behavior. So proofs that a predicate

holds of a new allocation, or that a new predicate holds and a guarantee is satisfied after a write, are valid *when the allocated/stored expression terminates*, and otherwise the program does not terminate at runtime. Most rely-guarantee logics and other program logics are modulo termination, and this proof-by-nontermination issue arises only when the imperative code has explicitly used mutation to introduce indirect recursion. Disallowing recursion through the heap in the pure fragment is also straightforward: an implementation could for example restrict the storage of closures containing deferred dereference expressions into the heap. Or the recursion through the heap could be constrained to be *productive*, thus corresponding to *co*-recursion [184, 29, 135, 30, 11, 182] rather than arbitrary non-termination, which is permitted in logically consist end dependent type theories.

Equational Reasoning with Dereference

The unusual simultaneous-dereference semantics in the pure sublanguage highlight an important subtlety of equational reasoning (a subtlety that would exist even with alternative restrictions to remove the unusual semantics). Specifically, it is unsound to equate two dereference expressions that may be reduced in different heaps!

There are a few approaches to handling the issues with equating dereferences that are evaluated in different heaps. The approach we favor is to permit full equational reasoning only for properties of pure terms whose return type contains no closures with deferred dereference. In this case, all dereferences of the same location will be reduced with the same heap (before control returns to the imperative fragment), so equating syntactically identical dereferences is sound. In cases where a pure subterm returns a value that may contain unreduced dereference expressions, either the whole term or (a conservative overapproximation of) the closures that may be unreduced when the pure term becomes a value must be abstracted, hiding them from equational reasoning principles.

This is also the motivation for disallowing dereferences in the definition of predicates and relations in the core language. Rather than complicating the type system's core ideas with richer checks to prevent dereferences in predicates and the like, we simply prevent the introduction of references into those subterms. An alternative would be to introduce more

distinct syntactic categories so dereference expressions in predicates, rely, and guarantee relations simply would not even parse. Our prepend-only list in Section 4.3.2 does project a location (and another value) from the term language into a predicate, but this is sound because it is not used in a heap-sensitive way (dereference).

Our prototype does not enforce the required restrictions on equational reasoning, because it is in some sense "too shallow" of an embedded DSL: because we directly leverage Coq's dependent product for RGref's dependent product, we cannot restrict its application without an additional preprocessor or adding a compiled OCaml plugin to restrict interaction between native CIC and RGref-specific terms. We believe this is a reasonable trade-off. The prototype's goal is to evaluate the rely-guarantee reference approach's proof burden. The technically required restrictions on equational reasoning should not be difficult in principle to enforce, and we do not believe they would noticeably affect how code would be written. A production-quality implementation of RGref would of course need to enforce the richer restrictions. It is worth noting that this weakness from interaction between Coq's raw terms and DSL-specific terms is not unique to us; the Ynot [189, 52] implementation of Hoare Type Theory [188, 185] has similar risks when Coq primitives are used in unintended ways with Ynot axioms. This source of unsoundness could be avoided with a deeper embedding, as has been done for some separation logic work, or in a system not based on dependent type theory (which would need to construct many reasoning principles on its own).

4.5.2 Data Structure Design

We were able to use normal data structures for the examples in Section 4.3, but the types require careful consideration for propagating information through data flow. For example, the return type of `Cons` in Figure 4.1 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, leaving this as future work (Chapter 7). 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 require support only 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.

4.5.3 Proof Burden

RGref imposes proof obligations for precision, folding, containment, and guarantee satisfaction. For verifying the examples in Sections 4.3.1, 4.3.2, and 4.3.3, 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.[7] 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

[7] Not all are difficult; Section 4.3.1's guarantee obligation is discharged by automatic proof search with arithmetic hints: `auto with arith`.

in verifying functional programs can be reused in our implementation.

4.6 Future Work: Extensions and Adaptations

The type system we present in this chapter has a few technical limitations beyond the implementation limits described in Section 4.5.

The system we present here is modeled on the *deep* interpretation of reference immutability (the standard interpretation), where the permission of one reference constrains the permissions on read results through that reference (this can be seen best in the rely-guarantee folding in the dereference type rule). This is contrasted against the *shallow* interpretation of reference immutability, where a permission affects only the actions on a particular heap cell (so a writable reference could be obtained by reading through an immutable cell). Our formalism does admit uses of shallow relations and predicates: these propositions simply ignore the heaps, so containment and folding are trivial.

Our type system could be converted to a fully shallow model by:

- removing the rely-guarantee folding,
- removing the restrictions that a rely condition must account for the rely conditions of any reference reachable from it, and
- adding tighter footprint restrictions on rely-guarantee conditions and refinement predicates.

For the last point, we mean specifically that the rely, guarantee, and predicate should be insensitive to heap contents outside the heap cell they apply to. The simplest way to accomplish this is to remove the heap arguments from these predicates, making them closed over only the value itself. Note that any rely/guarantee and predicate matching these restrictions is already valid in our system, and folding and containment are semantically no-ops because component-wise projection of these more restricted relations do not impose any new restrictions.

One weakness of our current system is that every individual effectful action must fall within a reference's guarantee. Some operations (for example, splay tree rotation) are

difficult or impossible to write this way while satisfying guarantees that preserve interesting refinements on aliases (such as set membership), limiting the properties the system can verify. Other systems based on various forms of object invariant have a notion of *focusing* or *opening* an object for a series of operations that together preserve an invariant, but where intermediate states violate the invariant [24, 230, 18]. Type systems with focus often make the typing judgment modal, with one "unfocused" mode and one where a particular object is focused. We could add such support to our system. This would grant us additional flexibility, and also allow us to subsume much of the expressive power of Militão et al.'s recently proposed rely-guarantee inspired typestate system (see Section 4.7).

Another limitation is that effectful function types do not summarize strong updates to predicates of references provided as arguments. Such summaries could be useful, and should not be too difficult to add (particularly as the CC translation already uses a monad that preserves more information than the current source language types).

A promising direction for future work is to further explore the resource-like semantics of splitting references by rely and guarantee, and allowing *recovery* [105] of "stronger" references from the results of splitting, either based on combining provably equal references or by controlling scope. This would help control the gradual loss of precision in the current system when non-duplicable (non-self-splitting) references are repeatedly split to less precise relations. A resource-based approach to splitting, as in deny-guarantee program logics [73], suggests one promising approach to preserving possible actions.

A natural extension to rely-guarantee references is to exploit rely-guarantee reasoning (originally crafted for concurrent program verification) for concurrent programming via rely-guarantee references. Chapter 5 does exactly this.

4.7 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. These were each surveyed in Chapter 2, so here we point out only a few key relationships and technical distinctions.

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 [260], 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 [82]. 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 [73]. 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 from Chapter 3, 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. [269] apply a modularized rely-guarantee logic to treat (non-)interference in the degenerate case of sequential access to the UNIXv7 memory manager. Related systems [71, 260, 82] could be applied similarly, but to our knowledge haven't. Most have only first-order treatment of interference. Only Concurrent Abstract Predicates [71] 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 [105]).

Since the original presentation of this work [103], Militão et al. have proposed rely-guarantee protocols [174], a system for ensuring that actions through aliases cooperate in manipulating state correctly in a calculus of linear capabilities. Protocols for deep segments of the heap require substantial encoding, and the system is limited in expressiveness, with examples focusing primarily on typestate-like protocols. Our system is considerably more expressive due to the specification of rely/guarantee and predicate in higher order logic,

but this comes at the cost of requiring more substantial proof automation; any reasonable implementation of rely-guarantee references will be undecidable, while Militão et al. have a decidable type system (and working implementation thereof).

Local Rely-Guarantee [82] adapts rely guarantee reasoning for concurrent separation logic by lifting separating conjunction to relations as well, allowing successive splitting and joining of heap regions with different interference requirements. We do not support rejoining references, and concurrency is deferred until Chapter 5, but we support a form of overlapping but inequivalent regions, since the heap reachable from two references may overlap. This avoids the need for a framing invariant as present in LRG [82], which is required for the relational equivalent of precise assertions [258] (our notion of precision is baked into our interpretation of relations over the reachable heap fragment).

Many others have worked on integrating dependent types into imperative programming languages. Most take the approach of using refinement types [95, 243] that restrict modification to mutable data, but the refinements themselves may depend only on immutable data. Examples include DML [273], ATS [48], and X10's constrained types [198]. The refinement language is often also restricted to some theory that can be effectively decided by an SMT solver, as in Liquid Types [230]. 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) [188, 185, 190] and its implementation Ynot [189, 52]. 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 [52] 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 factored out a small set of combinators and transformers in hopes of some day

enabling 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 [139, 260, 82]). 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. Nanevski's implementations of HTT [189, 190] use binary postconditions rather than unary, giving some flavor of two-state specifications similar to rely-guarantee references.

VCC [57, 58] is a verification effort for concurrent C programs, using methodology derived directly from Spec# [18]. The main idea is to use two-state invariants for shared state, which must be preserved by any action. Objects are packed and unpacked as in Spec#, where upon re-packing the object's old (at unpacking) state and new state must satisfy the two-state invariant. They encode a sort of rely-guarantee-style reasoning using *claim objects*, whose existence ensures no other thread can have the shared object open. A claim is essentially an object whose invariant depends on other objects. Because multi-object invariants are permitted only when admissible (any action preserving the invariant of one object must not violate the invariant of the other, similar to assertion stability in rely-guarantee logics or rely containment and predicate stability in our system), a claim serves as a witness that no other part of the program is actively using the claimed objects. It is possible to encode some rely-guarantee reference style specification into VCC using claims to enable or disable different possible changes to an object, but the encoding would be quite heavyweight.

F*'s Dijkstra monad [245] (Section 2.2.2) includes an `iDST` variant with (transitive reflexive) two-state invariants over heaps, where a previous heap can be captured (in the logic) and witnessed as having a certain property. Later the fact that the previous and current heaps are related by the two-state invariant can be used to introduce new properties implied by the previous property and relation on heaps. F* also includes a reference type

constrained with a two-state single-cell relation, essentially equivalent to collapsing our rely and guarantee. However, it does not appear to be used to prove local preservation of invariants.

4.8 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.

More broadly, rely-guarantee references generalize the coarse-grained per-reference control of mutation present in reference immutability (Chapter 3) to richer properties, and more nuanced control over state modifications. Chapter 5 shows that rely-guarantee references can be adapted in an unintrusive manner to support concurrent programming, preserving the property of reference immutability that stepping from sequential to concurrent programming can be a small change given the right foundations.

$\boxed{\Gamma \vdash \tau \prec \tau \divideontimes \tau}$ $\dfrac{\tau \in \{\mathtt{nat}, \mathtt{bool}, \mathsf{unit}, \mathsf{Prop}, \mathsf{Type}, \mathsf{heap}, \_ = \_, \Pi x : \tau \to \tau', \tau \xrightarrow{\mathcal{M}} \tau'\}}{\Gamma \vdash \tau \prec \tau \divideontimes \tau}$

$\dfrac{\Gamma \vdash \tau \prec \tau_a \divideontimes \tau_b \qquad \Gamma \vdash \sigma \prec \sigma_a \divideontimes \sigma_b}{\Gamma \vdash (\tau, \sigma) \prec (\tau_a, \sigma_a) \divideontimes (\tau_b, \sigma_b)}$

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

$\boxed{\Gamma \vdash e : \tau}$ $\dfrac{}{\Gamma \vdash n : \mathtt{nat}}$ $\dfrac{}{\Gamma \vdash b : \mathtt{bool}}$ $\dfrac{}{\Gamma \vdash \mathsf{tt} : \mathsf{unit}}$ Axiom $\dfrac{}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}}$ V $\dfrac{\Gamma \vdash \tau \prec \tau \divideontimes \tau}{\Gamma, x : \tau \vdash x : \tau}$

$$\Pi\text{-I}\ \frac{\Gamma \vdash \tau \prec \tau \divideontimes \tau \qquad \forall \tau \in \Gamma. \vdash \tau \prec \tau \divideontimes \tau \qquad \Gamma \vdash (\Pi x : \tau \to \tau') : \sigma \qquad \sigma \in \{\mathsf{Type}, \mathsf{Prop}\} \qquad \Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.\, e) : \Pi x : \tau \to \tau'}$$

$$\Pi\text{-E}\ \frac{\Gamma \vdash e_1 : \Pi x : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'[x/e_2]} \qquad \frac{\epsilon \vdash \tau \qquad \epsilon \vdash \tau' \qquad x : \tau \vdash e : \tau' \Rightarrow \Gamma'}{\Gamma \vdash (\lambda_{\mathcal{M}} x : \tau.\, e) : \tau \xrightarrow{\mathcal{M}} \tau'}$$

Deref
$$\frac{\Gamma \vdash e : \mathsf{ref}\{\tau \mid P\}[R, G] \qquad \tau' = [R, G] \gg \tau \qquad \Gamma \vdash \tau \prec \tau \divideontimes \tau \qquad G\ \mathsf{reflexive}}{\Gamma \vdash !e : \tau'}$$

Π-F
$$\frac{\Gamma \vdash \tau : \gamma \qquad \Gamma \vdash \tau \prec \tau \divideontimes \tau \qquad \Gamma, x : \tau \vdash \tau' : \sigma \qquad \gamma, \sigma \in \{\mathsf{Type}, \mathsf{Prop}\}}{\Gamma \vdash \Pi x : \tau \to \tau' : \sigma}$$

Conv
$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash e : \tau'}$$

$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau}$

P-$\Rightarrow$
$$\frac{\forall v, h.\ P\ v\ h \to P'\ v\ h \qquad \mathsf{stable}\ P'\ R}{\Gamma \vdash \mathsf{ref}\{\tau \mid P\}[R, G] \rightsquigarrow \mathsf{ref}\{\tau \mid P'\}[R, G]}$$

R-$\subset$
$$\frac{[\![R]\!] \subseteq [\![R']\!] \qquad \mathsf{stable}\ P\ R'}{\Gamma \vdash \mathsf{ref}\{\tau \mid P\}[R, G] \rightsquigarrow \mathsf{ref}\{\tau \mid P\}[R', G]}$$

$$\text{G-}\subset\ \frac{[\![G']\!] \subseteq [\![G]\!]}{\Gamma \vdash \mathsf{ref}\{\tau \mid P\}[R, G] \rightsquigarrow \mathsf{ref}\{\tau \mid P\}[R, G']}$$

with metafunctions $hrel(\tau) \stackrel{\text{def}}{=} \tau \to \tau \to \mathsf{heap} \to \mathsf{heap} \to \mathsf{Prop}$ and $hprop(\tau) \stackrel{\text{def}}{=} \tau \to \mathsf{heap} \to \mathsf{Prop}$

Figure 4.4: Typing. For standard recursors for naturals, booleans, pairs, and identity types [129], as well as well-formed contexts, and (pure) expression/type conversion ($\Gamma \vdash \tau \rightsquigarrow \tau$), see (Figure 4.6).

$\boxed{\Gamma \vdash e : \tau \Rightarrow \Gamma}$

$$\text{V-}\emptyset\ \frac{}{\Gamma, x : \tau \vdash x : \tau \Rightarrow \Gamma} \qquad \text{V-}\divideontimes\ \frac{\Gamma \vdash \tau \prec \tau' \divideontimes \tau''}{\Gamma, x : \tau \vdash x : \tau' \Rightarrow \Gamma, x : \tau''}$$

$$\mathcal{M}\text{-I}\ \frac{\forall \tau \in \Gamma. \vdash \tau \prec \tau \divideontimes \tau \qquad \Gamma, x : \tau \vdash e : \tau' \Rightarrow \Gamma'}{\Gamma \vdash (\lambda_{\mathcal{M}}(x : \tau).\, e) : \tau \xrightarrow{\mathcal{M}} \tau' \Rightarrow \Gamma} \qquad \mathcal{M}\text{-E}\ \frac{\Gamma \vdash e_1 : \tau \xrightarrow{\mathcal{M}} \tau' \Rightarrow \Gamma_1 \qquad \Gamma_1 \vdash e_2 : \tau \Rightarrow \Gamma_2}{\Gamma \vdash e_1\ e_2 : \tau' \Rightarrow \Gamma_2}$$

$$\textsc{Write}\ \frac{\begin{array}{c}\Gamma \vdash e : \tau \Rightarrow \Gamma', x : \mathsf{ref}\{\tau \mid P\}[R, G] \\ \forall h, h' : heap.\, h' = [\![e]\!](h) \to P\ (!x)\ h' \to P'\ [\![e]\!]\ h'[x \mapsto [\![e]\!]] \\ \mathsf{stable}\ P'\ R \qquad \forall h, h' : heap.\, h' = [\![e]\!](h) \to P\ (!x)\ h' \to G\ (!x)\ e\ h'\ h'[x \mapsto [\![e]\!]]\end{array}}{\Gamma \vdash x \leftarrow e : \mathsf{unit} \Rightarrow \Gamma', x : \mathsf{ref}\{\tau \mid P'\}[R, G]}$$

$$\textsc{IDeref}\ \frac{\Gamma \vdash e : \mathsf{ref}\{\tau \mid P\}[R, G] \qquad \tau' = [R, G] \gg \tau \qquad \Gamma \vdash \tau \prec \tau \divideontimes \tau \qquad G\ \mathsf{reflexive}}{\Gamma \vdash !e : \tau' \Rightarrow \Gamma'} \qquad \textsc{Alloc}\ \frac{\Gamma \vdash e : \tau \Rightarrow \Gamma' \qquad \emptyset \subset G \qquad \mathsf{stable}\ P\ R \qquad \forall h : heap.\, P\ e\ h}{\Gamma \vdash \mathsf{alloc}\ e : \mathsf{ref}\{\tau \mid P\}[R, G] \Rightarrow \Gamma'}$$

$$\textsc{Swap}\ \frac{\Gamma \vdash x \leftarrow e : \mathsf{unit} \Rightarrow \Gamma', x : \mathsf{ref}\{\tau \mid P'\}[R, G]}{\Gamma \vdash \mathsf{swap}(x, e) : \tau \Rightarrow \Gamma', x : \mathsf{ref}\{\tau \mid P'\}[R, G]} \qquad \textsc{Pure}\ \frac{\epsilon \vdash e : \tau \qquad \epsilon \vdash \tau : \mathsf{Prop}}{\Gamma \vdash e : \tau \Rightarrow \Gamma}$$

$$\Pi\text{-E}\ \frac{\Gamma \vdash e_1 : \Pi x : \tau \to \tau' \Rightarrow \Gamma' \qquad \Gamma' \vdash e_2 : \tau \Rightarrow \Gamma'' \qquad x \notin FV(\tau')}{\Gamma \vdash e_1\ e_2 : \tau' \Rightarrow \Gamma''}$$

$\boxed{\Gamma \vdash \tau : \sigma}$

$$\textsc{wf-Ref}\ \frac{\begin{array}{c}\epsilon \vdash R : hrel(\tau) \qquad \epsilon \vdash G : hrel(\tau) \qquad \epsilon \vdash P : hprop(\tau) \qquad \mathsf{closed}\ R \\ \mathsf{precise}\ R \qquad \mathsf{precise}\ G \qquad \mathsf{precise}\ P \qquad \mathsf{stable}\ P\ R \qquad \epsilon \vdash \tau : \mathsf{Prop}\end{array}}{\Gamma \vdash \mathsf{ref}\{\tau \mid P\}[R, G] : \mathsf{Prop}}$$

Figure 4.5: Typing, continued.

$$\boxed{\Gamma \vdash \tau \rightsquigarrow \tau \text{ cont.}} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \rightsquigarrow \tau} \qquad \frac{\Gamma \vdash \tau \rightsquigarrow \tau' \qquad \Gamma \vdash \tau' \rightsquigarrow \tau''}{\Gamma \vdash \tau \rightsquigarrow \tau''} \qquad \frac{\Gamma \vdash \sigma \rightsquigarrow \tau \qquad \Gamma, x : \tau \vdash \tau' \rightsquigarrow \sigma'}{\Gamma \vdash \Pi x : \tau \to \tau' \rightsquigarrow \Pi x : \sigma \to \sigma'}$$

$$\frac{\Gamma \vdash \sigma \rightsquigarrow \tau}{\Gamma \vdash (\lambda x : \tau . e) \rightsquigarrow (\lambda x : \sigma . e)} \qquad \frac{\tau =_{\beta v} \tau'}{\Gamma \vdash \tau \rightsquigarrow \tau'} \qquad \boxed{\vdash \Gamma} \qquad \frac{}{\vdash \epsilon} \qquad \frac{\vdash \Gamma \qquad x \notin \Gamma \qquad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}$$

$$\boxed{\Gamma \vdash \tau : \sigma \text{ cont.}} \qquad \frac{\tau \in \{\mathtt{nat}, \mathtt{bool}, \mathsf{unit}, \mathsf{True}, \mathsf{False}\}}{\Gamma \vdash \tau : \mathsf{Prop}} \qquad \frac{\Gamma \vdash \tau : \gamma \qquad \Gamma \vdash \sigma : \gamma}{\Gamma \vdash (\tau, \sigma) : \gamma}$$

$$\frac{\Gamma \vdash \tau : \mathsf{Prop} \qquad \Gamma \vdash \tau' : \mathsf{Prop}}{\Gamma \vdash \tau \overset{\mathcal{M}}{\to} \tau' : \mathsf{Prop}} \qquad \frac{\Gamma \vdash A : \sigma \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}(a) : a = a} \qquad \frac{\Gamma \vdash \tau \prec \tau \circledast \tau}{\Gamma \vdash =_\tau : \tau \to \tau \to \mathsf{Prop}} \qquad \frac{}{\Gamma \vdash \mathsf{truth} : \mathsf{True}}$$

$$\frac{\Gamma \vdash u : \mathsf{unit} \qquad \Gamma \vdash d : C[x/\mathsf{tt}] \qquad \Gamma, x : \mathsf{unit} \vdash C : \sigma \qquad \sigma \in \{\mathsf{Prop}, \mathsf{Type}\}}{\Gamma \vdash \mathcal{R}_{\mathsf{unit}}(u, d) : C[x/u]} \qquad \frac{\Gamma \vdash p : (\tau * \sigma)}{\Gamma \vdash \mathsf{fst}\ p : \tau} \qquad \frac{\Gamma \vdash p : (\tau * \sigma)}{\Gamma \vdash \mathsf{snd}\ p : \sigma}$$

$$\frac{\Gamma \vdash h : \mathsf{heap} \qquad \Gamma \vdash e : \mathsf{ref}\{A \mid P\}[R, G]}{\Gamma \vdash h[e] : A} \qquad \frac{\Gamma \vdash b : \mathsf{False} \qquad \Gamma, x : \mathsf{False} \vdash C : \sigma \qquad \sigma \in \{\mathsf{Prop}, \mathsf{Type}\}}{\Gamma \vdash \mathcal{R}_{\mathsf{False}}(b) : C[x/b]}$$

$$\frac{\begin{array}{c}\Gamma \vdash n : \mathtt{nat} \\ \Gamma \vdash c : C[x/0] \qquad \Gamma \vdash f : \Pi n : \mathtt{nat} \to (C[x/n] \to C[x/S\,n]) \qquad \Gamma, x : \mathtt{nat} \vdash C : \sigma \qquad \sigma \in \{\mathsf{Prop}, \mathsf{Type}\}\end{array}}{\Gamma \vdash \mathcal{R}_{\mathtt{nat}}(n, c, f) : C[x/n]}$$

$$\frac{\Gamma \vdash b : \mathtt{bool} \qquad \Gamma \vdash t : C[x/\mathsf{true}] \qquad \Gamma \vdash f : C[x/\mathsf{false}] \qquad \Gamma, x : \mathtt{bool} \vdash C : \sigma \qquad \sigma \in \{\mathsf{Prop}, \mathsf{Type}\}}{\Gamma \vdash \mathcal{R}_{\mathtt{bool}}(b, t, f) : C[x/b]}$$

$$\frac{\Gamma \vdash b : \mathsf{True} \qquad \Gamma \vdash t : C[x/\mathsf{truth}] \qquad \Gamma, x : \mathsf{True} \vdash C : \sigma \qquad \sigma \in \{\mathsf{Prop}, \mathsf{Type}\}}{\Gamma \vdash \mathcal{R}_{\mathsf{True}}(b, t) : C[x/b]}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \sigma \qquad \Gamma \vdash c : a =_A b \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash d : C[x/a, y/a, z/\mathsf{refl}(a)] \qquad \Gamma, x : A, y : A, z : x = y \vdash C : \mathsf{Prop}\end{array}}{\Gamma \vdash \mathcal{R}_{=}(c, d) : C[x/a, y/b, z/c]}$$

For auxilliary function definitions, see Figure 4.7.

Figure 4.6: Selected auxiliary judgments

$$\begin{array}{rcl}
\mathsf{stable}_\tau\ (P : hprop(\tau))\ (R : hrel(\tau)) & \stackrel{\text{def}}{=} & \forall x : \tau.\, \forall h, h' : \mathsf{heap}.\, P\ x\ h \to R\ x\ h\ h' \to P\ x\ h' \\
\mathsf{precise}_\tau\ (P : hpred(\tau)) & \stackrel{\text{def}}{=} & \forall x, h, h'.\, (\forall l, \mathsf{ReachFromIn}\ l\ x\ h \to h[l] = h'[l]) \to P\ x\ h \to P\ x\ h' \\
\mathsf{precise}_\tau\ (R : hrel(\tau)) & \stackrel{\text{def}}{=} & \forall x, x', h, h', \overline{h}, \overline{h'}.\, (\forall l, \mathsf{ReachFromIn}\ l\ x\ h \to h[l] = h'[l]) \to \\
& & \quad (\forall l, \mathsf{ReachFromIn}\ l\ x'\ \overline{h} \to \overline{h}[l] = \overline{h'}[l]) \to \\
& & \quad R\ x\ x'\ h\ \overline{h} \to R\ x\ x'\ h'\ \overline{h'}
\end{array}$$

$$\mathsf{prims} \stackrel{\text{def}}{=} \{\mathtt{nat}, \mathtt{bool}, \mathsf{unit}, \Pi x : \tau' \to \tau'', \tau \xrightarrow{\mathcal{M}} \tau', \mathsf{True}, \mathsf{False}, \mathsf{heap}, \_ = \_\}$$

$$\mathsf{closed}_\tau\ (R : hrel(\tau)) \stackrel{\text{def}}{=} \begin{cases} \mathsf{True} & \text{if } \tau \in \mathsf{prims} \\ \mathsf{closed}\ R.1 \wedge \mathsf{closed}\ R.2 & \text{if } \tau = (\sigma * \sigma') \\ \mathsf{closed}\ R' \wedge \forall l, h, h'.\, R'\ h[l]\ h'[l]\ h\ h' \to R\ l\ l\ h\ h' & \text{if } \tau = \mathsf{ref}\{\tau' \mid P'\}[R', G'] \end{cases}$$

$$[R, G] \gg \tau \stackrel{\text{def}}{=} \begin{cases} \tau & \text{if } \tau \in \mathsf{prims} \\ ([R.1, G.1] \gg \sigma * [R.2, G.2] \gg \gamma) & \text{if } \tau = (\sigma * \gamma) \\ \mathsf{ref}\{\sigma \mid P\}[R', \begin{array}{l}(\lambda a, a', h, h'.\, G'\ a\ a'\ h\ h' \wedge \\ \quad (\forall l, h[l] = a \to h'[l] = a' \to G\ l\ l\ h\ h'))\end{array}] & \text{if } \tau = \mathsf{ref}\{\sigma \mid P\}[R', G'] \end{cases}$$

where for $R : hrel(\sigma * \sigma')$:

$$R.1 \stackrel{\text{def}}{=} \lambda x, x' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall y : \sigma'.\, R\ (x, y)\ (x', y)\ h\ h'$$

$$R.2 \stackrel{\text{def}}{=} \lambda y, y' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall x : \sigma'.\, R\ (x, y)\ (x, y')\ h\ h'$$

Figure 4.7: Auxilliary functions

$$\begin{array}{rl}
 & (\lambda r : \mathsf{ref}\{\mathtt{nat} \mid \ldots\}[\ldots, \ldots].\, (\lambda pf :!r =!r.\, (\lambda u : \mathsf{unit}.\, (\lambda pf2 :!r =!r.\, !r)\mathcal{R}_{\mathtt{bool}}(\mathsf{true}, pf, \mathsf{refl}(!r))))\ \mathsf{refl}(!r))\ \ell \\
\to & (\lambda pf :!\ell =!\ell.\, (\lambda u : \mathsf{unit}.\, (\lambda pf2 :!\ell =!\ell.\, !\ell)\mathcal{R}_{\mathtt{bool}}(\mathsf{true}, pf, \mathsf{refl}(!\ell))))\ \mathsf{refl}(!\ell) \\
\to & (\lambda pf :!\ell =!\ell.\, (\lambda u : \mathsf{unit}.\, (\lambda pf2 :!\ell =!\ell.\, !\ell)\mathcal{R}_{\mathtt{bool}}(\mathsf{true}, pf, \mathsf{refl}(!\ell))))\ \mathsf{refl}(\langle !\ell \rangle 3)
\end{array}$$

Figure 4.8: The simplest term we can imagine that observes the multiple-dereference reduction

$$e ::= \ldots \mid \langle !\ell \rangle e \mid \mathsf{fold}\ e\ e\ e$$

$$v ::= \mathsf{true} \mid \mathsf{tt} \mid n \mid b \mid \mathsf{Type} \mid \mathsf{Prop} \mid \Pi x : \tau \to \tau' \mid \lambda x : \tau.\, e \text{ (no annotations in } e) \mid (\lambda_{\mathcal{M}} x : \tau.\, e) \text{ (where } FV(e) = \emptyset)$$

$$H : \mathsf{loc} \rightharpoonup v \qquad \Sigma : \mathsf{loc} \rightharpoonup \tau$$

$$\boxed{H; \Sigma; \Gamma \vdash M : N \text{ (extends } \Gamma \vdash M : N)}$$

$$\frac{H; \Sigma; \Gamma \vdash !\ell : \tau' \qquad H(\ell) = v \qquad \epsilon \vdash v : \tau \qquad [R, G] \gg \tau = \tau' \qquad v' =_\beta (\mathsf{fold}\ R\ G\ v)}{H; \Sigma; \Gamma \vdash \langle !\ell_{\tau,P,R,G} \rangle v' : \tau'} \qquad \frac{\Sigma(\ell) = \tau}{H; \Sigma; \Gamma \vdash \ell_{\tau,P,R,G} : \mathsf{ref}\{\tau \mid P\}[R, G]}$$

$$\boxed{H; \Gamma \vdash M \rightsquigarrow N \text{ (extends } \Gamma \vdash M \rightsquigarrow N)} \qquad \frac{H(\ell) = v \qquad v' =_\beta (\mathsf{fold}\ R\ G\ v)}{H; \Gamma \vdash \langle !\ell_{\tau,P,R,G} \rangle v' \rightsquigarrow !\ell_{\tau,P,R,G}}$$

$$\boxed{H; e \to e'} \qquad \frac{}{H; (\lambda x : \tau.\, e)\ v \to e[x/v]}\ \beta\lambda \qquad \frac{H(\ell) = v}{H;\ !\,\ell_{\tau,P,R,G} \to \langle !\,\ell_{\tau,P,R,G} \rangle (\mathsf{fold}\ R\ G\ v)}$$

$$\frac{H;\ e \to e'}{H;\ \langle !\,\ell_{\tau,P,R,G} \rangle e \to \langle !\,\ell_{\tau,P,R,G} \rangle e'} \qquad \frac{v \in \{\mathtt{nat}, \mathtt{bool}, \mathsf{unit}, (\lambda x.\, e), (\lambda_{\mathcal{M}} x.\, e), \mathsf{true}, \mathsf{refl}(\_)\}}{H;\ (\mathsf{fold}\ R\ G\ v) \to v}$$

$$\frac{}{H;\ (\mathsf{fold}\ R\ G\ (v_1, v_2) \to (\mathsf{fold\ havoc}\ G.1\ v_1, \mathsf{fold\ havoc}\ G.2\ v_2)}$$

$$\frac{}{H;\ (\mathsf{fold}\ R\ G\ (\ell_{\tau,P,R',G'})) \to \ell_{\tau,P,R',(\lambda a, a', h, h'.\, G'\ a\ a'\ h\ h' \wedge (\forall l.\, h[l] = a \to h'[l] = a' \to G\ l\ l\ h\ h'))}}$$

$$\boxed{S; H; e \to S'; H'; e'} \qquad \frac{H; e \to e' \qquad \langle !\ell \rangle a \in e' \qquad e'' = \mathsf{heap\_indep}(e')}{S; H; \mathsf{pure}\ e \to S; H; \mathsf{pure}\ e''[\langle !\ell \rangle a/a]} \qquad \frac{H; e \to e' \qquad \forall \ell, a.\, \langle !\ell \rangle a \notin e'}{S; H; \mathsf{pure}\ e \to S; H; \mathsf{pure}\ e'}$$

$$\frac{}{S; H; \mathsf{pure}\ v \to S; H; v} \qquad \frac{y \in FV(e)}{S; H; (\lambda_{\mathcal{M}} x : \tau.\, e) \to S; H; (\lambda_{\mathcal{M}} x : \tau.\, e[y/S(y)])}$$

$$\frac{FV(e) = \emptyset \qquad y \text{ fresh}}{S; H; (\lambda_{\mathcal{M}} x : \tau.\, e)\ v \to S[y \mapsto v]; H; e[x/y]} \qquad \frac{S(x) = \ell \qquad \ell \in \mathsf{dom}(H)}{S; H;\ x \leftarrow v\ \to S; H[\ell \mapsto v]; \mathsf{tt}}$$

$$\frac{S(x) = \ell \qquad H(\ell) = v'}{S; H; \mathsf{swap}(x, v) \to S; H[\ell \mapsto v]; v'} \qquad \frac{\ell \notin \mathsf{dom}(H)}{S; H; \mathsf{alloc}\ v \to S; H[\ell \mapsto v]; \mathsf{tt}} \qquad \frac{}{S; H; \mathsf{consume}\ x \to S \setminus x; H; S(x)}$$

$$\frac{S(x) = v}{S; H; \mathsf{split}\ x\ \tau\ \tau' \to S[x \mapsto v\ \mathsf{as}\ \tau]; H; v\ \mathsf{as}\ \tau'} \qquad \frac{}{S; H; \mathsf{pureApp}\ (\lambda x : \tau.\, e)\ v \to S; H; \mathsf{pure}\ ((\lambda x : \tau.\, e)\ v)}$$

$$\frac{H(\ell) = v}{S; H;\ !\,\ell_{\tau,P,R,G} \to S; H; \mathsf{pure}\ \langle !\,\ell_{\tau,P,R,G} \rangle (\mathsf{fold}\ R\ G\ v)}$$

Figure 4.9: Values, typing of runtime values, and main operational semantics for RGref.

$\boxed{H; e \to e'}$

$$\frac{H; e_1 \to e_1'}{H; e_1\ e_2 \to e_1'\ e_2} \qquad \frac{H; e_2 \to e_2'}{H; v\ e_2 \to v\ e_2'} \qquad \frac{H; e_1 \to e_1'}{H; (e_1, e_2) \to (e_1', e_2)}$$

$$\frac{H; e_2 \to e_2'}{H; (v, e_2) \to (v, e_2')} \qquad \frac{H; e \to e'}{H; !\, e \to !\, e'} \qquad \frac{H; e \to e'}{H; \mathsf{refl}\ e \to \mathsf{refl}\ e'}$$

$\boxed{S; H; e \to S'; H'; e'}$

$$\frac{S; H; e_1 \to S'; H'; e_1'}{S; H; e_1\ e_2 \to S'; H'; e_1'\ e_2} \qquad \frac{S; H; e_2 \to S'; H'; e_2'}{S; H; v\ e_2 \to S'; H'; v\ e_2'}$$

$$\frac{S; H; e \to S'; H'; e'}{S; H; x \leftarrow e \to S'; H'; x \leftarrow e'} \qquad \frac{S; H; e \to S'; H'; e'}{S; H; \mathsf{swap}(x, e) \to S'; H'; \mathsf{swap}(x, e')}$$

$$\frac{S; H; e \to S'; H'; e'}{S; H; \mathsf{alloc}\ e \to S'; H'; \mathsf{alloc}\ e'} \qquad \frac{S; H; e_1 \to S'; H'; e_1'}{S; H; \mathsf{pureApp}\ e_1\ e_2 \to S'; H'; \mathsf{pureApp}\ e_1'\ e_2}$$

$$\frac{S; H; e_2 \to S'; H'; e_2'}{S; H; \mathsf{pureApp}\ v\ e_2 \to S'; H'; \mathsf{pureApp}\ v\ e_2'}$$

Figure 4.10: Structural / context operational semantics for RGref.

$$\boxed{\vdash S; H; e : \Gamma; \Sigma; \tau \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\vdash \Sigma \\ \vdash H : \Sigma \qquad \vdash \Gamma \qquad H; \Sigma \vdash S : \Gamma \qquad H; \Sigma; \Gamma \vdash e : \tau \Rightarrow \Gamma' \qquad \forall \tau, \ell, P, R, G.\, \ell_{\tau,P,R,G} \in S; H; e \implies [\![P]\!]\, H(\ell)\, H \\ \forall \tau, \ell, P, P', R, R', G, G'. \left(\begin{array}{l} \ell_{\tau,P,R,G} \in S; H; e \wedge \\ \ell_{\tau,P',R',G'} \in S; H; e \wedge \\ \text{the references are in different locations} \Rightarrow \\ (G \implies R' \wedge G' \implies R) \end{array} \right)\end{array}}{\vdash S; H; e : \Gamma; \Sigma; \tau \Rightarrow \Gamma'}$$

$$\boxed{H; \Sigma \vdash S : \Gamma} \qquad \frac{}{H; \Sigma \vdash S : \epsilon} \qquad \frac{H; \Sigma \vdash S/x : \Gamma \qquad H; \Sigma; \epsilon \vdash S(x) : \tau}{H; \Sigma \vdash S : \Gamma, x : \tau}$$

$$\boxed{\vdash \Sigma} \qquad \frac{\forall \ell \in \mathsf{dom}(\Sigma).\, \emptyset; \emptyset; \epsilon \vdash \Sigma(\ell) : \mathsf{Prop}}{\vdash \Sigma} \qquad \boxed{\vdash H : \Sigma}$$

$$\frac{\mathsf{dom}(H) = \mathsf{dom}(\Sigma) \qquad \forall \ell \in \mathsf{dom}(H).\, \emptyset; \Sigma; \epsilon \vdash H(\ell) : \Sigma(\ell)}{\vdash H : \Sigma}$$

Figure 4.11: Dynamic state typing.

Chapter 5

CONCURRENT REFINEMENT TYPES AND FUNCTIONAL CORRECTNESS VIA RELY-GUARANTEE REFERENCES

> ... the effect of executing $S_1, \ldots, S_n$ in parallel is the same as executing each one by itself, provided the processes don't "interfere" with each other. The key word is of course "interfere."
>
> Owicki and Gries, *An Axiomatic Proof Technique for Parallel Programs I*, 1976 [204]

Our development of RGrefs in Chapter 4 laid the foundations for precisely reasoning about interference between aliases, and exploited this for proving invariants and step-invariants in sequential programs. But this omits two critical goals for program verification: reasoning about concurrent programs, and proving functional correctness. This chapter extends RGrefs cleanly for both purposes.

RGrefs are built on rely-guarantee reasoning [139], a sound reasoning principle for concurrent programs. So it seems reasonable that RGrefs as presented in Chapter 4 are at least close to sound for shared-memory concurrent programs. Furthermore, such an adaptation would be very desirable: the Microsoft team using the prototype C# extension found a gradual refinement from C# to the prototype useful, and the gradual refinement of sequential reference immutability to RGrefs is a natural extension. Adding a progressive refinement to concurrency-safe RGrefs would then offer a single programming model where stepping from sequential unverified code to verified concurrent code is a matter of only a few incremental passes refining the code.

But the previous presentation is unsound for concurrency, for several reasons:

- When proving properties of an expression, all dereferences in the expression are treated

as if they occur in the same heap. But in some cases, such as

$$(\lambda y.\, \lambda z.\, (y, !x))(!x)$$

this is unsound. Treating the dereferences of x as equal allows concluding that the resulting closure produces identical pairs, while a write through x after this closure's creation may produce different results. This further exacerbates type preservation issues as well, since even $(!x, !x)$ may not evaluate to an identical pair, which means the type of refl $(\texttt{nat} \times \texttt{nat})$ $(!x, !x)$ will not be preserved.

- When proving that a write satisfies a guarantee, a dereference of the stored-to location in the stored expression is treated as equivalent to the old value; under fine-grained interleaving this may not be the case, similar to above.

Both of these issues can be addressed with small modifications to the underlying language:

- Restricting reasoning for pure terms that may dynamically perform multiple dereferences

- Weakening atomicity assumptions for heap accesses (compensated by adding new primitives such as compare-and-swap, whose dynamic atomicity properties reintroduce many atomic assumptions)

This means that as in the reference immutability case, given the right means to restrict mutation in a sequential program, transitioning to concurrent programming need only be a modest change.

Another class of verification not yet explored in this thesis is functional correctness. Chapter 3 proves only data race freedom, and Chapter 4 (plus the changes listed above and described below) proves only logical and two-state invariants. While the core ideas are simple, the type system is already quite technical, and it would be undesirable to add significant additional machinery to the type system to support functional correctness. Fortunately, we do not have to. RGREFs already embed sufficient information in the type system to prove functional correctness. So our extensions for functional correctness will be

only additional processing of the typing derivation, rather than adding further requirements to the type-checking process.

5.1 Correctness for Lock-Free Datastructures

The key challenge in proving correctness of fine-grained concurrent data structures (FCDs following recent terminology [254]) is the treatment of interference from other threads. One of the significant obstacles to such reasoning is that it typically requires introducing a host of new concepts absent from single-threaded verification (e.g., locks, or atomic regions). Though historically explicit interference was confined to concurrency reasoning, recent work on reference-based characterizations of interference [105, 103] (Chapter 3 and 4) shows that interference summaries are sensible in the sequential setting as well. This suggests a single programming model with only modest differences between serial and concurrent programming.

This chapter extends Chapter 4 to prove both safety properties and functional correctness for fine-grained concurrent data structures, specifically lock-free data structures. The original formulation of RGREFs made strong atomicity assumptions to ease reasoning, but these assumptions do not hold for concurrent programs. By carefully weakening these assumptions to reflect shared-memory concurrent execution,

- We provide the first refinement types [95, 273] for shared-memory concurrent heap structures.

- We show how to prove functional correctness for programs using RGREFs by synthesizing abstract execution traces suitable for specification refinement (a general technique).

- We provide mechanized proofs of safety properties for classic FCDs [251, 170] and the first mechanized proof of safety properties for a lock-free linearizable union-find [7].

- We implement a COQ DSL for concurrent RGREFs.

- We provide a new soundness proof for sequential and concurrent RGrefs based on the Views Framework [70], providing a new view on the meaning of rely-guarantee references.

A key advantage of the RGref approach to preserving invariants is that unlike related work [255, 254, 215, 137], it does not assume the set of operations on a data structure is closed: rather than verifying a collection of operations together and closing the set (e.g., using abstraction to create modules [255, 254, 253, 71, 242]), developers instead specify for each structure the assumptions about how memory may be used by each reference, and ensure that all operations on a FCD obey those behavioral constraints. This is a key advantage for extensibility: abstraction boundaries are not required to precisely match verification boundaries. In particular, existing type-safe but otherwise unrestricted code — which we will call, for lack of a better term, unspecified — continues to type-check in an RGref system. An additional advantage of RGrefs for concurrent programming is that the difference between the sequential and concurrent variants is modest: we are reusing sequential verification constructs for concurrent programs, rather than retrofitting support for thread interference.

Safety properties, however, are only half the story for FCDs. Functional correctness is also important, and we show that the semantic checks performed by RGrefs already include enough information to prove functional correctness for some lock-free data structures [251, 170]. We show how to synthesize an abstraction of all execution traces for an operation from its typing derivation, and can then prove these abstract traces refine a relational specification.

We implemented the type system and refinement approaches in a Coq DSL, which we have used to prove invariants for four lock-free data structures, and specification refinement for two of those. Our implementation shows feasibility of the approach, and we have used it for new results: we give the first mechanized proofs of invariants for a lock-free linearizable union-find implementation [7].

5.2 *Suitability of* RGrefs *for Concurrent Programming*

This section recalls 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. The original system is unsound for concurrency and can only prove invariants, but this chapter fixes the unsoundness and extends the system for proving functional correctness.

Strengths RGREFs subsume unspecified code: an RGREF whose rely, guarantee, and predicate impose no constraints is equivalent to a run-of-the-mill ML-style reference:

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

Most verification systems cannot use unspecified code without substantial conversion work. For example, most program logics can only assign the judgment $\vdash \{\, P \,\}\, C\, \{\, \mathsf{True} \,\}$ to unspecified code because there is no way to restrict how state is modified without also giving a precise postcondition, requiring strong verification to even be invocable. Thus most program logics cannot even hand off disposable state to unspecified code and recover useful values. Meanwhile unspecified code already type-checks with our enriched reference types, using only the naïve translation above. Inappropriate attempts to write through a restricted reference simply fail to typecheck. So interactions in both directions can be checked for safety, though little is known about the results of unspecified code beyond its basic type. This is important because it means 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 concurrent data structure using RGREFs can be verified safe in the context of a larger program without verifying the whole program.

Another advantage of RGREFs for concurrency is that correctly enforcing state change protocols encoded in rely and guarantee relations does not *require* abstraction. (We exploit this in Section 5.3.1.) The monotonically increasing counter above was proposed by

Pilkiewicz and Pottier [215] as a verification challenge, because it requires proving temporal properties of how a piece of memory is used, rather than the approach taken by most verification systems: characterizing the behavior of code on a given section of memory. Most solutions require creating modules that abstract the type of the counter [215, 255, 137, 254, 253, 71, 242, 145] to ensure non-interference from other program components. By contrast, 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. Pilkiewicz and Pottier's solution [215] and Jensen and Birkedal's solution [137] also ensure functional correctness of increment, relying on a sealed module to constrain interference. The RGREF counter from Section 4.3.1 ensures 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.

Finally, the RGREF specification style is a natural fit for fine-grained concurrent data structures. Rely-guarantee reasoning itself has long established its utility for concurrent programs. RGREFs in particular allow encoding some forms of protocols similar to recent work [254, 253]. For example, O'Hearn et al. noted that some very general lemmas about traversing lock-free singly-linked list structures are provable assuming the list links and node deletion proceed in a particular manner, expressed as a set of invariants and two-state step invariants on lists and nodes [201][1]. Most of these are naturally expressible as predicates and rely-guarantee relations, with the remaining couple easy to reformulate.

Disadvantages The chief limitation of RGREFs for concurrent programming is the strong atomicity assumptions due to the original design's sequential setting. This chapter makes RGREFs sound for concurrent programs. The original RGREFs are also limited to proving invariants and enforcing some protocols. We extend RGREFs for proving functional correctness.

[1]This chapter's system cannot verify the algorithm from their paper because O'Hearn's code assumes atomic writes to multiple discontiguous pieces of memory.

There are other limitations which are less severe, but may make it difficult to express certain sophisticated concurrent algorithms. We leave lifting these limitations to future work (Section 5.7) because they are less pressing and are not specific to concurrent programming. RGrefs cannot relate reference-centric specifications to other granularities of reasoning (e.g., region-based rely-guarantee [82]). This makes it difficult to relate structures without references between them, such as connecting the auxiliary structures used in lock-free algorithms with helping (e.g., an elimination stack). RGrefs also cannot represent asymmetry of knowledge (e.g., that a lock is held), or reason about the combination of *multiple* writes satisfying a guarantee. 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. With the exception of the region granularity, we believe future work (Section 5.7) could address these shortcomings using known approaches.

5.3 RGrefs *for Concurrency*

Rely-guarantee references already reflect a sound concurrent reasoning principle, but the original formulation [103] (Chapter 4) is unsuitable for use in a concurrent setting as might be inspired by uses of reference immutability for safe concurrency [105] (Chapter 3)[2]. To make it safe, we must alter the system to address two key issues: granularity of reasoning (the primary change), and read/write atomicity. We also restrict terms stored on the heap and stack to prohibit unevaluated dereference expressions. This is an additional restriction over the original version, but is not fundamental, and only serves to simplify our metatheory.

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 [103] (Chapter 4) considered e atomically because it was proven sound for the sequential setting. The original design even compared e atomically to $!x$, while still permitting dereferences in e. This is part of the power of the approach ($\forall h.\, \mathsf{inc}\ (!x)\ (!x+1)\ h\ h[x \mapsto \ldots]$ is straightforward to prove in a dependent

[2]See Section 5.3.4 for the relationship between Chapter 3's concurrency and this chapter's.

type theory). But when other threads' writes interleave between heap accesses (e.g., between the read of x in the increment and the storage of the new value) this comparison is unsafe for two reasons. First, interleaving writes break type soundness for expressions, because the expression language is dependently typed and includes dereference.[3] Second, expressions can no longer be compared atomically if they contain dereferences, since they are potentially evaluated in different heaps at run time. (This makes obligations like the inc example above marginally more complex to state and prove.)

To address granularity issues, we limit expressions to at most one dereference (fixing type soundness for expressions), and then restrict the rules for proving guarantees. Equational reasoning between locations and stored expressions (when proving guarantees) is permitted only for non-heap-accessing terms (e.g., the old and new values in a compare-and-swap). Otherwise stored values are made opaque, and richer information can be carried into a proof only by strong RGREFs that are converted to possibly-weaker references upon storage, e.g., to reason about sharing a previously-thread-local data structure node. This is a fundamental restriction to make thread-shared RGREFs concurrency-safe.

Read/Write Atomicity A less onerous restriction is reasoning about atomically-executable reads, which typically correspond to data types with machine-word-sized runtime representations (e.g., a pointer, machine-register-width integer, etc.). We treat this with an Atomic predicate on types indicating those that are machine-atomic. Our implementation includes support for fields, including compare-and-swap on a field. We use this in examples (Section 5.3.1), but omit fields from the formal treatment in Section 5.3.2.

5.3.1 Examples

We have used our system to verify invariants for an atomic counter (Section 5.3.1), a Treiber stack [251] (Section 5.3.1), a lock-free linearizable union-find implementation due to Anderson and Woll [7] (Section 5.3.1), and a tail-less Michael-Scott queue [170] (see our implementation; Section 5.5). Table 5.1 gives the lines of "specification" (code and lemma statements) and proof for our examples, as reported by COQ's `coqwc` tool, which gives a rough estimate

[3] E.g., the term refl($!x$) may have type $3 = 3$ in one heap, and type $4 = 4$ after an interleaved write.

Program	Lines of Code	Lines of Proof
Atomic Increment	7	12
Treiber Stack	72	47
Michael-Scott Queue	123	125
Union-Find	392	1171

Table 5.1: Lines of code and proof, per Coq's line count utility.

of the proof burden relative to code size. 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.

Atomic Counter

As a simple example, consider Figure 5.2's atomic extension to the monotonically increasing counter from Figure 5.1, a self-contained presentation of the example from Section 4.3.1. As suggested above, the type judgment for CAS (see Section 5.3.2) does not permit any dereferences in its nested expressions, which justifies unrestricted equational reasoning when discharging the proof obligation

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

Thus we have proven that when the CAS succeeds, it obeys the counter's guarantee. The CAS guarantee obligation introduces information refined by the success case: when the guarantee needs to be proven, it is known that the heap cell at c contains x, without knowing that x was derived by dereferencing c in a previous heap.

Treiber Stack

Figure 5.3 gives the code for Treiber's lock-free stack [251] using concurrent RGrefs to prove that the push and pop operations add or remove at most one node at a time. The stack (`ts`) is a reference to an option of a reference to an immutable `Node`, updated according to relation `deltaTS` (which permits single-node pushes and pops).

The push operation satisfies the `ts_push` case of `deltaTS`. Proving this relies on the

```
Require Import Coq.Arith.Arith.
Require Import RGref.DSL.DSL.
(** * A Strictly Positive Monotonic Counter *)
Definition increasing : hrel nat := (fun n n' h h' => n <= n').
Definition pos : hpred nat := (fun n h => n > 0).
(** Now the definition of a monotonically increasing counter is
    barely more work than a completely unchecked counter. *)
Program Definition posmonotonic_counter :=
    ref{nat|pos}[increasing,increasing].
Program Definition read_counter (c:posmonotonic_counter) : nat := !c.
Program Definition inc_monotonic { μ} (p:posmonotonic_counter)
    : rgref μunit μ:= [p]:= !p + 1.
Program Definition mkCounter { μ} (u:unit)
    : rgref μposmonotonic_counter μ:= Alloc 1.
Program Example test_counter { μ} (u:unit) : rgref μunit μ:=
  x <- mkCounter tt;
  inc_monotonic x.
```

Figure 5.1: A positive monotonically increasing counter using RGREFs, in our COQ DSL. The `rgref` monad for imperative code is indexed by input and output environments Δ of substructural values, not used in this example.

```
Program Definition atom_inc (c:monotonic_counter) :=
    RGFix _ _ (fun retry _ =>
        let x := !c in
        done <- CAS(c, x, x+1);
        if done then rgret tt else retry tt) tt.
```

Figure 5.2: Atomic increment for a monotonically increasing counter. This code uses a standard read-modify-CAS loop. Until the increment succeeds, it reads the counter's value, then tries to compare-and-swap the old value with the incremented result. `RGFix` is a fixpoint combinator, and `rgret` returns a value.

```
(** * Trieber Stack: A lock-free stack implementation. *)
(** ** Basic heap structure, and rely/guarantee interactions *)
Inductive Node : Set :=
  mkNode : nat -> option (ref{Node|any}[local_imm,local_imm]) -> Node.
Inductive deltaTS : hrel (option (ref{Node|any}[local_imm,local_imm])) :=
  | ts_nop : forall n h h', deltaTS n n h h'
  | ts_push : forall n hd hd' h h', h'[hd']=(mkNode n hd) -> deltaTS hd (Some hd') h h'
  | ts_pop : forall n hd hd' h h', h[hd]=(mkNode n hd') -> deltaTS (Some hd) hd' h h'.
Definition ts :=
  ref{option (ref{Node|any}[local_imm,local_imm])|any}[deltaTS,deltaTS].
(** Allocation *)
Program Definition alloc_ts {μ} (u:unit) : rgref μts μ:= Alloc None.
(** Push operation *)
Program Definition push_ts {μ} : ts -> nat -> rgref μunit μ:=
  RGFix2 _ _ _ (fun rec s n =>
    let tl := !s in
    (* 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 rgret tt else rec s n).
(** Pop operation *)
Program Definition pop_ts {μ} : ts -> rgref μ(option nat) μ:=
  RGFix _ _ (fun rec s =>
    match !s with
    | None => rgret None
    | Some hd =>
      observe-field hd --> val as n in (fun a h => (getF val a)=n);
      (* Γ = ..., n : nat, hd : ref{Node | λx, h. getF val x = n}[local_imm, local_imm] *)
      observe-field hd --> nxt as tl in (fun a h => (getF nxt a)=tl);
      (* Γ = ..., tl : ..., hd : ref{Node | λx, h. x = mkNode n tl}[local_imm, local_imm] *)
      success <- CAS(s,Some hd,tl');
      if success then rgret (Some n) else rec s
    end).
```

Figure 5.3: A Treiber Stack [251] using RGREFs, omitting proofs. The relation `local_imm` (not shown) constrains the immediate referent to immutable; `any` is the always-true predicate.

strong initial refinement when allocating the new head as an immutable node: λ `x h` $\rightarrow$ `x = mkNode n tl`. The CAS update satisfies the guarantee because under the assumption that the head of the stack is `tl` at the time of the write (an assumption the CAS rule uses to characterize its conditional behavior; see Section 5.3.2), the strong refinement on `new_node` (that its next pointer is `tl`) proves that the new head's next pointer is the old head.

The pop operation satisfies the `ts_pop` case of the `deltaTS` relation. Proving this relies on a special `observe-field` construct which is dynamically a simple field read, but statically uses an instantaneous observation to add a new stable predicate to a reference. After reading the value and tail stored at `hd`, `hd`'s predicate can be enriched with the field contents because the node is immutable (by its rely/guarantee). This 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.

Lock-Free Union Find

Anderson and Woll give a lock-free linearizable union-find implementation [7] using ranks and path compression to improve performance [63]. We have used RGREFs to verify the structural invariants for this data structure as well as that the only modifications are appropriate 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 are different, reparenting one (which previously had no parent) to the other. To improve asymptotic complexity, two optimizations are typically applied [63]. First, each node is equipped with a *rank*, which is an overapproximation of 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

```
(** * Lock-Free Linearizable Union-Find *)
Inductive cell (n:nat) : Set := mkCell : nat -> Fin.t n -> cell n.
Definition uf (n:nat) := Array n (ref{cell n|any}[local_imm,local_imm]).
Inductive chase (n:nat) (x:uf n) (h:heap) (i : Fin.t n) : Fin.t n -> Prop := ...
(* chase n x h i j provable when there is a chain of parent links reaching from i to j in array x *)
Inductive 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) *).
Inductive 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'.
(** Elided lemmas about chase, φ, δ... *)
Program Definition Find {μ n} (r:ref{uf (S n)|φ}[δ,δ]) (f:Fin.t (S n))
: rgref μ(Fin.t (S n)) μ:=
  RGFix _ _ (* RGRef fixpoint *) (fun find_rec f =>
     observe-field r → f as c
       in (λ x h, sameset f (getF parent (h[c])) x h)
       ⊓ (λ x h, rankSorted (h[c]) (h[x<|(getF parent (h[c]))|>]));
     observe-field c → parent as p in (λ x h, getF parent x = p);
     match (fin_beq p f) with (*if (p == f)*)
     (*then*) | true => rgret f (* found root --- f's parent = f *)
     (*else*) | false => (
         observe-field c → rank as rnk in (λ x h, getF rank x = rnk);
         observe-field r → p as p_ptr in
      (*A*) (λ x h, sameset p (getF parent (h[p_ptr])) x h)
      (*B*) ⊓ (λ x h, rankSorted (h[p_ptr]) (h[x<|(getF parent (h[p_ptr]))|>]))
      (*C*) ⊓ (λ x h, rankSorted (h[c]) (h[p_ptr]))
      (*D*) ⊓ (λ x h, getF parent (h[p_ptr]) ≠ p -> nonroot_rank p (getF rank (h[p_ptr])) x h)
      (*E*) ⊓ (λ x h, getF parent (h[p_ptr]) ≠ p ->
( rankSorted_strict (h[p_ptr]) (h[x<|(getF parent (h[p_ptr]))|>]) \/ fin_lt p (getF parent (h[p_ptr]))));
         observe-field p_ptr → parent as gp in (λ x h, getF parent x = gp);
         gp_cell <- Alloc! (mkCell _ rnk gp ) ;
         _ <- fCAS( r → f, c, convert gp_cell);
         find_rec p
     ) end) f.
```

Figure 5.4: A lock-free union find implementation [7] 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, making it a safe index into the array. ⊓ conjoins predicates.

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 paper, 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 have 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 discuss our overall results and outline the verification of path compression in lookup. Space prohibits a full exposition, but our proofs are available with our DSL implementation (Section 5.5).

The key invariants 1–3 are embodied in the refinement on the reference to the array, ϕ in Figure 5.4. 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: 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 5.2) 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 other work [255, 254, 215, 137], adding a same-set operation to an existing implementation requires re-verifying all operations. In our case, the same-set operation is simply verified after the other operations.

Verifying Path Compression Figure 5.4 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 is too strong. 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.

Thus, to verify that the lookup operation's path compression operation (the `fCAS`[4] 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 paper. 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 the node is its own parent 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

[4] `fCAS` is `CAS` on a single field. Its typing is analogous to CAS, but the guarantee is proven assuming conditional behavior and update to the specified field only.

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)[5]

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.

5.3.2 Concurrent RGrefs*, Formally*

This section offers a formal account of concurrency-safe RGrefs. As in the sequential case [103] (Chapter 4), 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 Pure Fragment

Figure 5.5 gives the core (runtime) typing rules for the language. The pure fragment is an extension to the Calculus of Constructions (CC [62]) with additional basic types and eliminations (natural numbers, booleans, and propositional equality of the form present in Coq's standard library), plus access to heap primitives. 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) which imposes additional criteria on pure terms. First, the type of any pure term must not contain a dereference, so the type retains the same meaning

[5]This helps establish a total rank+identity ordering on f and its ancestors.

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

$$\frac{\Gamma \vdash \tau \prec \tau \circledast \tau}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \qquad \frac{\Sigma(\ell) = A}{\Sigma; H; \Gamma \vdash \ell_{A,P,R,G} : \mathsf{ref}\{A \mid P\}[R, G]}$$

$$\frac{\Gamma \vdash \circledast A \qquad \Sigma; H; \Gamma \vdash M : \mathsf{ref}\{A \mid P\}[R, G] \qquad \mathsf{atomic}\ A \qquad \mathsf{reflexive}\ G}{\Sigma; H; \Gamma \vdash !M : \mathsf{fold}\ G\ A} \qquad \frac{\Sigma; H; \Gamma \vdash M : N \qquad \Gamma \vdash N \rightsquigarrow N'}{\Sigma; H; \Gamma \vdash \mathsf{convert}\ M : N'}$$

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash x : \mathsf{ref}\{A \mid P\}[R, G] \qquad \Gamma \vdash N : B \\ \Gamma \vdash B \rightsquigarrow A \qquad ((\mathsf{NoDerefs}(N) \wedge \forall x, h.\, G\ x\ (\mathsf{convert}\ N)\ h\ h[\ldots]) \vee (\forall x, b : B, h.\, G\ x\ (\mathsf{convert}\ b)\ h\ h[\ldots])) \\ ((\mathsf{NoDerefs}(N) \wedge \forall x, h.\, P\ x\ h \Rightarrow P\ (\mathsf{convert}\ N)\ h[\ldots]) \vee (\forall x, b : B, h.\, P\ x\ h \Rightarrow P\ (\mathsf{convert}\ b)\ h[\ldots]))\end{array}}{\Gamma; \Delta \vdash [x] := N \dashv \Gamma; \Delta}$$

$$\frac{\begin{array}{c}\Gamma \vdash y : \mathsf{ref}\{A \mid P\}[R, G] \qquad \Gamma \vdash N_0 : A \\ \Gamma \vdash N' : B \qquad \Gamma \vdash B \rightsquigarrow A \qquad \mathsf{NoDerefs}(N_0, N') \qquad (\forall h.\, h[y] = N_0 \Rightarrow G\ (h[y])\ (\mathsf{convert}\ N')\ h\ h[y \mapsto (\mathsf{convert}\ N')]) \\ \forall h.\, h[y] = N_0 \Rightarrow P\ N_0\ h \Rightarrow P\ (\mathsf{convert}\ N')\ h[y \mapsto (\mathsf{convert}\ N')]\end{array}}{\Gamma; \Delta \vdash x := \mathsf{CAS}(y, N_0, N') \dashv \Gamma, x : \mathtt{bool}; \Delta}$$

$$\frac{\Gamma \vdash M : \mathtt{bool} \qquad \Gamma; \Delta \vdash C \dashv \Gamma; \Delta}{\Gamma; \Delta \vdash \mathsf{while}\ (M)\ \{C\} \dashv \Gamma; \Delta} \qquad \frac{\Gamma \vdash B : \mathtt{bool} \qquad \Gamma; \Delta \vdash C_1 \dashv \Gamma'; \Delta' \qquad \Gamma; \Delta \vdash C_2 \dashv \Gamma'; \Delta'}{\Gamma; \Delta \vdash \mathsf{if}\ (B)\ \{C_1\}\ \mathsf{else}\ \{C_2\} \dashv \Gamma'; \Delta'} \qquad \frac{\Gamma; \Delta \vdash C_1 \dashv \Gamma'; \Delta' \qquad \Gamma'; \Delta' \vdash C_2 \dashv \Gamma''; \Delta''}{\Gamma; \Delta \vdash C_1; C_2 \dashv \Gamma''; \Delta''}$$

$$\frac{\Gamma_1; \Delta_1 \vdash C_1 \dashv \Gamma_1'; \Delta_1' \qquad \Gamma_2; \Delta_2 \vdash C_2 \dashv \Gamma_2'; \Delta_2'}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash C_1 \parallel C_2 \dashv \Gamma_1', \Gamma_2'; \Delta_1', \Delta_2'}$$

Figure 5.5: Core typing rules for concurrency-safe RGrefs. See also Figure 5.7 for meta-function definitions.

$$\frac{\begin{array}{c}\Gamma \vdash x : \mathsf{ref}\{\mathtt{nat} \mid P\}[R,G] \qquad \Gamma \uparrow x \vdash \divideontimes \mathsf{ref}\{\mathtt{nat} \mid P\}[R,G] \\ (\forall h, v.\, v = 0 -> P_0\ v\ h) \qquad \mathsf{stable}\ P_0\ R \qquad (\forall h, v, n.\, v = S\ n -> P_S\ v\ h) \\ \Gamma, n : \mathtt{nat} \vdash \mathsf{stable}\ P_S\ R \qquad \Gamma, x' : \mathsf{ref}\{\mathtt{nat} \mid P \cap P_0\}[R,G], pf : x \approx x'; \Delta \vdash C_1 \dashv \Gamma'; \Delta' \\ \Gamma, n : \mathtt{nat}, x' : \mathsf{ref}\{\mathtt{nat} \mid P \cap P_S\}[R,G], pf : x \approx x'; \Delta \vdash C_2 \dashv \Gamma'; \Delta'\end{array}}{\Gamma; \Delta \vdash !\mathcal{R}_{\mathtt{nat}}(x.f, Z \Rightarrow C_1, S\ n \Rightarrow C_2) \dashv \Gamma'; \Delta'}$$

Figure 5.6: Primitive to introduce new refinements based on observed values.

$$\mathsf{contains}_A\ R \stackrel{\text{def}}{=} \left(\begin{array}{c}\forall l : \mathsf{ref}\{A \mid Pa\}[Ra, Ga].\, \forall y : \mathsf{ref}\{B \mid Pb\}[Rb, Gb] \in h[l]. \\ \forall h, h'.\, Rb\ h[y]\ h'[y]\ h\ h' \Rightarrow R\ h[l]\ h'[l]\ h\ h'\end{array}\right)$$

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

$$\mathsf{precise}_r(R) \stackrel{\text{def}}{=} ! \notin R \wedge \forall x, x2, h, h', h2, h2'.\, \left(\begin{array}{l}(\forall l.\, \mathsf{ReachableFromIn}\ l\ x\ h \Rightarrow h[l] = h'[l]) \Rightarrow \\ (\forall l.\, \mathsf{ReachableFromIn}\ l\ x2\ h2 \Rightarrow h2[l] = h2'[l]) \Rightarrow \\ R\ x\ x2\ h\ h2 \Rightarrow R\ x\ x2\ h'\ h2'\end{array}\right)$$

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

$$\mathsf{fold}\ G\ (A * B) \stackrel{\text{def}}{=} (\mathsf{fold}\ G.1\ A * \mathsf{fold}\ G.2\ A)$$

$$\mathsf{fold}\ G\ (\mathsf{ref}\{A \mid P\}[R, G_0]) \stackrel{\text{def}}{=} R!G_0$$

$$\mathsf{fold}\ G\ A \stackrel{\text{def}}{=} A \text{ otherwise}$$

$$R.1 \stackrel{\text{def}}{=} \lambda x, x' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall y : \sigma'.\, R\ (x, y)\ (x', y)\ h\ h'$$

$$R.2 \stackrel{\text{def}}{=} \lambda y, y' : \sigma.\, \lambda h, h' : \mathsf{heap}.\, \forall x : \sigma'.\, R\ (x, y)\ (x, y')\ h\ h'$$

$$R!G \stackrel{\text{def}}{=} \lambda a, a', h, h'.\, G\ a\ a'\ h\ h' \wedge (\forall l, h[l] = a \rightarrow h'[l] = a' \rightarrow R\ l\ l\ h\ h)$$

$$\mathsf{atomic}\ A \stackrel{\text{def}}{=} A \in \{\top, \bot, \mathtt{nat}, \mathtt{bool}, \mathsf{unit}, (\Pi x : S.\, T[x]), \mathsf{ref}\{T \mid P\}[R, G]\}$$

Figure 5.7: Metafunctions for side conditions used in Figure 5.5.

in all future heaps. Second, the actual computed terms are restricted to 0 or 1 dynamic dereference expressions, via an implicit side condition MaxOneHeapAccess detailed in Section 5.3.5 on all pure subterms, and a predicate NoDerefs which prohibits use of dereference in an expression. There are reasons for this both internal and external to the pure fragment. For the pure fragment itself, it is necessary for type soundness (particularly type preservation) because unrestricted dereferences would allow construction of heap-sensitive terms: evaluating $\mathsf{refl}(!x)$ for x a reference to a `nat` might construct a value of type $3 = 3$ one time and $4 = 4$ the next, as another thread may have written through an alias in the meantime. But statically, these two terms have the same type despite potentially different types at runtime, breaking preservation. Limiting the scope of any pure term evaluation to terms with at most 1 dynamic dereference fixes type preservation. The imperative context's use for the restriction on dereferences is explained shortly.

Another restriction on pure terms is that any pure term whose result is returned to the imperative context must not contain deferred dereference expressions (e.g., dereference inside a closure). This ensures that any application of a function obtained by a stack variable or by heap dereference will not produce an infinite loop — that after at most one dynamic dereference, the term will be a straightforward term in CC, and therefore terminating under call-by-value reduction (in fact, strongly normalizing). The particular check we use to ensure at most one dynamic dereference, MaxOneHeapAccess, is already sufficient to satisfy this restriction (the predicate serves two purposes). It is essentially a predicate on the shape of the pure term, and is elaborated on in Section 5.3.5.

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 [103] (Chapter 4). 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. 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 original reference and the reference stored in the heap. This is called *folding*, shown by the fold construct in Figure 5.7. This permission-like structure is sufficient to model a form of reference immutability [252, 275, 276], which is useful for ensuring data race freedom in a concurrent programming language [105] (Chapter 3).[6] 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 the whole heap is immutable (making all predicates, even incorrect ones, stable).

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; as Chapter 4 explains, an RGREF's guarantee must imply its rely to allow free duplication without violating the compatibility invariant, and other references behave linearly (e.g., $\mathsf{ref}\{\mathtt{nat} \mid \mathsf{any}\}[\mathsf{dec}, \mathsf{inc}]$). 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. In the frequent case where $\tau = \tau' = \tau''$, we say values of type τ are *reflexively splittable*, and abbreviate the splitting judgment as $\Gamma \vdash \circledast\tau$. The pure fragment operates only on reflexively splittable data, and thus Γ contains only reflexively splittable data. Γ may be a dependent context, e.g. $\vdash x : \mathtt{nat}, y : \mathsf{ref}\{nat \mid \lambda vh.\, v = 3\}[\ldots, \ldots]$, an extension to the formal model in Chapter 4 that more closely matches the implementation embedded in COQ. 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 chapter treads lightly on the use of substructural values; any place a pure term may be used, a single variable from Δ may be used (and consumed in output Δ') instead, though we omit those rules from the current formal presentation. This is sufficient for allocating references with asymmetric rely and guarantee that support a very strong refinement, then weakening to a reflexively splittable type upon sharing (discussed near the end of this section).

[6] This actually requires a first-order heap; prior formulations of reference immutability are for object oriented programs with essentially defunctionalized program representations. The presence of a truly higher-order heap makes RI-style safe concurrency non-trivial. We discuss this further in Section 5.3.4.

One of the interesting features of the imperative fragment is shown by the $!\mathcal{R}_{\texttt{nat}}$ "refiner" in Figure 5.6. Based on the constructors of a type, the construct introduces an alias refined additionally with a new stable predicate implied by observed constructor. This is the core equivalent of the `observe-field` from Figure 5.3'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).

Treating Interleaving in Proofs

Uses of pure terms in the imperative context are subject to a number of restrictions to support equational reasoning. First, as explained above, no term may dynamically contain more than 1 dereference, which is enforced by the syntactic predicate MaxOneHeapAccess described in Section 5.3.5. This is necessary for soundness of the pure fragment, but has the added benefit that when modeling the semantics of the imperative context, atomically reducing pure terms is equivalent to modeling intermediate states, since each pure term observes the heap at most once.

Second, in some cases, no dereferences are permitted. This is required to enable equational reasoning between terms, of the sort supported when embedding RGref into a theorem prover like Coq. For example, when proving that a CAS satisfies a guarantee, we would like to compare *multiple* terms equationally; writing $b := \mathsf{CAS}(r, !r, !r + 1)$ for an atomic increment, unrestricted equational reasoning is not sound for proving that the result is an increment, because the evaluations of the "old" and "new" values would be interleaved with other threads' actions, and therefore may not actually compute two numbers differing by 1! Instead, in cases where multiple terms must be related for a proof obligation, we must restrict the terms to not accessing the heap, or restrict reasoning by abstracting a term away to its type. Prohibiting heap access in expressions appears in Figure 5.5's rule for CAS, where no nested expression may contain a dereference. Instead, relationships to the heap may be derived only from the conditional behavior of the CAS: proofs that the write respect the guarantee may assume the value overwritten ($h[y]$) is equal to the expected old

value (N_0). Abstracting to a type appears in the rule for writing to the heap: the guarantee must be proven either (1) with full term access for heap-independent terms or (2) with the term abstracted away to its type (which may have a strong refinement). This is sufficient for strong reasoning about heap-independent computations, and for 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) allows guarantee obligations to be proven by storing a precisely refined reference into the heap in Figure 5.3. This pattern of simultaneously publishing and weakening a rely-guarantee reference shows up repeatedly in lock-free algorithms. For example, see the CAS in Figure 5.3. Another good example is in enqueuing at the end of a Michael-Scott Queue (proving that the linked list remains null-terminated while appending the previously-thread-local new node). In our lock-free union-find implementation, we exploit this conversion to carry specific information about a node's rank and parent into guarantee proofs.

5.3.3 Soundness

We have proven soundness for our type system. At a high level, the proof is decomposed into two steps. First, we rely on the combination of sequential soundness (Chapter 4.4.4) with further simplifying heap restrictions (Section 5.3.5) to ensure soundness for the sequential case and termination of the pure fragment. Second, we compose soundness proofs of two threads by synthesizing classic rely-guarantee reasoning among threads [139] from the rely relations embedded in types in the typing context for each thread. The remainder of this section describes the proof in more detail in terms of embedding into the Views Framework [70], the same semantic framework used to prove soundness for concurrent reference immutability in Chapter 3.

The Views Framework abstracts core concepts of concurrent type systems and program logics into a well-specified abstract program logic; 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 logic. Proving that a given source language judgment (e.g., a typing derivation [105] 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.*[7] *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 [242] (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 with appropriate interleaving, and using syntactic λ-terms as components of the Views model $\mathcal{M}$. 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.

Because the proof mostly follows the standard Views Framework routine, for brevity we describe only our embedding in detail.

Following the parameter scheme from the main Views paper [70], 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, our views take the form:

$$\{p \in \mathcal{P}(\mathcal{M}) \mid \forall m \in p.\, \mathcal{R}(m) \subseteq p\}^8$$

[7] The influence of rely-guarantee reasoning — which can be embedded into Views — should be clear.

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

$$\text{Locations}\quad \widehat{Loc} \;::=\; \ell_{A,P,R,G} \qquad \text{Terms}\quad \widehat{M},\widehat{N} \;\in\; \text{source terms} \mid \widehat{Loc}$$
$$\text{Heaps}\quad \widehat{H} \;\in\; Loc \rightharpoonup \widehat{M} \qquad \text{Store}\quad \widehat{S} \;\in\; \mathcal{V} \rightharpoonup \widehat{M}$$

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

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

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

$$(H,\Sigma,S,\Gamma,\Delta) \bullet (H',\Sigma',S',\Gamma',\Delta') \stackrel{\text{def}}{=} \begin{array}{l}(H \cup H', \Sigma \cup \Sigma', S \uplus S', \Gamma \cup \Gamma', \Delta \uplus \Delta') \\ \mathsf{where}\ \mathsf{Valid}_{\mathcal{M}} \\ \mathsf{otherwise}\ \bot\end{array}$$

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

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

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

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

Figure 5.8: Views state space. "Hats" (e.g., $\widehat{M}$) indicate explicit annotations of RGREF components on locations, while the erased ($L\lfloor - \rfloor$) omits this.

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 5.9, 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 5.9 defines $[\![-]\!](-)$ on Views $\mathcal{M}$ for clarity and indicating how view state is updated to match actual execution (used in proving our treatment of atomic commands is sound).

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

$$
\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^H_{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}}{=} & \mathsf{let}\ \ell = S(x)\ \mathsf{in}\ (H[\ell \mapsto S(N) \Downarrow^H_{cbv}], \Sigma, S, \Gamma, \Delta) \\
[\![[x] := y]\!](H,\Sigma,S,\Gamma,\Delta) & \stackrel{\text{def}}{=} & \mathsf{let}\ \ell = S(x)\ \mathsf{in}\ (H[\ell \mapsto S(N) \Downarrow^H_{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^H_{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^H_{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^H_{cbv}], \Sigma, S[x \mapsto \mathsf{true}], (\Gamma, x:\mathtt{bool}), \Delta) \\ \quad \text{if } H(\ell) = S(M) \Downarrow^H_{cbv} \\ (H, \Sigma, S[x \mapsto \mathsf{false}], (\Gamma, x:\mathtt{bool}), \Delta) \\ \quad \text{otherwise} \end{cases} \\
[\![\mathcal{R}_{\mathtt{nat}} 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}_{\mathtt{nat}} \text{ rule's } Z \text{ case} \\
[\![\mathcal{R}_{\mathtt{nat}} 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}_{\mathtt{nat}} \text{ rule's } S \text{ case}
\end{array}
$$

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

Figure 5.9: Embedding into the Views Framework. We lift stack lookup ($S(-)$) to terms as well, substituting stack contents for variables.

components (including enforcing that they contain no dereference expressions), compatibility between references, and that all refinements are true.

The atomic statements for the framework are exactly the statements of the imperative fragment (except loops and conditionals are unrolled and become non-deterministic). Every command containing a pure term M steps by fully (call-by-value) reducing the term. This respects correct thread interleaving semantics because the typing judgments in Figure 5.5 enforce that the call-by-value reduction of any pure term will access the heap at most once.

We must also prove the axioms respect execution. Axiom soundness intuitively states that the axioms for atomic commands (the embedding of Figure 5.5's imperative typing 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 5.9. 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 20 (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 [103, 104]) given the restrictions on heap contents (Section 5.3.5). 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: This is only an atomic action when N contains no dereferences (the left disjunct in the predicate and guarantee proofs). Because the stored term contains no dereferences, it can be treated explicitly in proving the guarantee is satisfied. 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.

□

Note that writing to the heap is not always considered an atomic action! When translating a source term into the Views Framework, we sometimes decompose writes to respect proper thread interleavings. In particular, in the case where N in Figure 5.9's write interpretation contains dereferences, the granularity of interleaving is incorrect: it would in fact execute an atomic read-modify-write on the heap, rather than permitting other threads to interleave between the dereference in N and the actual heap update. This is addressed by decomposing writes with dereferences in their expressions.

When translating a type-correct write

$$[x] := N$$

into the Views Framework, we conditionally decompose the write:

$$\downarrow [x] := N \downarrow \stackrel{\text{def}}{=} \begin{cases} [x] := N & \text{if } \mathsf{NoDerefs}(N) \\ x' := \mathsf{interp'}_\tau(N); [x] := x' & \text{otherwise} \end{cases}$$

To continue reasoning modularly, we must prove an additional lemma supporting the desired proof rule for the source-level write rule.

Lemma 21 (Write Soundness)**.** *For all* $\Gamma, \Delta, \Gamma', \Delta'$*, if* $\Gamma; \Delta \vdash [x] := N \dashv \Gamma'; \Delta'$*, then*

$$\{[\![\Gamma, \Delta]\!]\} \downarrow [x] := N \downarrow \{[\![\Gamma', \Delta']\!]\}$$

Proof. The proof is by case analysis on the translation. In the first case, we must prove

$$\{[\![\Gamma, \Delta]\!]\}[x] := N\{[\![\Gamma', \Delta']\!]\}$$

for $\mathsf{NoDerefs}(N)$. This follows from axiom soundness. If N does contain dereferences, we must prove:

$$\{[\![\Gamma, \Delta]\!]\} x' := \mathsf{interp}'_\tau(M); [x] := x' \{[\![\Gamma', \Delta']\!]\}$$

We prove this using the sequencing rule, deriving the proof:

$$\begin{array}{l} \{[\![\Gamma, \Delta]\!]\} \\ x' := \mathsf{interp}'_\tau(M); \\ \{[\![\Gamma, \Delta, x' : \tau]\!]\} \\ [x] := x' \\ \{[\![\Gamma', \Delta']\!]\} \end{array}$$

This proof is derivable via sequencing and axiom soundness, re-using the antecedents of the source derivation which proved the guarantee and predicate using only the type of the stored value due to the presence of dereferences. □

Lemma 20 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 Lemmas 20 and 21, and the Views Framework's soundness.

Theorem 6 (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 20, while the compound command cases proceed mostly by the inductive hypothesis and soundness of the underlying Views Framework Logic, with the exception of heap writes, proven sound by Lemma 21. 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).

1. Heap writes: Follows from Lemma 21.

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 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.

□

5.3.4 Reference Immutability for Safe Parallelism, for RGREFs

Readers may wonder, given the relationship between rely-guarantee references and reference immutability, whether the approach used for safe concurrency in that setting [105] might be adapted for RGREFs. It can, but the requisite checks will not be so straightforward; unlike all reference immutability work, RGREFs actually have a higher-order heap, which may contain functions returning references with write capabilities. (All prior reference immutability work [28, 252, 275, 276, 132, 105] has been done in the setting of statically-typed class-based object oriented languages, essentially working on defunctionalized code with a first-order store and type tags.) Supporting read-sharing with a higher-order store also requires reasoning about the result types of any transitively reachable functions, in context or in heap, which might provide a writable reference to an imperative calling context. This is intuitively straightforward, but technically subtle. This chapter focuses on simply enabling invariant preservation in the concurrent setting, rather than expressing all semantically-justifiable forms of safe concurrency.

Parameter A (Atomic Commands) The atomic commands as in Figures 5.5 and 5.6. The compositional commands are simply inherited from the Views Framework language.

Parameter B (Machine States) $\mathcal{S}$ as in Figure 5.8.

Parameter C (Interpreting Atomic Commands) $[\![-]\!](-)$ as in 5.9, 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 5.8.

Parameter E (Axiomatization) Embedding of the typing rules for atomic commands from Figures 5.5 and 5.6, using the environment embedding from Figure 5.9.

Parameter H (Separation Algebra) $(\mathcal{M}, *, \mathsf{emp})$ as in Figure 5.8.

Parameter I (Separation Algebra Reification) Erasure $\lfloor - \rfloor$.

Parameter K (Interference Relation) $\mathcal{R}$as in Figure 5.8.

Parameter L (Axiom Soundness III) Lemma 20, 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.

Figure 5.10: Views Framework parameter instantiation, according to Dinsdale-Young et al. [70].

5.3.5 Heap Restrictions

For a variety of reasons, interactions between the pure terms and heap must be constrained. Limiting each pure term to a single dynamic heap access under call-by-value ensures that type preservation holds under fine-grained interleaving with other threads.[9] Preventing any term with a deferred dereference from returning to the imperative context ensures that applying functions from stack variables or heap dereferences will not produce an infinite loop — Landin's knot. This makes a syntactic analysis of pure terms for the number of heap accesses feasible, and ensures that after stack variable substitution and at most one heap access, the resulting term is a plain CIC term. This in turn ensures that reducing a pure subterm under call-by-value terminates, and ensures that a naïve embedding of the pure fragment using Coq's expression language allows us to construct proof terms without accidentally using terms that would not normalize. This restriction is not a fundamental requirement, but simplifies our soundness proof.

The rest of this section explains MaxOneHeapAccess and how it ensures strong normalization for the pure fragment.

Restricting Heap Access

Assuming stack variables and the heap contain no deferred dereference expressions, we describe a simple proof system over pure terms that ensures that at most one heap access will be performed dynamically during a call-by-value evaluation of the term. This directly helps type preservation for pure terms under interleaving semantics, and also justifies the simplification in Section 5.3.3 of full-reducing pure terms instead of stepping them. As a byproduct, given the restriction on the heap contents, this ensures that any pure term embedded in the imperative fragment will call-by-value reduce to a value with no deferred dereference expressions (which is then suitable for storage back to the heap or stack without violating their restrictions).

We use a simple syntactic proof system on pseudoterms, given in Figure 5.11. We use

[9] Otherwise propositional equalities on dereferences evaluated in multiple heaps breaks type preservation: $(\lambda x : \mathsf{ref}\{\mathtt{nat} \mid \ldots\}[\ldots,\ldots].\,(\mathsf{refl}\ !x, \mathsf{refl}\ !x))$ statically has type $\Pi x : \mathsf{ref}\{\mathtt{nat} \mid \ldots\}[\ldots,\ldots].\,(!x = !x, !x = !x)$, but an application may reduce to a value of type $(3 = 3, 4 = 4)$.

a combination of two predicates. $\mathsf{NoDerefs}(M)$ simply ensures that no dereferences are present in the term. Then because stack variables are never bound to closures with deferred dereference expressions, reduction of the term will not access the stack and will produce a dereference-free term. $\mathsf{MaxOneHeapAccess}(M)$ is a bit more subtle. It is defined in terms of $\mathsf{Restrict}_a^c(M)$, examines the application structure of a term and summarizes (1) how many times a term must be applied to another before its dereferences will be reduced under CBV evaluation (a); and (2) how many subexpressions that may execute together (in the same heap) contain dereferences (c), both as upper bounds. The structure of $\mathsf{Restrict}$ is necessarily incomplete; for example, because it does not reason about the applications of an argument, $\mathsf{Restrict}$ is limited to reasoning about function applications whose arguments will already be dereference-free at invocation: higher-order arguments that perform dereference when invoked are not handled.[10]

Lemma 22. *NoDeref Preservation If* $\mathsf{NoDerefs}(M)$, $FV(M) = \emptyset$, $\vdash H : \Sigma$, *and* $H; M \rightarrow^*_{\mathsf{cbv}} H'; N$, *then* $\mathsf{NoDerefs}(N)$.

Proof. By induction on the reduction sequence and concrete reductions. Most cases are immediate or by induction, except the dereference reduction, which violates the first hypothesis. □

The restriction to closed terms in Lemma 22 may seem onerous, but remember that the execution semantics in Section 5.3.3 substitute in stack variable contents that themselves contain no dereferences prior to reduction.

Lemma 23. *Restrict Soundness For all* H, Σ, Γ, M, N, *if* $\vdash H : \Sigma$ *and* $H; \Sigma; \Gamma \vdash M : N$, *then* $\mathsf{Restrict}_n^c(M)$ *implies that*

1. $H; M \rightarrow^*_{\mathsf{cbv}} v$ *for some* v,

2. *that* M *applied to at least* n *appropriately typed dereference-free arguments produces a term that CBV reduces to a value containing no dereference expressions after at most* c *heap reads.*

[10] Note that the only time this becomes a significant imposition is when trying to reuse such a dereferencing closure multiple times, which is exactly what we are trying to prohibit.

Proof. By induction on the Restrict derivation.

- No derefs: the term contains no dereferences, so it will trivially reduce to a term with no dereferences without accessing the heap.

- Increasing application and heap access counts: follows from the inductive hypothesis.

- Dereference: by the inductive hypothesis and the fact that the result of the dereference will be a value containing no dereferences.

- λ term: The term is already a (CBV) value. By the rule's premise and the inductive hypothesis, one application to a dereference-free term produces a term that reduces to a dereference-free value after at least a applications, with at most c heap accesses, so for this term the goal holds for $a + 1$ applications to dereference-free terms.

- Application term: Given the antecedents:

  - $\mathsf{Restrict}^{b}_{a+1}(M)$
  - $\mathsf{Restrict}^{c}_{0}(N)$

  N CBV reduces to a dereference-free term in at most c dereferences. Thus, reducing the application under CBV will be an application to a dereference-free term. By the inductive hypothesis, M, after $a + 1$ applications to dereference-free terms, will CBV reduce to a dereference-free term after at most b heap accesses. Thus performing one application to a dereference free term reduces by one the number of required applications to produce a dereference-free result.

- Recursors: The discriminee reduces to a dereference-free term after at most b heap accesses. Each branch of the recursor has the same restriction judgment (possibly using the lifting rules). Thus, regardless of which branch is returned, the result will require a applications to dereference-free terms, and will not exceed c heap accesses during its reduction (including after applications), so we simply add the reductions from the discriminee.

$$\frac{! \notin M}{\mathsf{NoDerefs}(M)} \qquad \frac{\mathsf{Restrict}_0^1(M)}{\mathsf{MaxOneHeapAccess}(M)} \qquad \frac{\mathsf{NoDerefs}(M)}{\mathsf{Restrict}_a^c(M)}$$

$$\frac{\mathsf{Restrict}_a^c(M) \qquad a \leq a' \qquad c \leq c'}{\mathsf{Restrict}_{a'}^{c'}(M)} \qquad \frac{\mathsf{Restrict}_0^0(M)}{\mathsf{Restrict}_0^1(!M)}$$

$$\frac{\forall v.\, \mathsf{NoDerefs}(v) \Rightarrow \mathsf{Restrict}_a^c(M[v/x])}{\mathsf{Restrict}_{a+1}^c(\lambda x.\, M)} \qquad \frac{\mathsf{Restrict}_{a+1}^b(M) \qquad \mathsf{Restrict}_0^c(N)}{\mathsf{Restrict}_a^{b+c}(M\ N)}$$

$$\frac{\mathsf{Restrict}_0^b(V) \qquad \forall i.\, \mathsf{Restrict}_a^c(M_i)}{\mathsf{Restrict}_a^{b+c}(\mathcal{R}_\tau(V, \overline{M}))} \qquad \frac{}{\mathsf{Restrict}_0^0(x)}$$

Figure 5.11: Predicates for ensuring call-by-value reduction of pure terms (1) accesses the heap at most once, and (2) produces a value with no deferred dereferences.

- Variable: Variables may only be instantiated by dereference-free terms, so the premise follows immediately.

□

Strong Normalization from Restricted Terms

The Views framework embedding given in Section 5.3.3 assumes that call-by-value reduction terminates on the pure subterms. Additionally, embedding RGref as a DSL in a proof assistant based on dependent type theory requires not accidentally introducing terms that should not terminate into proof terms (breaking strong normalization, and therefore logical consistency). Here we show that assumption holds, given the stack and heap restrictions enforced in the previous section.

The high-level intuition is that because each pure subterm is restricted to at most one heap access, and neither heap access nor variable usage can introduce additional dereference

expressions, after variable substitution and the possible heap access, the remaining term is in the CIC fragment of the term language, which is well known to be strongly normalizing.

Type soundness for pure expressions follows from type soundness for CC (really CIC given our additional data types and dependent eliminators) along with:

- Preservation holds for dereference expressions, by heap and location typing
- Progress holds for dereference expressions, since locations do not exist unless allocated and properly typed
- Definitional equality is extended such that for a location ℓ, heap type Σ and heap H typed by Σ, $\Gamma \vdash !\ell_{T,P,R,G} \equiv \mathsf{dofold}(G, H(\ell))$, where dofold performs any necessary value modification to effect folding (e.g. rewriting the terms for a pair's element types); dereference reduction selects a type-specific dofold.

This is very similar to the proof given for the original RGref system [103, 104].

The additional restriction in this chapter over the previous that no deferred dereference expressions occur in the heap or imperatively-bound variables — enforced by MaxOneHeapAccess from the previous section — rules out infinite recursion through the store. This ensures that pure terms satisfying MaxOneHeapAccess are strongly normalizing because after at most one dereference, the term is in a fragment of CIC + neutral terms (locations and the reference type constructor). In particular, membership of the reduction result in this clearly strongly normalizing language fragment follows directly from the proof that MaxOneHeapAccess ensures the absence of dereferences in the call-by-value result of its subject, since dereferences are the only non-neutral term we add to CIC.

5.4 Refinement from Rely-Guarantee Reasoning

Sequential rely-guarantee references [103] and the concurrency extension described thus far ensure only data invariants and limited temporal invariants, not functional correctness. Type-checking the lock-free increment proves that if any state change occurs it will be an increment, but does not imply that such an increment *does* occur, or occurs at most

once. But rely-guarantee references encode sufficient information about interactions between portions of a program to prove functional correctness as a relation between the input and output heaps for a procedure.

Intuitively, traversing the guarantee proofs in a typing derivation in call-by-value program order, it is possible to synthesize an *abstract trace* of an operation's execution from a typing derivation. The semantics of a trace are to build, by sequencing, a binary relation between the input heap of a procedure and the output heap; this is then interpreted as a specification. Directly reflecting an implementation into a relational specification can naturally be quite verbose, so we also support refining traces to more abstract traces. A trace t refines a trace t' if the relational interpretation of t is included in the relational interpretation of t'. This abstraction then permits us to relate implementations to more succinct and abstract specifications.

Because this abstraction process may lose information about intermediate interleavings, we are not proving any properties relating the implementation to an ideal non-interleaved execution (see Section 5.4.2). Such specifications are valuable, but our approach is useful and simpler to prove, while still laying the groundwork for proving such specifications in the future.

We demonstrate our trace refinement with a simple example, refining a trace of Figure 5.2's atomic increment function to its specification. Intuitively, the typing derivation for the atomic increment function can be traversed to provide (after minor simplification) the abstract trace:

$$\texttt{((r inc@c)}\hookrightarrow\texttt{(l eq@c))}^{+}\hookrightarrow\texttt{(r inc@c)}\hookrightarrow\texttt{(l inc@c)}$$

The abstract trace consists of one or more observations of remote (`r`) interference followed by a local (`l`) heap access to `c` (`!c` generates `eq@c`), followed by one final interference and an increment (the guarantee from typing the success case of the CAS). The relation for the interference is taken from `c`'s rely which — recalling the definition of `monotonic_counter` from Figure 5.1 — is `inc`. With a few simple rewrite rules, this can be proven to refine the specification $\texttt{(r inc@c)}\hookrightarrow\texttt{(l inc@c)}$. Without yet addressing the details of the rewrite rules: unused local observations can be dropped, and repetitions of transitive interference relations can be reduced to a single interference, yielding the a concise specification. The

only limitation of this is that taking `inc` as an abbreviation for Figure 5.1's `increment`, this permits the implementation to be a no-op: `increment` is reflexive due to the $\leq$ rather than $<$ (the $\leq$ is necessary to permit reads — reflexive heap access). To remedy this, the type rules for heap writes may be extended to check that a write through a reference with guarantee G satisfies a more precise relation G', and that $G' \Rightarrow G$. This allows the typing derivation to still check for guarantee satisfaction while carrying more precise information for tracing. When we discuss in this section taking the guarantee of a heap write, we mean the more precise G', which in this example could be a strict (irreflexive) increment.

We extend this to a pair of general techniques for abstract trace synthesis and trace refinement. For trace synthesis to be sound, any synthesized abstract trace for a given operation must describe a superset of the heap interactions and function returns that any concrete execution would observe. For the refinement to be sound, each rewrite rule must preserve or enlarge the set of executions covered by the input trace. This is similar to other refinement calculi embodied in rewrite rules, such as Elmas et al.'s program rewriting system QED [79, 78], which is in turn inspired by Lipton's theory of reduction [158]. Unlike those techniques, however, we use the relational information embodied in our traces to permit more context-dependent refinements. (We elaborate further in Section 5.6.)

We have used this approach to refine (manually constructed) traces of Figure 5.2's counter, and a Treiber stack [251], with others in progress. The only adjustment needed from verifying invariants of these examples is essentially refining the trace to show which case of the guarantees are satisfied by each heap write.

5.4.1 Relational Program Traces

We view program traces through an abstraction, as "sequential composition" of trace actions viewed from one thread's perspective, with a relational interpretation on heaps along with a treatment of return values. The thread-local view on the trace is to distinguish the actions of a single thread — which itself performs no thread creation or joins — from the interference of all other threads, to prove the actions of the single thread are correct. Trace actions include both local and remote applications of relations, which is rich enough to dis-

tinguish local observations and modifications from remote interference. This also exposes enough information to reason semantically about the commutativity and combination of trace events. The coinductive definitions for concrete traces $\mathcal{C}$ and relational traces $\mathcal{T}$ is given in Figure 5.12. An additional side condition on the append in concrete traces is that in $(H, H')_X \hookrightarrow (H'', H''')$, $H' = H''$.

A relational trace is an abstraction in the sense that in each concrete execution distinguishing the actions of one thread from the actions of others, the relational trace specifies a subset of the heaps observed by the concrete execution, such that the relations in the abstract trace model the relationship between concrete heaps observed. When an action may model multiple execution steps, the intermediate heaps do not matter (hence, a single non-iterated observation of a transitive interference relation like increment can model any number of interfering steps on an atomic counter). The actual semantic judgment for trace abstraction is given in Figure 5.12 as $\mathcal{C} \vDash \mathcal{T}$. Valid abstraction in context is modeled by applying the same (capture-avoiding) variable substitution to both traces.

5.4.2 Relationship to Sequential-Relative Specifications

Concurrent data structures have a range of possible correctness conditions — including atomicity [90, 79, 78], linearizability [122], and observational refinement [84, 255, 254] — which relate the behavior of a concurrent data structure to a sequential or atomic specification of the operation. We refer collectively to these styles of specifications as sequential-relative, or SR specifications. These are desirable specifications for concurrent data structures to support human reasoning, because they support reasoning about interleaved operations at an operation (method, function) granularity rather than interleavings of instructions. As Turon et al. [254, 253] succinctly point out, this *granularity abstraction* is highly useful to human reasoning. Granularity abstraction is also useful to tools that reason explicitly about thread interleavings, such as model checkers [181]. But we wish to abstract over interleavings.

There are two reasons we do not focus on SR specifications. First, they have limited advantage over input-output specifications for verifying *clients*. Because other threads in-

$$\mathcal{T}_A ::= \mathsf{return}\ M \mid \mathsf{local}(R) \mid \mathsf{remote}(R) \mid \mathcal{T} \hookrightarrow \mathcal{T} \mid \epsilon \mid \mathcal{T}* \mid \mathcal{T} + \mathcal{T} \mid (\zeta x : N.\, \mathcal{T})$$

$$\mathcal{C}_A ::= \epsilon \mid (H, H)_L \mid (H, H)_R \mid \mathcal{C}_A \hookrightarrow \mathcal{C}_A \mid \mathsf{return}\ M$$

$$\boxed{\mathcal{C}_A \vDash \mathcal{T}_A} \qquad \frac{}{\mathsf{return}\ M \vDash \mathsf{return}\ M} \qquad \frac{\mathcal{C}_1 \vDash \mathcal{T}_1 \qquad \mathcal{C}_2 \vDash \mathcal{T}_2}{\mathcal{C}_1 \hookrightarrow \mathcal{C}_2 \vDash \mathcal{T}_1 \hookrightarrow \mathcal{T}_2} \qquad \frac{\mathcal{C} \vDash \mathcal{T}_1 \vee \mathcal{C} \vDash \mathcal{T}_2}{\mathcal{C} \vDash \mathcal{T}_1 + \mathcal{T}_2}$$

$$\frac{}{\epsilon \vDash \epsilon} \qquad \frac{}{\epsilon \vDash \mathcal{T}*} \qquad \frac{\mathcal{C}_1 \vDash \mathcal{T} \qquad \mathcal{C}_2 \vDash \mathcal{T}*}{\mathcal{C}_1 \hookrightarrow \mathcal{C}_2 \vDash \mathcal{T}*} \qquad \frac{\mathcal{C} \vDash \mathcal{T}}{\mathcal{C} \vDash \mathcal{T}*} \qquad \frac{\exists v : N.\, \mathcal{C} \vDash \mathcal{T}[v/x]}{\mathcal{C} \vDash (\zeta x : N.\, \mathcal{T})}$$

$$\frac{R\ H_0\ H_n}{(H_0, H_1)_R \hookrightarrow_R \ldots (H_{n-1}, H_n)_R \vDash \mathsf{remote}(R)} \qquad \frac{R\ H_0\ H_n}{(H_0, H_1)_L \hookrightarrow_L \ldots (H_{n-1}, H_n)_L \vDash \mathsf{local}(R)}$$

$$\begin{array}{rl}
\mathcal{T}_1 \ll \mathcal{T}_2 & \text{where } \mathcal{T}_1 \approx \mathcal{T}_2 \\
\mathsf{local}(R) \ll \mathsf{local}(R') & \text{where } R \Rightarrow R' \\
\mathsf{remote}(G) \ll \mathsf{remote}(G') & \text{where } G \Rightarrow G' \\
\epsilon \ll \mathcal{T}* & \\
\mathcal{T} \ll \mathcal{T}* & \\
\mathcal{T} \hookrightarrow \mathcal{T} \ll \mathcal{T}* & \\
\mathcal{T}* \hookrightarrow \mathcal{T} \ll \mathcal{T}* & \\
\mathcal{T} \hookrightarrow \mathcal{T}* \ll \mathcal{T}* & \\
\mathcal{T} \ll \mathcal{T} + \mathcal{T}' & \\
\mathcal{T} \ll \mathcal{T}' + \mathcal{T} & \\
\mathcal{T} + \mathcal{T}' \ll \mathcal{T}'' + \mathcal{T}''' & \text{where } \mathcal{T} \ll \mathcal{T}'' \wedge \mathcal{T}' \ll \mathcal{T}''' \\
\mathcal{T} \hookrightarrow \mathcal{T}' \ll \mathcal{T}'' + \mathcal{T}''' & \text{where } \mathcal{T} \ll \mathcal{T}'' \wedge \mathcal{T}' \ll \mathcal{T}''' \\
(\zeta x : N.\, \mathcal{T}) \ll (\zeta y : N.\, \mathcal{T}') & \text{where } \forall v : N.\, \mathcal{T}[v/x] \ll \mathcal{T}'[v/y] \\
(\zeta x : N.\, \mathcal{T}) \hookrightarrow \mathcal{T}' \ll (\zeta x : N.\, \mathcal{T} \hookrightarrow \mathcal{T}') & \text{where } (x \notin FV(\mathcal{T}'))
\end{array}$$

$$\begin{array}{rl}
\mathcal{T} \approx \mathcal{T} & \\
\mathcal{T}' \approx \mathcal{T} & \text{where } \mathcal{T} \approx \mathcal{T}' \\
\mathcal{T}_1 \approx \mathcal{T}_3 & \text{where } \mathcal{T}_1 \approx \mathcal{T}_2 \wedge \mathcal{T}_2 \approx \mathcal{T}_3 \\
\mathcal{T} \hookrightarrow \mathcal{T}' \approx \mathcal{T}'' \hookrightarrow \mathcal{T}''' & \text{where } \mathcal{T} \approx \mathcal{T}'' \wedge \mathcal{T}' \approx \mathcal{T}''' \\
\mathcal{T} \hookrightarrow (\mathcal{T}' \hookrightarrow \mathcal{T}'') \approx (\mathcal{T} \hookrightarrow \mathcal{T}') \hookrightarrow \mathcal{T}'' & \\
\mathcal{T}* \approx \mathcal{T}'* & \text{where } \mathcal{T} \approx \mathcal{T}' \\
\mathcal{T} + \mathcal{T}' \approx \mathcal{T}' + \mathcal{T} & \\
(\mathcal{T} + \mathcal{T}') \hookrightarrow \mathcal{T}'' \approx (\mathcal{T} \hookrightarrow \mathcal{T}'') + (\mathcal{T}' \hookrightarrow \mathcal{T}'') & \\
\mathcal{T} \hookrightarrow (\mathcal{T}' + \mathcal{T}'') \approx (\mathcal{T} \hookrightarrow \mathcal{T}') + (\mathcal{T} \hookrightarrow \mathcal{T}'') & \\
(\zeta x : N.\, \mathcal{T}) \approx (\zeta x : N.\, \mathcal{T}') & \text{where } \forall v : N.\, \mathcal{T}[v/x] \approx \mathcal{T}'[v/x] \\
\mathsf{local}(L) \hookrightarrow \mathsf{remote}(R) \approx \mathsf{remote}(R) \hookrightarrow \mathsf{local}(L) & \text{where } R \circ L \Rightarrow L \circ R \\
\mathsf{local}(L_1) \hookrightarrow \mathsf{local}(L_2) \approx \mathsf{local}(L_3) & \text{where } L_2 \circ L_1 \Rightarrow L_3 \\
\mathsf{remote}(R_1) \hookrightarrow \mathsf{remote}(R_2) \approx \mathsf{remote}(R_3) & \text{where } R_2 \circ R_1 \Rightarrow R_3
\end{array}$$

Figure 5.12: Relational program traces, thread-specialized concrete program traces, trace abstraction, and selected trace equivalence and refinement rules. $\mathcal{T}$ and $\mathcal{C}$ are coinductive, modeling possibly-infinite execution traces.

teract with a structure before and after an SR operation, there is limited utility in knowing the operation was SR: other threads will destabilize most interesting postconditions of an SR operation, such as membership in a concurrent collection data structure. That said, SR specifications can still be useful for our current goals because at their core SR specification proofs give strong abstraction relations over possible executions. We plan to explore in future work.

Second, with a sufficiently expressive relational specification method, the important features of a SR specification may be encoded. Vafeiadis et al. [259] proposed (and Liang et al. [157] proved sound) a way to prove linearizability using classic thread-based rely-guarantee reasoning [139], by tying linearization points to updates of auxiliary variables representing abstract state. This even permits proofs about non-local linearization points. While our work does not presently support this, the addition of auxiliary/ghost state would be straightforward, and the proof technique should carry over as well. Thus our focus here is on building a sound foundation for functional correctness proofs, which may be extended.

5.4.3 Tracing Type Judgments

We can derive an abstract trace from a typing derivation according to Figure 5.13. It roughly follows our intuition for extracting traces: it traverses a term in call-by-value reduction order, appending recursively-generated traces. To simplify trace generation, we use a partial trace generation (it works on only a subset of programs).

Lemma 24 (Sound Trace Synthesis)**.** *For any source derivation J of $\Gamma, \Delta \vdash C \dashv \Gamma', \Delta'$, in any heap H and heap typing Σ such that $\vdash H : \Sigma$, for any execution $E = H; C \overset{\{l|r\}}{\to} H_1; C_1 \ldots H_n; v$, if $\mathsf{Trace}(J)$ is defined then $E \vDash \mathsf{Trace}(J)$.*

Proof. By coinduction on C. □

5.4.4 Refining Relational Traces

We define trace refinement as a binary relation $\ll$ on traces. It is naturally reflexive and transitive, and includes component-wise refinements (e.g., if $A \ll A'$ and $B \ll B'$, then

$$
\begin{aligned}
\mathsf{Trace}_{\_}(-) &: (\Gamma : \mathsf{Env}) \to \{C : \mathsf{Term} \mid \Gamma, \Delta \vdash t \dashv \Gamma', \Delta'\} \to \mathcal{T}_A \\
\mathsf{Trace}_\Gamma(\mathsf{return}\ \ M) &\stackrel{\text{def}}{=} \mathsf{interfere}(\Gamma) \hookrightarrow \mathsf{return}\ \ M \\
\mathsf{Trace}_\Gamma(C_1; C_2) &\stackrel{\text{def}}{=} \mathsf{ITrace}_\Gamma(C_1) \hookrightarrow \mathsf{ITrace}_{\mathsf{PostEnv}(C_1)}(C_2) \\
\mathsf{Trace}_\Gamma(\mathsf{while}\ (M)\ \{C\}) &\stackrel{\text{def}}{=} \mathsf{cofix}\ t\ (\mathsf{local}(\mathsf{Obs}(M, true)) \hookrightarrow \mathsf{ITrace}_\Gamma(C) \hookrightarrow t) \\
&\qquad \| \ (\mathsf{local}(\mathsf{Obs}(M, false))) \\
\mathsf{Trace}_\Gamma(x := \mathsf{alloc}_{A,P,R,G}\ M; C) &\stackrel{\text{def}}{=} (\zeta x : \mathsf{ref}\{A \mid P\}[R, G].\, \mathsf{local}(\lambda h, h'.\, \mathsf{Obs}(M, h'[x])) \\
&\qquad \hookrightarrow \mathsf{ITrace}_{\Gamma, x:\ldots}(C)) \\
\mathsf{Trace}_\Gamma([x] := N) &\stackrel{\text{def}}{=} \mathsf{local}(G@x) \\
\mathsf{Trace}_\Gamma(x := \mathsf{interp}(!M_r); C) &\stackrel{\text{def}}{=} (\zeta x.\, \mathsf{local}(\lambda h, h'.\, h = h' \wedge h'[M_r] = x) \hookrightarrow \mathsf{ITrace}_{\Gamma, x:\tau}(C)) \\
\mathsf{Trace}_\Gamma(x := \mathsf{interp}(M); C) &\stackrel{\text{def}}{=} (\zeta x.\, \mathsf{local}(\mathsf{Obs}(M, x))) \hookrightarrow \mathsf{ITrace}_{\Gamma, x:\tau}(C) \\
\mathsf{Trace}_\Gamma(x := \mathsf{CAS}(r, M, N); C) &\stackrel{\text{def}}{=} (\zeta x : \mathtt{bool}.\, (\mathsf{local}(\lambda h, h'.\, h[r] = M \wedge h'[r] = N \wedge x = true)) \\
&\qquad \| \ (\mathsf{local}(\lambda h, h'.\, h[r] \neq M \wedge x = \mathsf{false}))) \hookrightarrow \mathsf{ITrace}_\Gamma(C)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{interfere}(\Gamma) &\stackrel{\text{def}}{=} (\forall r : \mathsf{ref}\{A \mid P\}[R, G] \in \Gamma.\, \mathsf{remote}(R@r) \hookrightarrow)* \\
\mathsf{ITrace}_\Gamma(C) &\stackrel{\text{def}}{=} \mathsf{interfere}(\Gamma) \hookrightarrow \mathsf{Trace}_\Gamma(C) \\
\mathsf{Obs}(M_f\ (!M_r), v) &\stackrel{\text{def}}{=} \lambda h, h'.\, h = h' \wedge M_f\ (fold\ G\ A\ h[M_r]) = v \\
R@r &\stackrel{\text{def}}{=} \lambda x, x', h, h'.\, R\ h[r]\ h'[r]\ h\ h'
\end{aligned}
$$

Figure 5.13: Tracing type judgments. We describe the algorithm by traversal over a well-typed AST, assuming we can trivially map from the AST to the associated fragment of a typing derivation.

$A \hookrightarrow B \ll A' \hookrightarrow B'$). It also respects an equivalence relation $\approx$ on traces, which treats syntactic equivalence (e.g., reassociating the trace append operator) and semantic ones (e.g., relational equivalence among observations). Most rules are given in Figure 5.12.

Lemma 25 (Sound Trace Equivalence)**.** *For all abstract traces t and t', if $t \approx t'$, then for all concrete executions $\mathcal{C}$, $\mathcal{C}$ is captured by (abstracted by) t if and only if it is captured by t': $\mathcal{C} \vDash t \iff \mathcal{C} \vDash t'$.*

Proof. By coinduction on the derivation of equivalence. □

Theorem 7 (Sound Trace Refinement)**.** *For all abstract traces t and t', if $t \ll t'$, then all executions captured by t are also captured by t' ($\forall \mathcal{C}. \mathcal{C} \vDash t \Rightarrow \mathcal{C} \vDash t'$).*

Proof. By coinduction on the refinement derivation. Most rules are either simply structural and proceed by the inductive hypothesis; or exploit trace equivalence, relying on Lemma 25; or semantic, in that an observation of a relation among heaps is a refinement of an observation of another relation when the first is a subrelation of the second. □

5.5 Implementation

The type system described in Section 5.3 is implemented as a modification of the original rely-guarantee reference implementation [103], itself a shallow DSL embedding in Coq. The implementation also replaces Figure 5.7's total formulation of folding by a partial formulation. This can simplify some folding declarations, and makes others possible. For example, Figure 5.3's Node type has a specific pointer type baked in for the next node, but this is not exposed as an index to the type. So if the pointer type allowed mutation, a weakening of that member could no longer be used to construct a Node (in contrast to pairs, where the index type for the member could be weakened).

We have manually constructed the explicit naïve traces for Figure 5.2's counter, and a Treiber stack [251]; and we have carried out the refinement proofs according to the refinement logic in Section 5.4.4 in Coq. We expended only minimal effort on proof automation for trace refinement, so the proofs are currently somewhat verbose.

Our implementation is available at

`https://github.com/csgordon/rgref-concurrent/`

5.6 Related Work

This chapter is related to work in three main areas: general approaches to proving correctness for concurrent programs, adding dependent types to imperative languages, and concurrent program logics. Chapter 2 covered the latter two well, so we focus here on correctness criteria for concurrent programs, with only minor technical notes relating to the other topics.

5.6.1 Atomicity, Linearizability, and Observational Refinement

The most intuitive correctness condition for concurrent data structures is *atomicity*; that all observable effects of an operation appear atomically to other threads, and no threads observe intermediate states (and the operation otherwise has the desired sequential behavior). The main approach to proving atomicity is derived from from Lipton's theory of *reduction* [158] (often called *movers* in later literature), a technique for reasoning about the commutativity of certain program actions with actions in other threads. The standard approach to proving atomicity is Flanagan and Freund's type and effect approach, enriching a data race freedom type system where each statement's effect is a mover and sequential composition combines adjacent movers [90]. This is sufficient to coalesce actions for an operation using two-phase locking (possibly weakened using purity analysis [88]) into a single atomic action. Elmas et al. [79, 78] use a set of rewriting rules on programs to rewrite machine-executable programs into atomic ones (executable programs are a subset of the language used, which includes atomic blocks). Their refinement approach is quite effective, but the fundamental limitation is that every program transformation used must be valid in all environments. Our approach permits refinements for more programs (or traces) because the rely component of references bounds interference.

Linearizability [122] is a related correctness condition for concurrent data structures. Informally, it ensures that any effect of a data structure operation (1) satisfies the operation's specification, and (2) takes effect atomically at one point in time between the operation's invocation and its return. Somewhat more formally, it ensures that there is an abstraction map from the physical data structure to an abstract representation, defined throughout an

operation's execution, and the abstract state changes at only one point in time during the operation's execution — the linearization point[11] — and that the effects of any execution are equivalent to some serialization of interleaved operations. The main factors that may make linearizability unpleasant to prove are that it (1) assumes a closed set of operations on a data structure; and (2) one operation's linearization point may occur in the execution of another operation. We are not the first to exploit rely-guarantee reasoning to relax these restrictions for reasoning about concurrent programs. As discussed in Section 5.4.2, Vafeiadis et al. [259] proposed a technique using rely-guarantee logics with auxiliary variables representing the abstract state to observe non-local linearization points. This was only recently proven sound [157] by Liang and Feng in a variant of LRG [82]. We believe this approach could be adapted for our abstract trace approach. Unlike Vafeiadis et al., we use a localized notion of rely-guarantee reasoning, which means that we can verify a data structure operation once for any number of structures, while their approach requires redoing the proof for every new instance of the data structure. Unlike Liang and Feng, our approach builds on a verification approach designed to scale beyond individual methods.

Alternatives have been proposed as more intuitive, notably observational refinement [84], which has recently inspired a flurry of work in the program logic community by Turon et al. [255, 254, 253] using explicit notions of lifetime protocols for how nodes of an FCD evolve over time. They use knowledge of a node's current protocol state and the protocol itself to induce a rely relation on the structure, enabling them to prove that FCD operations contextually refine atomic implementations. Their semantics and logic treat higher-order functions with heap write access, while for simplicity we do not. To our knowledge, their approach is unimplemented, and proofs using their system have been done by hand.

Atomicity, linearizability, and observational refinement (for concurrent data structures) all relate concurrent operations to atomic executions of sequential versions. But each is also internally focused on a data structure implementation, ignoring client concerns as explained in Section 5.4.2.

Our trace format is reminiscent of the interleaved trace semantics for concurrent sepa-

[11] This ignores effectless operations, such as checking if an element is in a structure.

ration logic [43]. Ours, however, is weaker because we cannot reason directly about thread creation and termination. The next logical step for our work is to define a logic that interacts with the rely-guarantee references to prove functional correctness, as each set of references conceptually induces a set of stable assertions about program state. Thus this chapter plays a similar role to Turon et al.'s semantic reasoning about FCDs [254], where proofs are carried out in the interleaved trace semantics based on protocols for state evolution. Our next step is then similar to their subsequent work on a proof system over those semantics [253].

5.6.2 Dependent Types for Concurrent Programming

Integrating dependent types and state is a long-standing challenge for program verification. One approach is allowing types to depend on immutable data [95], often using a decidable theory of refinements [273, 230]. More recent and powerful approaches include Hoare Type Theory (HTT) [185, 189] as implemented in several Coq embeddings [52, 190, 50] and the Dijkstra Monad [245] as implemented in F$\star$ [243]. HTT is in some ways a monadic embedding of Hoare logic and separation logic in CIC, but this understates its elegance: the expression language (which may not access the heap) is dependently typed and logically consistent, which lends itself quite cleanly to embedding in a dependently-typed proof assistant like Coq. The Dijkstra Monad has a similar flavor, but focuses on Dijkstra's predicate transformers [69]. HTT has considered concurrency in the form of transactional memory [186], but does not handle directly-executable fine-grained concurrency. Treatments of the Dijkstra monad have not yet explored concurrency. F$\star$ includes a variant of the Dijkstra monad which enforces that all heap updates must satisfy a two-state invariant on heaps (as a binary relation), and includes axioms for proving properties of a current heap by relating it 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 shared-state protocol.

By contrast, the version of rely-guarantee references presented in this chapter allow direct statement of refinement types over arbitrary mutable data, subject to a semantic

check (stability) that the type is sensible and no operations will arbitrarily change the meaning of the type.

Other applications of dependent type theory to concurrency have been limited to singleton types — types parameterized by the identity of some object, typically over a location or the identity of a synchronization primitive. Examples include associating data with the guarding lock [3, 87, 86, 85, 39, 38], or associating capabilities to acquire further locks with something granting the capability [102].

5.6.3 Interference Summary Verification

The idea of explicitly treating interference between program components originated in the concurrent program logic literature, though recently the idea has begun to take root in sequential verification. Most concurrency work focuses on a notion of separation (e.g., Concurrent Separation Logic [43, 219]), read-sharing (e.g. SL with fractional permissions [36]), or rely-guarantee reasoning [204, 139, 260, 82, 73, 71]. The separating and read-sharing approaches prevent interference by other threads, allowing sequential separation logic proof rules to go almost unchanged. The one exception is critical sections, where only invariants can be established. This can in turn be seen as a particular form of the stability requirement from rely-guarantee approaches, where interference is *explicitly* considered, and all assertions must be stable with respect to interference (the exposition of the Views Framework [70] makes explicit the notion that these weaker approaches to non-interference are simply restricted treatments of interference). Verification by interference summary does not necessarily require an explicit rely relation as here; actions/interference may take the form of transition systems [260, 82, 187], capabilities with interpretations [71, 242], or abstract representations of other threads' contributions [156].

Only a handful of work treats interference explicitly in the sequential setting. Wickerson et al. [269] provide a notion of explicitly stabilizing an assertion with respect to a (potentially abstract) rely relation, essentially generating a verification which may be instantiated by a range of rely relations (addressing modularity issues with traditional rely-guarantee). They study a sequential version of the UNIX v7 memory manager using this approach. Our

`observe-field` / refiner constructs bear strong resemblance to Wickerson's $\lceil p \rceil_R$ operator, which produces the strongest assertion weaker than p and stable with respect to R. The original rely-guarantee references [103] are an adaptation of rely-guarantee reasoning between threads [139] to references, essentially treating actions through each reference as occurring in different threads. Most recently, Milião et al. [174] propose rely-guarantee protocols, a system of linear capabilities where two capabilities to the same heap cell exist only if they are compatible according to a rely-guarantee simulation. Their system is considerably less expressive than ours, focused on handling strong updates (of capabilities), while we can verify much stronger properties. But as a result, their system is more amenable to automation.

5.7 Conclusions

We have shown how to enable refinement types constraining mutable data in a shared-memory concurrent setting, and how to exploit restrictions on mutation to refine abstract traces to sensible relational specifications. The latter contribution was not an initial design goal for the system we extend (RGREFs), so we view this work as an instance of exploiting semantic checks of rich type systems for additional purposes. 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 exploiting O'Hearn et al.'s Hindsight Lemma [201] to coalesce trace actions from traversing a linked data structure (e.g., a linked list) into a single heap relation witnessing reachability. The criteria for its applicability are given as a set of invariants and two-state invariants on data structure links, all readily expressed using rely-guarantee references. In our work verifying traces, we have already abstracted a general characterization of when the Hindsight Lemma is applicable, in terms of RGREFs whose rely and guarantee enforce the necessary conditions from O'Hearn et al.'s description. Another 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 [71, 242]. Finally, proving refinement within the type system (as mentioned in Section 5.6.1) rather than over an (abstract) semantics would also be desirable.

More broadly, this chapter shows the power of viewing aliasing as a matter of interference between references. Both the modest technical changes to the RGREF type system to support concurrency, and the fact that no additional technical machinery was required within the type system itself to prove functional correctness, strongly suggest that alias interference is truly a foundational consideration. Adequate description of alias interference is sufficient to treat concurrency and functional correctness, which are typically considered significantly more complex than preserving invariants in sequential programs.

Chapter 6

RELY-GUARANTEE REFERENCES FOR EXISTING CODE

> A program should be structured in such a way that the argument for its correctness is feasible.
>
> Dijkstra, EWD 1298

Chapters 4 and 5 develop a rich theory for rely-guarantee references. But in contrast to Chapter 3's discussion of a Microsoft team's long-term experience using reference immutability — a once-in-a-lifetime opportunity for thorough evaluation — the implementations backing those chapters are for a synthetic language, designed for exploring the expressive boundaries of the RGref approach. This means that not only is all code ever written in that language described in this dissertation, but the goal of pushing expressiveness leads to a prototype that is unlikely to be usable by anyone lacking significant Coq experience. This latter point in particular is a completely unreasonable expectation for practicing developers for the foreseeable future, which begs the question of what an RGref implementation would look like if designed for (motivated, sophisticated) practicing developers rather than experts in program verification. The goal of this chapter is to explore such a design.

In short, our goal is to convert existing code written using mutable references to use rely-guarantee references. This requires an RGref implementation full-featured enough to easily port existing non-trivial code. The Coq DSL from previous chapters does not fit this bill, nor does its Agda counterpart, as both DSLs model only enough language features to explore the core design space for rely-guarantee references. Instead, we adapt Liquid Haskell [261, 262, 263] to support an embedding of rely-guarantee references.

Liquid Haskell is the latest in the line of work on Liquid Types [229, 141, 142, 230, 138, 140, 228, 227]. 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 is in some ways similar to F* [243, 245, 241], though F* uses proper inductive types for proofs (producing a proof certificate), while Liquid Types identify propositions with boolean predicates, meaning Liquid Types have semi-decision procedures for predicates.

To adapt rely-guarantee references to Haskell, we simplify the design slightly. The key simplification is that we omit transitive heap access in predicates and relations. This sacrifices significant 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). Less significant simplifications are described in Section 6.2. Our simplification and choice of Haskell for a target language likely skews our results to be more positive than if we adapted code from a stateful object-oriented language, or even OCaml, but we believe these choices strike a good balance between realistic evaluation and something we can effectively evaluate with available personnel.

In short, the contributions of this chapter are:

- We describe a simplification of RGrefs which we believe strikes a useful balance between expressiveness and effective tooling.
- We describe the design and implementation of Liquid RGrefs, an RGref implementation for Haskell built on Liquid Haskell.
- We show through case studies on real Haskell code that Liquid RGrefs can verify important properties of real code.
- We propose RGrefs as a natural stateful complement to pure refinement types

The rest of this chapter gives background on manipulating state in Haskell (Section 6.1.1) and the Liquid Haskell approach to refinement types (Section 6.1.2), discusses the technical challenges with embedding rely-guarantee references' higher-order logical proof obligations into Liquid Haskell (Section 6.2), details an embedding of rely-guarantee references into Liquid Haskell (Section 6.3), and discusses application to several small but real Haskell data structures (Section 6.4) before discussing related work and concluding.

6.1 Background: State and Refinement Types in Haskell

Haskell is an unusual programming language in that it strictly encapsulates stateful computation inside monads [178, 264]. For expressing imperative programs, a monad can be thought of intuitively as a data type *representing* a computation, which is then interpreted by the runtime.[1] This pushes Haskell code to avoid mutation more than the more prevalent "functional-first" language designs (i.e., Scala, OCaml, F#, etc.). As a consequence, there are some important differences between Haskell and the languages studied in Chapters 4 and 5 that we must explain. In addition, we also explain an existing implementation of refinement types [95] for Haskell, upon which we build our RGref implementation for Haskell.

6.1.1 State in Haskell

Heap access in Haskell is always monadic. This means that a procedure that accesses the heap is represented by a function returning a representation of an imperative computation, which is then interpreted by the Haskell runtime system. In Haskell, the monad for accessing mutable state is the `IO` monad, and representations of imperative computations are typically called *IO actions*. An element of the type `IO t` is a representation of an imperative computation that returns an element of the type `t`. For example, the type of an imperative computation of the ith Fibonacci number might be

```
Int -> IO Int
```

This type describes a function from an integer to an IO action computing an integer.[2] The IO monad provides a number of basic imperative computations, each accessing the heap at most once, which may be composed via sequencing (monadic bind). A consequence of these primitives is that dereference is no longer an expression subject to the useful algebraic

[1] We intentionally avoid an involved introduction to monads. This intuitive understanding should suffice for the non-expert reader. The interested reader can readily find the products of the small cottage industry of producing monad tutorials, or preferably should refer to one of the earliest and still clearest introductions, by Wadler [264].

[2] Technically, mutable heap access is also available in the `State` monad [150, 149], and the `IO` monad specializes `State`. All of our constructions could also be developed for `State` as well, but both for simplicity and because our examples use `IO`, we focus on the `IO` monad.

reasoning exploited in Chapter 4. This relieves us of concern for inappropriately equating dereference expressions reduced in different heaps (4.5.1, 5.3.5). (It also prohibits recursion through the store in the pure fragment of the language, though this is less important because we are not using Haskell itself as a logic; it need not be normalizing.) Equational reasoning is valid only within the expressions passed to monadic primitives. For example, an expression of the form $f(!a)$ in earlier chapters must be restructured[3] monadically as

```
do {
  x <- readIORef a;
  return f x
}
```

where a evaluates to an `IORef` (the Haskell reference type). In this case, equational reasoning is available over a and $f\ x$, but not over the composed transformation of heap access. This invalidates some of the convenient reasoning principles exploited in Chapter 4 and to a lesser extent in Chapter 5. Fortunately, we already demonstrated in Chapter 5 that simply the type of x is often sufficient to carry a lot of information. With the addition of Liquid Haskell's direct refinement types (as opposed to RGREF's indirect refinements), the type is sufficient even more often.

The monadic heap access also discourages heavy use of mutation, and encourages references to larger pure-functional structures.[4] This in turn should simplify many RGREF proof obligations.

One final factor that simplifies reasoning is the primitive `modifyIORef`, which replaces a reference's current value with the result of a function applied to the current value. Thus, to increment the value at a reference `r :: IORef Int`, we may write

```
modifyIORef (\ x -> x + 1)
```

This allows sound equational reasoning between the old and new value of the heap cell.

---

[3] Assuming no further dereferences in f or a.

[4] Another reason for using less state modification in Haskell is that Haskell's lazy evaluation can be used for memoization, lazy initialization, and other idioms which requires mutation in strict languages. See for example `http://www.haskell.org/haskellwiki/Performance/Laziness`.

6.1.2 Refinement Types in Haskell

As mentioned earlier, Liquid Types [229, 141, 142, 230, 138, 140, 228, 227, 261, 262, 263] 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 which are useful for verifying safety properties with modest annotation burden and user expertise.

The latest incarnation of these ideas, Liquid Haskell [261, 262, 263], implements Liquid Types for Haskell, extending the base theory to tackle issues with type-classes, generating verification conditions for lazy evaluation [263], and polymorphism over refinements [261], 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 refinements 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.

As a simple concrete example, consider the specification of `min` on integers, due to Vazou et al. [261]:

```
{-@ 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.

Measures, Axioms, and SMT When verifying a program, it is generally necessary to give new logical definitions in order to write (and prove) rich specifications. In Liquid Haskell, these definitions arise in two ways. First, *measures*[5] may be defined which behave as partial computable predicates over some data.[7]

Second, axioms may expand the meaning of uninterpreted functions with select computation rules. We demonstrate the use of axioms by simplifying an example due to Liquid Haskell's authors [261] 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.

Figure 6.1 demonstrates a use of axioms in Liquid Haskell.

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

[5]The name originates from the original use for proving termination (e.g., *measuring* the size of a data structure as a termination metric)[6]

[7]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 [263]).

```
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)))
```

Figure 6.1: Example usage of Liquid Haskell axioms.

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 where the axiom is necessary to prove an entailment. In this case, the return value of the `fibhs`[8] routine requires the axiom. Without knowledge of the meaning of the measure `fib`, Liquid Haskell cannot prove 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

[8]The `hs` suffix matches the `.hs` file suffix for Haskell programs, to distinguish it from the SMT definition of the ith Fibonacci number.

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.

6.2 Embedding Rely-Guarantee References in Liquid Haskell

There are two essential issues to consider when embedding RGRefs into Liquid Haskell. First, there are design and expressiveness issues to resolve. Second, there are some technical challenges in embedding RGRef proof obligations into Liquid Haskell's type system.

6.2.1 Design of a Simplified RGRef *System*

In designing a variant of RGRefs for non-experts in program verification, we must make careful choices in restricting RGRefs. Cutting features from the designs in Chapter 4 and 5 costs expressiveness, but drops some mental burden on developers and simplifies encoding into Liquid Haskell. Proper evaluation of our choices would ultimately require a user study, which we do not undertake here. Instead, we leverage our experience with the weaker but heavily used reference immutability prototype of Chapter 3 and attempt to design a system that feels, to our taste, to mesh well with Liquid Haskell.

We make three essential simplifications to RGRefs for Liquid Haskell.

First, we give up the transitive heap access in predicates and relations. This is a significant loss of expressiveness in exchange for a significantly simpler design, implementation, and use; but the impact is tempered by properties of typical Haskell programs. In particular, the system described in this chapter cannot express global predicates and relations over large pointer structures. For a different target language, this would likely be too severe a loss of expressiveness. But deep mutable pointer structures in Haskell are rare, so in this particular setting the expressiveness is not so hampered. In exchange for these weaker specifications, we remove some of the most subtle proof obligations of Chapters 4 and 5: containment and precision checks are no longer necessary, and relation folding collapses to the identity transformation in all cases (so it is essentially not present).

Second, we restrict our Liquid RGRefs to reflexively splittable references: each reference's guarantee relation must imply its rely. The substructural values in earlier prototypes

were inconvenient to use, and Haskell's focus on immutability makes them less useful.

Finally, we collapse rely and guarantee relations. This is mostly for simplicity, though splitting them in our implementation would mostly be a matter of verbosity rather than technical challenge, and could in theory be addressed somewhat by Liquid Haskell's support for abbreviations. However, unlike the previous two simplifications, this could be undone in the future without breaking existing code, by defining a new type for asymmetric rely and guarantee, and redefining the current type as an alias that instantiates both the rely and guarantee with the same relation. We defer asymmetric rely and guarantee relations in Liquid RGREFs to future work because this chapter's primary goal is to demonstrate that RGREFs do not fundamentally require expertise in dependent type theory or theorem proving; collapsing the relations is orthogonal to this aim, and does not change the shape of our solution.

6.2.2 Encoding Higher Order Proofs in a First Order Logic

The second issue to consider is the technical matter of embedding our chosen simplifications to RGREFs into Liquid Haskell. There are two primary distinctions between the CC/CIC/COQ setting for the original RGREF work and the logic of refinements used by Liquid Haskell, one minor, the other more fundamental.

Classical Logic The minor difference is the change to using classical, truth-valued logic. A consequence of this is the absence of inductive definitions for propositions. Since we have typically used inductive definitions of predicates and relations in the past as essentially recursive disjunctions, and the need for recursion is mostly obviated here by our simplification to local predicates and relations, this is not a deep change.

First-order Logic The logic of refinements in Liquid Haskell is a first-order logic over Haskell program values as atoms. This means that the refinement logic cannot even state relationships between quantified predicates and relations, which is required to type-check any form of RGREFs. In particular, the requirement that a predicate be stable with respect to some rely relation is a statement in higher-order logic, because it states a property of a

predicate and relation, rather than just atoms. This is a more substantial hurdle, and the remainder of this subsection explains our encoding.

Recall that the higher-order proof obligations for RGrefs with the simplifications from Section 6.2.1 are:

- *Stability*: A well-formed rely-guarantee reference must have a predicate preserved by its rely. In CIC, the heapless version of this property is given simply by the type

$$\mathsf{stable}_A\ P\ R = \forall x.\, \forall y.\, P\ x \to R\ x\ y \to P\ y$$

- *Relational Implication*: To check validity of reference duplication in an implementation with separate rely and guarantee relations, an essential check is that the guarantee of each reference implies the other's rely. This is again a construction on predicates that cannot be performed with Liquid Haskell's single-level predicative logic.

$$G \Rightarrow_A R = \forall x.\, \forall y.\, G\ x\ y \to R\ x\ y$$

Liquid Haskell lacks constructions on refinements other than application, thus the above definitions cannot be directly expressed in the refinement logic.

An important insight, however, is that the context for Liquid Haskell's refinement logic includes Haskell values. And the SMT solver discharges proofs in its logic *in the context* of a number of Haskell bindings. So we can actually (ab)use Haskell's binders for the quantification we need.

We can encode each of these obligations (for fixed parameters) as a particular refined function type.

- *Stability*: For some fixed A, P, and R, a proof of $\mathsf{stable}_A\ P\ R$ can be encoded as a Haskell function with refinement type:

$$\{x : A \mid P\ x\} \to \{y : A \mid R\ x\ y\} \to \{z : A \mid P\ z \wedge z = y\}$$

  The only possible implementation of a function with this type is

$$\lambda x.\, \lambda y.\, y$$

so supplying a term with this type is not difficult for the user (or an elaboration mechanism such as Template Haskell [237]), but this term only has the correct type if the SMT solver can prove the desired implication.

- *Relational Implication*: For some fixed A, R, and G, a proof of $G \Rightarrow_A R$ can be encoded as a Haskell function with refinement type:

$$\{x : A \mid \top\} \to \{y : A \mid G\ x\ y\} \to \{z : A \times A \mid R\ (\mathsf{fst}\ z)\ (\mathsf{snd}\ z) \wedge z = (x, y)\}$$

  As above, this has only one possible implementation:

$$\lambda x.\, \lambda y.\, (x, y)$$

  which is only well-typed if the SMT solver has proven the correct implication for the instantiation of the relations.

So we can force the SMT solver to solve specific instances of these obligations, but these obligations show up repeatedly at different types; every allocation for example requires both of these proofs, but each allocation may use different predicates. Thus to associate a proof with each place they are required, we must add additional runtime arguments to some primitives, with the types given above. This is inconvenient, but not fundamental, for a few reasons:

- Liquid Haskell could in principle be modified to permit erasable arguments of this type.

- These terms could be provided by an elaboration mechanism, such as Template Haskell [237] since there is only one possible implementation for each proof term.

- These terms are used in only a few places in code, essentially as refinement casts (see Section 6.3). Thus an optimizing compiler should be able to erase most uses, and with some compiler hints, all uses.[9].

[9]GHC actually includes pragmas to rewrite certain terms to other terms, allowing us to rewrite our extended primitives into direct calls to the true Haskell implementations.

We emphasize that these are not fundamental issues, each has solutions, and our goals are to evaluate expressiveness rather than to build a high-performance implementation.

6.2.3 Guiding Refinement Inference

Liquid Haskell inherits the highly effective refinement inference procedures of earlier work on Liquid Types [229, 141, 142, 230, 138, 140, 228, 227]. This inference is so effective that it often infers that assuming a function argument's refinement to be false will discharge all implications! Sadly, this is an instance where the design of Liquid Haskell diverges from our particular needs. In general, it is useful for Liquid Haskell to infer false as a refinement for a variety of reasons, notably handling dead branches when pattern matching a refined value. RGRefs were intended to never permit false predicates or guarantees. With our current simplification to a single relation, this also means no false rely relations. Thus to guide Liquid Haskell in the right direction, some of our primitives require additional "inhabitation" arguments, whose types imply that the predicate and relation are non-empty. On occasion even this is insufficient, and we have resorted to refactoring some existing code to give more explicit annotations to things like allocations.

One final subtlety, more a matter of use than definition, is guiding inference of *relations*. This is specifically the goal to have Liquid Haskell infer a refinement for one variable that must use some other variable. For example, consider inferring a relationship between `x` and `y` as a predicate on `y` that uses `x`, as in the context of a function type

$$f :: \forall \langle P :: a \to b \to \mathsf{Prop} \rangle .\, x : \{v : a \mid \ldots\} \to y : \{v : b \mid P\ x\ y\} \ldots$$

inferring the instantiation of P at a call to this function (this appears in the type of RGRef allocation). In these instances, for inference to work correctly, the argument passed for `y` must use the exact binding of `x`, not merely a syntactically equivalent expression. So to infer a relation that is (at least) reflexive when `x` is a nontrivial expression, rather than invoking `f` as

```
f (g e) (g e) ...
```

`f` must instead be invoked as:

```
let x = g e in
f x x ...
```

This restriction is not unreasonable to infer an explicit connection to distinguish a genuine relationship from happenstance syntactic equality, even in a purely functional setting. The alternative would be explicit application of type parameters a la System F [222, 100]. However, this is not in line with the spirit of Haskell, where one of the primary design goals is to have no explicit type applications in the source language, though they are present in the internal core language used in the most popular Haskell compiler.

In one instance during our evaluation, the refactorings described here proved insufficient to infer the correct rely relation at an allocation site. To work around this, we provided the inhabitation witness `undefined` — a term which raises an error when evaluated rather than producing a value, and therefore carries the refinement `false`. This is unfortunate as it may be used to prove any refinement! But we have done so only after any meaningful obligations have been discharged, in the context of a refactored allocation, making it essentially a very awkward encoding of explicit predicate passing.

6.3 Defining Liquid RGrefs

With the simplifications of Section 6.2.1 and the encoding approach of Section 6.2.2, we can define the core primitives of Liquid RGrefs in Liquid Haskell. Figure 6.2 gives the code for our core primitives, omitting the `atomicModifyRGRef` (wrapping `atomicModifyIORef`) and a small number of additional primitives built atop these. These are essentially wrappers around the standard Haskell IORef primitives, which we describe in turn before presenting a familiar example program, and discussing the Liquid RGref version of Chapter 5's construct for observing new stable refinements.

Representing RGrefs A rely-guarantee reference is implemented as a wrapper `RGRef` around an underlying `IORef`, using Liquid Haskell to parameterize the type over a predicate and combined rely-guarantee relation. The refinement, because it only applies to the immediate heap cell, can actually be represented directly in Liquid Haskell, giving the underlying `IORef` storage type `a<p>` rather than simply type `a`. This allows some of the further prim-

```
{-@ 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 >.
                  e:a<p> -> e2:a<r e> -> f:(x:a<p> -> y:a<r x> -> {v:a<p> | (v = y)}) ->
                  IO (RGRef <p, r> a) @-}
newRGRef :: a -> a -> (a -> a -> a) -> IO (RGRef a)
newRGRef e e2 stabilityPf = do {
                          r <- newIORef e;
                          return (Wrap r)
                        }

{-@ modifyRGRef :: forall <p :: a -> Prop, r :: a -> a -> Prop >.
                  rf:(RGRef<p, r> a) ->
                  f:(x:a<p> -> a<r x>) ->
                  pf:(x:a<p> -> y:a<r x> -> {v:a<p> | (v = y)}) ->
                  IO () @-}
modifyRGRef :: RGRef a -> (a -> a) -> (a -> a -> a) -> IO ()
modifyRGRef (Wrap x) f pf = modifyIORef x (\ v -> pf v (f v))

{-@ measure pastValue :: RGRef a -> a -> Prop @-}
{-@ qualif PastValue(r:RGRef a, x:a): (pastValue r x) @-}
{-@ measure terminalValue :: RGRef a -> a @-}
{-@ qualif TerminalValue(r:RGRef a): (terminalValue r) @-}
{-@ assume axiom_pastIsTerminal :: forall <p :: a -> Prop, r :: a -> a -> Prop>.
                           ref:RGRef<p,r> a ->
                           v:{v:a | (pastValue ref v)} ->
                           pf:(x:{x:a | x = v} -> y:a<r x> -> {z:a | ((z = y) && (z = x))}) ->
                           { b : Bool | (((terminalValue ref) = v) <=> (pastValue ref v))}
                           @-}
axiom_pastIsTerminal :: RGRef a -> a -> (a -> a -> a) -> Bool
axiom_pastIsTerminal = undefined

{-@ assume readRGRef :: forall <p :: a -> Prop, r :: a -> a -> Prop, pre :: RealWorld -> Prop>.
                  x:RGRef<p, r> a -> IO ({res:a<p> | (pastValue x res)}) @-}
readRGRef (Wrap x) = readIORef x
```

Figure 6.2: RGREFs in Liquid Haskell

itives to rely less on RGREF-specific axiomatization than they might otherwise (they still rely on correctness of Liquid Haskell).

Allocation `newRGRef` wraps `newIORef`, allocating `e` on the heap. It also requires two extra parameters `e2` and `stabilityPf`. `e2` is present to guide inference as described in Section 6.2.3, acting as evidence that the relation is non-empty. `stabilityPf` is a stability proof parameter, exactly as described in Section 6.2.2.

Updating an `RGRef` `modifyRGRef` wraps Haskell's `modifyIORef`, taking a reference to update and a function to apply to the reference's heap contents, as with the underlying primitive. It also takes a stability proof for the predicate and relation, as with the allocation primitive. Unlike the allocation case, where the proof is required for soundness, in this case the proof is used as a coercion. In the body of `modifyRGRef`, the anonymous function

```
(\ v -> pf v (f v))
```

returns the same value as the function `f` itself, but with `p` as the refinement on the result, rather than `r` applied to the argument, making it compatible with Liquid Haskell's interpretation of the `IORef a<p>` underlying the `RGRef`. Recall from Section 6.2.2 that the only sensible implementation of a stability proof is

```
(\ x y -> y)
```

so substituting the body produces an η-expansion of `f`.

In principle, we could use Liquid Haskell's `assume` to avoid the need to prove to Liquid Haskell that the update to the underlying `IORef` was acceptable, but for now we prefer the more principled approach, despite the modest runtime cost. Another alternative approach would be to expose an explicit cast function that applied a stability proof to the result of an appropriately typed function (e.g., `modifyRGRef`'s `f` argument), and leverage GHC's rewrite pragmas to optimize the calls inside the compiler to a direct call to simply `f`.

Reading an `RGRef` Reading an RGREF using `readRGRef` is simple in the most basic case; the primitive simply returns the referent, refined to satisfy the stated predicate. The

```
module MonotonicCounter where
import RG

{-@ stable_monocount :: x:{v:Int | v > 0 } -> y:{v:Int | x <= v } ->
                        {v:Int | ((v = y) && (v > 0)) } @-}
stable_monocount :: Int -> Int -> Int
stable_monocount x y = y

{-@ alloc_counter :: () -> IO (RGRef<{\x -> x > 0}, {\x y -> x <= y}> Int) @-}
alloc_counter :: () -> IO (RGRef Int)
alloc_counter _ = newRGRef 1 3 stable_monocount

{-@ inc_counter :: RGRef<{\x -> x > 0}, {\x y -> x <= y}> Int -> IO () @-}
inc_counter :: RGRef Int -> IO ()
inc_counter r = modifyRGRef r (\x -> x + 1) stable_monocount
```

Figure 6.3: A sequential monotonically increasing counter implemented in Liquid Haskell + Liquid RGREFs.

specification is given using a Liquid Haskell `assume` because the result is also refined with the opaque `pastValue` refinement, described in Section 6.3.2.

So in only a few lines of code, we can define the essence of RGREFs in a form we show in Section 6.4 is useful for verifying real code.

6.3.1 A Familiar Example

To demonstrate the prototype in use, we present two flavors of a familiar example: sequential and lock-free concurrent versions of a monotonic counter. Figure 6.3 gives a sequential version, and Figure 6.4 gives a lock-free version.

The module `MonotonicCounter` in Figure 6.3 defines three functions. The first, `stable_monocount` is a stability proof in the style of Section 6.2.2, with explicit Liquid Haskell type annotation to guide inference. The second and third functions should be familiar: `alloc_counter` allocates a new counter, and `inc_counter` increments a monotonically increasing counter. In

```
module LockfreeMonotonicCounter where

import RG
import MonotonicCounter

{-@ 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) stable_monocount
```

Figure 6.4: A lock-free increment for a monotonically increasing counter implemented in Liquid Haskell + Liquid RGREFs.

the latter case, the use of `modifyRGRef` typechecks because Liquid Haskell infers that the increment operation has type

$$x : \{v : Int \mid v > 0\} \rightarrow \{v : Int \mid x <= v\}$$

which relates the input and output according to the inferred increment relation. The module `LockfreeMonotonicCounter` in Figure 6.4 defines a nearly identical increment function, using instead the `atomicModifyRGRef` primitive that wraps `atomicModifyIORef`. This atomic primitive produces an IO action with a compare-and-swap in a loop, producing executions just like those in Chapter 5's atomic counter increment.

6.3.2 Past and Terminal Values

It is often useful to use past observations to infer information about the current program state. To support this, Chapter 5 introduced a notion of *refiners*, presented as `observe-field` in the COQ DSL. These constructs were used to simultaneously observe the value stored in some heap cell (or a field thereof) and to use that observation to introduce new stable predicates over the reference used to access the heap. In 5.3.1, these were used to observe immutable field values, and lower bounds on field values. We can expose the same capabilities in Liquid RGREFs.

Figure 6.2 defines an uninterpreted proposition `pastValue` and an uninterpreted function

`terminalValue` to support this reasoning.[10] The `readRGRef` primitive tags each value read from any `RGRef r` with the refinement predicate `pastValue r`, indicating that the value was observed in the heap cell referenced by `r` at some point in the program's execution. Unlike the refiners from Chapter 5, this is essentially a temporal assertion over the program history, as opposed to a primitive for directly and immediately inducing new stable predicates. Essentially, the `pastValue` refinement separates the observation from its stable implications.

Instead of immediately producing a new refinement,[11] we rely on the axiom `axiom_pastIsTerminal` in places we need consequences of the past observation (we explain how to use Liquid Haskell axioms in Section 6.4). This axiom accepts a reference and appropriately refined past value of that heap cell, along with a proof that once the referent has that particular value, it will retain that value permanently under the combined rely-guarantee relation. In exchange the axiom allows two conclusions: first that the observed heap cell has *reached* its terminal value (we take `terminalValue` as a partially-defined predicate), and second that the program has already observed the terminal value.

For an arbitrary value bound to a variable, this is not particularly useful unless the transition relation is the identity relation (e.g., the reference contents are immutable). But we will see in Section 6.4 that combining this with Liquid Haskell's pattern matching refinement allows more nuanced conclusions.

6.3.3 Implementation Availability

Our implementation and the source code for our examples in Section 6.4 are available online:

`https://github.com/csgordon/rghaskell/`

---

[10] The `qualif` declarations of `PastValue` and `TerminalValue` are Liquid Haskell predicate export statements.

[11] This could also be done in the prototype, but would require introducing a new reference aliased to the first with the new refinement. This is because our prototype sits above both Haskell and Liquid Haskell, and therefore does not control the type environment to perform the necessary strong update for the ideal model of this.

6.4 *Using Liquid* RGrefs

This section discusses our experience applying the Liquid RGref prototype to some actual Haskell code. We first discuss two examples for which we formally verified properties using Liquid RGrefs, then discuss examples representative of a significant number of Haskell applications of mutable state which appear straightforward but onerous or uninteresting to verify.

6.4.1 STM Undo Log

STM Haskell [163, 116] is an implementation of software transactional memory (STM) [236] for Haskell. The key concept in STM is the exposure of an *atomic* block, whose semantics are intuitively transactional: to run atomically with respect to other atomic blocks.[12] For executing an atomic block to have these semantics, STM implementations follow one of two approaches: either accumulate a log of reads and pending writes to memory which is then committed atomically using a protocol for n-ary CAS; or speculatively issue writes directly to memory while keeping an undo log, which is used to reverse effects when a conflict is detected with another thread's transaction.

Haskell's STM is implemented in C, but the Glasgow Haskell Compiler exports a fallback module for compiling STM code without STM support. Its semantics are to run each transaction directly, sequentially. But in the event of an exception or other runtime error mid-transaction, the earlier effects of the transaction must still be undone. So the fallback implementation maintains an undo log in the form of an `IORef` to an `IO ()`.

We have used Liquid RGrefs to verify that when the undo log is extended, reversal operations are added to the correct "end" of the IO action (i.e., the most recently performed actions are undone first, before running the rest of the undo log). To do so, we defined a relation `fwd_extends` on IO actions such that `fwd_extends y x` is provable in the SMT logic only if `y` is an IO action that performs some (possibly empty) set of actions before also performing the actions of `x`. It is introduced via a trusted refinement on the sequencing

[12] This explanation is oversimplifying significantly, as the actual consistency of transactions with respect to each other is surprisingly subtle [179, 33].

operator for IO actions.

6.4.2 Lockfree Linked List

One of the test cases for the Glasgow Haskell Compiler[13] is a lock free linked list similar to what would be seen in a more conventional programming language, along the lines of that originally proposed by Harris [117, 121].

To make the discussion of the verification itself more comprehensible, we first discuss lock-free linked list algorithms in general, then how Haskell features and implementation affect implementations of those algorithms, and finally conclude by discussing an actual lock-free linked list with properties verified 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 references to the tail of the list. 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 [251] (Section 5.3.1), while appending a node is precisely the enqueue operation from a (tail-less) Michael-Scott queue [170].

The algorithms get more sophisticated once modification of *interior* links of the list is required, as for deletion, or sorted insertion [119, 118]. We will first discuss deletion, without which insertion is actually also the same as a Treiber stack push operation, just later in the list. 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.

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. Consider the case where a node d to be removed is located, and a thread is just about to execute a CAS instruction to swap the predecessor's next pointer to d's next

[13] `testsuite / tests / concurrent / prog003 / CASList.hs` in the compiler source tree.

pointer, unlinking d. Between the time d is located and the CAS executes successfully, the predecessor whose link is being updated could be removed from the list! Thus, the CAS could succeed, but update a reference that is no longer part of the actual list.

To address this issue, each node carries a flag indicating whether it has been logically deleted. The procedure for deletion then merely locates the desired node and sets the deleted flag, and any traversal of the list then performs physical removal of any logically deleted nodes found during traversal.

For physical deletion to be correct, an attempt should only succeed if it actually updates a predecessor that is still in the list (thereby succeeding in physical removal). One way to achieve this is to set up the data structure so the CAS on the predecessor's next pointer to skip over the deleted node succeeds only if the predecessor is still in the list. This requires that the CAS compare not only the next pointer, but also the deleted flag (always giving the unset option as the expected value for the deletion flag).

Mechanisms for this vary, from using a 2CAS,[14] 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 which relies on the standard memory layout of algebraic data types in functional programming languages.

This separation of physical and logical deletion is typically referred to as a *lazy* deletion, and forms the basis for a number of other lock free data structures, including sets [119, 118, 201], and is an implementation of one of the most powerful concurrency primitives [120] (atop an equally powerful primitive — CAS).

The lock free lazy list was actually one of the inspirations for the concurrent RGrefs in Chapter 5. O'Hearn et al. give criteria for applying a pair of lemmas for reasoning about execution traces from traversing the spines of these list-shaped lock-free data structures. The lemmas show the equivalence of finding a node still-undeleted in an interleaved execution and an uninterrupted traversal finding the same node, which is a powerful mechanism for proving linearizability. They characterize a variant of Heller's lock-free linearizable set [119, 118] in terms of invariants and two-state step invariants, some subset of which are necessary for

[14] Compare-and-swap on two adjacent (typically aligned) machine words of memory. Most architectures, e.g. x86/amd64 [64] include a double-width CAS.

applying the lemmas about link traversal history. These invariants and two-state invariants are in fact predicates and (symmetric) rely-guarantee relations[15] when viewed through the use of RGREFs.

Compare-and-Swap in Haskell

In this GHC test case, CAS is performed not on a field of the node, but on the *whole node*, because the nodes themselves are actually immutable in Haskell, so a CAS on the whole object amounts to a CAS on a pointer to an immutable record.

Another subtlety is that Haskell's runtime system makes aggressive use of immutability assumptions — particularly the garbage collector. Specifically, GHC's garbage collector relaxes its GC barriers to allow occasional duplication of immutable data,[16] which can cause uses of a raw hardware CAS to fail more often, if for example the GC duplicates the expected-value reference. To address this, Newton implements a ticket system in `Data.Atomics` which wraps values compared in a CAS in such a way as to prevent the problematic optimizations. In principle we could give trusted annotations for his ticket system, but here we stay close to the original test case.

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 pseudocode given for CAS, except it uses the equality comparison from the `Eq` typeclass (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

[15] A couple need to be slightly reformulated, but most are directly expressible.

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

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. In principle, we could extend our prototype to handle Newton's ticket system, but have not done so.

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.

To prove that the lock-free linked list followed the appropriate protocol — as far as we could express it — we followed a simple development methodology:

1. Define logical (SMT) field accessors (Liquid Haskell measures) for various fields.[17]

2. Define the rely/guarantee relation for interior pointers (we already knew from previous work the protocols involved).

3. Convert `IORef`s to `RGRef`s.

4. Annotate the definition of list nodes to apply the rely/guarantee relation to interior pointers.

5. Where inference fails to infer sensible refinements, factor out use of key primitives to explicitly annotate their refinement types.

6. Where inference still fails, use `undefined`.

7. Where Liquid Haskell verification of mutations fails, examine errors and if necessary use `liquidAssume` (explained shortly) to apply axioms about the contents of immutable heap cells.

```
{-@
data List a = Node (node_val :: a)
                (node_next :: ((RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a))))
            | DelNode (del_next :: (RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a)))
            | Null
            | Head (head_next :: (RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a)))
@-}
data List a = Node a (UNPACK(RGRef (List a)))
            | DelNode (UNPACK(RGRef (List a)))
            | Null
            | Head (UNPACK(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
-- Wrap rgCAS with the refinements made concrete, to help inference
{-@ rgListCAS :: Eq a => RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a) ->
                old:(List a) -> new:{v:(List a) | (ListRG old v)} -> IO Bool @-}
rgListCAS :: Eq a => RGRef (List a) -> List a -> List a -> IO Bool
rgListCAS r old new = rgCAS r old new any_stable_listrg
```

Figure 6.5: Excerpt from a lock-free linked list implemented using Liquid RGREFs: definitions.

We imagine this would be approximately the standard routine for converting existing Haskell code to use Liquid RGRefs.

Figure 6.5 gives some key 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 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. Not shown are a set of measures, `isNull`, `isNode`, `isDel`, and `isHead`, which are true for the corresponding constructor and otherwise false. In proving the guarantee later, the SMT solver will need to know that being a `DelNode` is mutually exclusive with being any other variety of node, so we also declare the axiom `isDelOnly`, which states this property for deleted nodes. 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. To aid type (refinement) inference, we also wrap the underlying rgCAS operation with an explicit typing.

The only remaining part of the figure is the `ListRG` predicate itself, which 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 6.5:

1. Appending a node: replacing a `Null` node with a `Node`. In the *deep* rely-guarantee reference systems from Chapter 4 and 5, we would also require that at the time of update, the new node's next field pointed to another null node, but this system lacks support for the sharing idiom required to prove such an obligation. Our implementation of `enqueue` for a Michael-Scott queue [170] (Section 5.5) does exactly this.

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

[17] Currently this requires converting records projected in predicates into algebraic data types because Liquid Haskell does not support measures on records.

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.

One unfortunate limitation to the current verification is that the allocation of a new `Null` node uses `undefined` to force inference to conclude (as a consequence of $\bot$) that the rely is inhabited. This is unpleasant, but we view it as equivalent to the ideal of explicitly providing a relation argument to the `RGRef` allocation routine.

Figure 6.6 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[18] to a newly allocated `DelNode` preserving `curNode`'s next pointer (Line 15). 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 20). 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 24) or `Node` (Line 29) 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

[18]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 a compare-and-swap of a pointer to an immutable record.

```
1  delete :: Eq a => ListHandle a -> a -> IO Bool
2  delete (ListHandle head _) x =
3    do startPtr <- readIORef head
4       go startPtr
5     where
6       {-@ go :: RGRef<{\x -> (1 > 0)},{\x y -> (ListRG x y)}> (List a) -> IO Bool @-}
7       go prevPtr =
8         do do prevNode <- forgetIOTriple (readRGRef prevPtr)
9               let curPtr = myNext prevNode -- head/node/delnode have all next
10              curNode <- forgetIOTriple (readRGRef curPtr)
11              case curNode of
12                Node y nextNode ->
13                  if (x == y)
14                  then -- node found and alive
15                  do b <- rgListCAS curPtr curNode (DelNode nextNode)
16                        if b then return True
17                         else go prevPtr -- spin
18                  else go curPtr -- continue
19                Null -> return False -- reached end of list
20                DelNode nextNode ->
21                        -- atomically delete curNode by setting the next of prevNode to next of curNode
22                        -- if this fails we simply move ahead
23                       case prevNode of
24                         Node v _ -> do b <- rgListCAS prevPtr prevNode
25                                                   (Node v (liquidAssume
26                                                            (axiom_pastIsTerminal curPtr curNode
27                                                               (terminal_listrg curPtr curNode)) nextNode))
28                                        if b then go prevPtr else go curPtr
29                         Head _ -> do b <- rgListCAS prevPtr prevNode
30                                                  (Head (liquidAssume
31                                                         (axiom_pastIsTerminal curPtr curNode
32                                                           (terminal_listrg curPtr curNode)) nextNode))
33                                      if b then go prevPtr else go curPtr
34                         DelNode {} -> go curPtr -- if parent deleted simply move ahead
```

Figure 6.6: Excerpt from a lock-free linked list implemented using Liquid RGREFs: deletion.

terms of `terminalValue`. Recall from Section 6.3.2 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 `liquidAssume` (Section 6.1.2) to inject properties into the refinement of a relevant value. In this case, we use it to inject an instantiation of the `axiom_pastIsTerminal` axiom into the refinement of `nextNode`. This also relies upon a use of `terminal_listrg`, which generates the stability proof required for the axiom to convert a past observation into a terminal observation. 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.

6.4.3 HalVM Xen Ring Buffer

HalVM[19] is infrastructure to run Haskell programs directly on the Xen virtual machine monitor [16]. It includes just enough unsafe low-level code to get GHC's runtime system up and running in a virtual machine, along with a substantial library for interacting with the VMM itself.

One part of this is a ring buffer implementation[20] for use in writing device drivers in Haskell. A ring buffer is a bidirectional queue of pending work items (in this context, events to process), represented as a chunk of memory with head and tail slots which continue to move in one direction as requests are enqueued and dequeued, modulo wraparound (hence the name *ring* buffer). Xen uses these for paravirtualized device drivers. When some Xen devices must notify a virtualization-aware kernel (such as HalVM) of an interrupt, the data associated with that interrupt is enqueued into a ring buffer. When a virtualized guest such as HalVM must request an operation from a virtualized device, it writes a request into the request side of the ring. The ring itself is a paravirtualized hardware device, requiring low-level primitives to interact with it. HalVM's ring buffer implementation is

[19] `http://galois.com/project/halvm/`

[20] `https://github.com/GaloisInc/HaLVM/blob/master/src/XenDevice/Communication/RingBuffer.hs`

an abstraction over this, which deals with batching requests to hardware that may not yet fit in the virtualized device's buffer, and exposing a blocking routine to wait for responses from the hardware device (as well as a range of other functionality, such as initialization).

We have not verified HalVM's ring buffer using Liquid RGRefs, but have examined the code in detail to determine the feasibility of doing so. With a split `RGRef` (separate rely and guarantee), Liquid Haskell verification of some supporting HalVM code, *and an encoding of Hoare Logic into Liquid Haskell*, it would be possible to verify some safety properties for how one of the internal routines manages the pending queues (represented as Haskell streams) of requests to send to the hardware, and responses received from the hardware. Unfortunately, neither we nor the creators of Liquid Haskell have yet managed to construct a working Hoare logic over the IO or State monads in Liquid Haskell, despite some effort in this direction; the types are just a bit beyond the current Liquid Haskell tool's capabilities.

The primary ways other code interacts with the HalVM ring buffer is to call `frbWriteRequests` (to send requests to hardware) or `frbReadResponses` (to read one or more responses produced by the hardware). Internally, sending a request to hardware appends the requests to a stream of requests that have not made it to hardware yet, and calls a procedure `advanceFrontRingState` that interacts with hardware. Requesting responses sets up a potentially blocking notification mechanism, calls `advanceFrontRingState`, and then waits for a notification (which may have already arrived).

`advanceFrontRingState` follows a strict protocol for modifying the mutable state in the ring buffer's representation. The list of pending requests to the hardware is represented as an `IORef` to a stream of outgoing requests, which `advanceFrontRingState` must only pull from — it must only update the pending requests by pulling a prefix of the stream off, and replacing the old value with the remaining stream. Similarly for notifying waiters of responses from hardware, there is a queue of notification objects (`MVars`) which must be pulled from in a similar manner. Each of these is accompanied by a counter (`IORef Int`) which must only be incremented. The structure of the code, which requires intermingling read results from multiple references to calculate new values, means a Hoare logic is required to track values of heap cells across multiple heap interactions, and rewriting code to use primitives like `modifyRGRef` would be inadequate. So we take this as an example for how

the combination of a Hoare logic with rely-guarantee references would be useful.

6.4.4 Other Imperative Haskell Code

After many hours of searching code repositories for uses of mutable state — both via `IORef`s and `STRef`s — in Haskell, we were unable to locate other examples of mutation that were significantly different in character from the other examples discussed in this chapter. In particular, a surprising number of mutable state uses in Haskell were for monotonicity properties over a mutable reference to a functional data structure. This is something clearly handled well by RGrefs and Liquid RGrefs in particular, as shown by our monotonic counter and STM undo log verifications. (A significant number of state uses we found were, in fact, monotonically increasing counters.)

6.5 Related Work

General rely-guarantee reference related work has been explored exhaustively elsewhere in this dissertation, so here we focus only on very closely related treatments of state using refinement types.

The most similar treatment of state to the one presented here comes from F* [243, 241, 245]. F* uses the Dijkstra Monad [245], based on Dijkstra's predicate transformers [69] and described in more detail in Chapter 2 to reason about imperative state. The implementation includes a variant of the Dijkstra Monad, the `iDST` monad, indexed also by a (reflexive, transitive) binary relation on heaps. Updates to the heap in that monad are then required to respect the binary relation, presenting an environment that is essentially a rich Hoare logic augmented with a monotonicity condition resembling global rely-guarantee with the relations collapsed (and only sound for sequential programs). Among the primitives for manipulating this monad are operations for observing the current heap, witnessing that the current heap was the program heap at some point in time, and applying knowledge that a specific previous heap (with some useful property) was related by the monotonicity condition to the current heap to draw new conclusions. Earlier implementations also had an additional reference type constrained by a single local binary relation — an `mref` — very similar to the single-relation `RGRef` presented in this chapter. However, F*'s `mref` appears

not to be used to prove properties directly, with the exception of an axiom specialized to a relation that prohibits clearing a boolean flag. Thus there has been no use of reasoning similar to our `pastValue` and `terminalValue` (at reference granularity, though an axiom to that effect has been stated and left unused in F*), or to Chapter 5's general refiners. F*'s implementation of these ideas was contemporaneous with and independent of our design work. Unlike F*'s implementations, we may soundly apply them to fine-grained concurrent programs like Section 6.4.2's lock-free linked list, and in particular this chapter's version applies to a fairly widely used programming language. On the other hand, F* supports a working program logic, so an implementation of RGREFs in F* would be able to treat examples like the HALVM ring buffer.

Both we and F* exploit a very limited form of temporal reasoning over histories. This suggests a more general — temporal — specification logic for rely and guarantee relations may be useful. Indeed, Chapter 5's study of union-find had to weaken the guarantee for performing path compression to admit arbitrary set-preserving reparenting, because the most precise restriction would require the assertion that the new parent *was once an ancestor*, a temporal property over histories. Thus far, it is rare for full blown temporal logic to pay off, except for very sophisticated problems such as reasoning about memory reclamation for lock-free data structures [96, 108].

Another important point of comparison is between our `pastValue` and Turon et al.'s CaReSL [254, 253], which has a similar notion of observing that a heap cell has reached some particular point in a protocol and using that as a form of lower-bound.

6.6 Fruitful Directions for Further Investigation

The most natural direction for future work, as evidenced by the HALVM ring buffer example, is the combination of (Liquid) RGREFs with some form of Hoare logic. This is useful because occasionally reading a reference and updating it *must* be interleaved with other state operations, and knowledge of past reads must be carried across those operations in order to prove an update has a specific relationship to the overwritten value.[21]

[21]The astute reader may have noticed that the intended use for this logic is to carry information about the exact contents of a heap cell, even when those contents are *not* preserved by a reference's rely. Such a

The other direction we should explore going forwards is application of RGrefs to a more mutation-heavy language. Haskell, through its practice of monadically encapsulating imperative code, essentially discourages use of mutable state except when the benefit is substantial. This is part of what leads to the prevalence of mutable references to purely functional (persistent) data structures in Haskell code. While this is idiomatic in Haskell, this pattern is quite powerful and quite Haskell-specific. In the concurrent setting, updating such a reference using only CAS or `atomicModifyIORef` yields an implementation of Herlihy's universal construction of lock-free data structures [120]! This is very much not how data structures, sequential or even lock-free, are typically implemented in languages like C or Java. Support for such languages would likely require a form of reasoning about non-atomic state updates; for example, an imperative splay tree implementation requires an operation that *must* break the tree apart into multiple segments and reassemble them. Permitting this in current full RGrefs would require weakening the rely and guarantee on the internal references enough to permit each individual update, which fails to preserve most properties. Most likely some notion of Hoare logic would be useful here as well, using the logic only across a number of non-atomic updates, and using the pre- and postcondition for some span to then prove the updates in aggregate respect some guarantee.

6.7 Conclusions

This chapter has shown first that RGrefs can be made more usable by integrating with SMT-based verification tools for standard refinement types. We have also shown that the simplification we made is actually quite useful and adequate for many uses of mutable state in Haskell. While we have certainly not examined all Haskell programs in use today, the fact that many hours searching even Haskell-specific code repositories did not yield examples fundamentally beyond the capabilities of the Liquid RGref design presented here suggests we capture at least the most common idioms for stateful programming in Haskell. Finally, the fact that our implementation is fairly concise, and requires only modest encodings

---

logic would clearly be unsound for concurrency, but the cases this logic would be useful are for thread-local data: HalVM's ring buffer and the Text module's data structures are not safe to access from multiple threads.

to embed into Liquid Haskell shows that RGREFs are a natural complement to standard refinement types.

Chapter 7

DIRECTIONS FOR FUTURE WORK

Throughout this dissertation, we have noted various limitations and useful extensions to our work on reference-based characterization of interference. Here we collect those limitations and extensions together, and elaborate on promising avenues for this future work: a number of extensions and modifications to rely-guarantee references to increase expressivity or simplify specification, as well as other avenues for gradual verification.

7.1 Direct Extensions to RGrefs

Thus far we have laid out foundations for reference-based mutation control, interpreted in a certain manner: all capabilities for mutation are explicitly identified with references at runtime, and *mutation* is taken literally to mean restrictions on runtime state. These choices were made in order to appeal to developer intuition about treating references as capabilities for mutating runtime state, and to explore the limits of a limited number of concepts. Breaking with either or both of the choices will yield more powerful systems, at the cost of making specification and automation more complex. The extensions we consider here are substructural capabilities (Section 7.1.1), ghost state (Section 7.1.2), explicit static region abstractions (Section 7.1.3), recovering stronger references after splitting and weakening (Section 7.1.4), and alternative specification languages (Sections 7.2 and 7.2.1).

7.1.1 Capabilities Divorced from Values

Tightly coupling permissions for state change with concrete values that are necessary at runtime to effect state change is a natural starting point. However, it has a weakness that exchanging capabilities requires exchanging values, and asymmetry of information is difficult to represent.

We can take a note from work on Deny-Guarantee [73], Concurrent Abstract Predicates

(CAP) [71, 242], and derivatives [254, 253]. These systems have notions of sets of capabilities (to regions of memory) each ascribing different possible actions to the threads holding specific unique-per-region capabilities. Notably these systems reason not only about the presence of capabilities when performing actions, but internalize reasoning that possessing a capability locally denies its actions to other threads. This makes developing structures such as mutual exclusion locks relatively straightforward. When a thread that acquires a lock using CAS it also acquires a capability for performing the unlock action. Because other threads cannot also possess the unlock action, the knowledge that the lock is held becomes stable for the holding thread, which can then enable further actions on the guarded state. Related capability systems [216, 13, 145] derive similar expressiveness, sometimes by restricting permissions to state forming a natural monoid [145].

Other sources of inspiration include Ley-Wild and Nanevski's subjective auxiliary state [156]. This is a technique for concurrent program logics that adds an explicit dichotomy between thread-local and environmental (other threads) contributions to shared state that forms a partial commutative monoid. With strong updates to these assertions, it is possible to represent asymmetric knowledge such as ownership of a lock.

Adding linear capabilities in the style of CAP [71, 242] or Mezzo[216, 13] should be a fairly direct addition to the system, with clear gain in expressiveness. Adding more novel techniques for asymmetric knowledge, such as subjective reasoning, would be a more invasive change because it would require changing the formulation of rely-guarantee relations and predicates. This would likely yield a more expressive system than simply adding linear capabilities, but it is unclear how the cost-benefit ratio compares to linear capabilities.

7.1.2 Ghost State

The most direct interpretation of *mutation control* is to restrict mutation of runtime state. But other correctness concerns can be encoded using *ghost state* or *auxiliary state*. Some examples of this is the use of auxiliary variables in program logics [36, 156]; ghost state for mathematical entities to relate abstract states to physical representation [152]; or synchronization protocol information like lock levels [151].

We informally explored this idea in earlier work [102]. We designed a system for statically ensuring deadlock freedom, based around the idea that acquiring one lock may grant exclusive permission to acquire additional locks. With a few restrictions on the shape of the capability-granting relation, this ensures deadlock freedom. To enforce this discipline statically, we associate a static capability to each lock, and a distinguished reference to each lock usable for changing the capability for acquiring the lock. Other references to the locks can exist, but are usable only for acquiring a thread's first lock. This is another instance of mutation control, but for ghost state, and the system employs concepts similar to impoverished rely-guarantee reasoning to ensure a consistent view of lock-capability associations.

Further applications of mutation control to ghost state could include partial solutions to other limitations, such as adding ghost references to permit relations to constrain the relationship between heap segments that do not point to each other at runtime.

7.1.3 Region Granularity

For more expressive forms of reference-based interference specifications, it is sometimes the case that the reference-reachable heap fragment is so narrow a focus for the specification as to become awkward. In general, folding guarantees in particular can produce some very complex relations. Despite the advantages of specifying interference on a per-reference basis, sometimes it is preferable to specify interference over some other conceptual or explicit statically-identifiable *region* [246, 249, 109, 244] of the heap. For example, the system in Chapter 5 makes it difficult to verify lock-free algorithms that use helping,[1] because there are no pointers between the primary data structure and the auxiliary helping structures, so enforcing relationships between them would be difficult without region-granularity reasoning.

Systems such as LRG [82] and CAP [71, 242] segment the heap into statically identified regions, specifying interference over the contents of each region. CAP explicitly names

[1] *Helping* is a technique used in lock-free data structures to improve performance. A data structure is augmented with a place for threads to record the operations they are attempting, and one thread may sometimes complete another thread's proposed action (*help* the other thread) when it helps the first thread as well, thereby increasing performance by reducing contention on the main portion of a data structure. A classic example of this is the elimination stack [121], where a thread pushing onto a stack may simply cancel out the action of some other delayed thread that simultaneously attempts to pop from the stack.

regions, while LRG uses precise (in the concurrent separation logic sense [200, 258]) invariants to delimit regions, which may then be split and merged along with the rely and guarantee relations over those regions. This dynamic splitting and merging carries some of the advantage of RGREFs' ability to break up regions more granularly, but allows more flexibility in choosing region decomposition; RGREFs essentially have the region breakdown for each specification fixed in the metatheory. By contrast, LRG supports only *disjoint* regions. In some ways RGREFs solve the harder problem of permitting multiple interference specifications framed in terms of *overlapping* regions to soundly coexist. An ideal system would permit both the developer-chosen decomposition of regions and sound coexistence of overlapping regions (specifically, coexistence of user-defined regions with reference-based specifications).

It is not obvious how to best extend RGREFs to support these features together, though it seems clear that on occasion it would be desirable to create a new interior pointer into a data structure tied to region-based specifications rather than the parent pointer's rely and guarantee. While it is likely that the folding approach described in Chapter 4 could be extended naïvely to support this coexistence, the result would likely be quite difficult to use effectively. Two pieces of related work offer useful suggestions.

Hobor and Villard's study of *ramification* [128] — the non-local impact of local updates — in the context of separation logic suggests a useful style of reasoning. Rather than explicitly weakening permissible updates,[2] they develop a (derivable) rule for separation logic and some theory of non-local reasoning for common data structures. The key to their approach is novel application of separating implication $A \mathrel{-\!\!*} B$: the assertion that if the current heap is extended with a heap satisfying A, then the total heap would satisfy B. Something similar to this is possible to formulate in RGREF specifications, but we have not explored its use. Folding and containment could be seen as ad hoc semantic approaches to ramification, though the exact relationship is unclear because Hobor and Villard do not consider relational specifications. It is possible that studying their derived RAMIFY rule (explained in Chapter 2) in the context of a separation logic with binary postconditions

[2]This is actually a non-issue in their system since they use standard separation logic, but would be if adapting the ideas to RGREFs in the most naïve manner.

(e.g., [190]) could yield insights.

Nanevski's recent work on concurroids [187] offers another approach, also in the setting of a concurrent separation logic. A concurroid combines subjective monoids in the style of Ley-Wild et al. [156] with protocols as state transition systems. A key novelty of the work is their *entanglement* operator on concurroids, to build larger concurroids out of smaller ones, which is intuitively similar to the notion of separating conjunction in LRG [82], but with uses of auxiliary state replaced by subjective reasoning [187]. Unlike LRG, concurroids cannot express asymmetric restrictions between threads; e.g., forking threads where one has increment permissions to a counter, but the other thread has simply read-only permissions (Nanevski et al. give a derivation of a thread's rely and guarantee from a concurroid, but not vice versa). Adapting their entanglement operator to rely-guarantee references would likely be non-trivial; as with LRG's separating conjunction on rely- and guarantee-relations, they require the state governed by entangled concurroids to be disjoint.

7.1.4 Splitting, Merging

In the systems of Chapters 4 and 5, the aggregate program-wide permissions to a heap fragment degrade over time — the program may lose the ability to perform some operations, but can never recover them. By contrast, Chapter 3's reference immutability system includes rules for *recovering* isolated or immutable permissions from the weaker writable and readable permissions. Extending this for RGREFs would be very useful: it would enable a pattern of taking a reference with a strong refinement, temporarily splitting it with weaker permissions to perform some updates efficiently or express updates more naturally while preserving some strong property, then recovering the original reference's rely and guarantee, and splitting again with *different* permissions. An example is alternating access to a collection between adding and removing elements, tracking upper and lower bounds on contents. There are two natural approaches to supporting this.

Scoped Recovery Chapter 3's recovery mechanism relies on restricting permissions on inputs to a particular lexical scope. It then exploits the fact that those input constraints permit fewer possible aliasing relationships between references with weaker permissions and

those with stronger permissions. Something similar could be added to RGREFs, but would likely be challenging to specify or use given the flexibility of using relations rather than fixed permissions. Something we have considered previously, but discarded due to its complexity, is using *in*compatibility to infer non-aliasing. This simpler check seems too complex to be worthwhile, and supporting full recovery would require additional semantic checks to reason about where aliases to a hypothetically recovered reference might be stored.

Explicit Splits and Merges An alternative is to use a technique similar to Boyland's fractional permissions [41] or Bornat's counting permissions [36] to track how many times a given reference had been split, with some static representation of the original rely and guarantee. Then collecting all of those aliases could soundly restore the original relations.

7.2 Richer Specification Languages

The most sophisticated specifications explored in this work are essentially unary and binary relations on state transitions, along with some carefully chosen support for reasoning about the relationship between states further separated in time. Clearly this is useful, but we have seen an example where expressing the correct property required more sophisticated temporal reasoning.

Chapter 5's union find example simplified the guarantee relation to permit any set-membership-preserving update to a union find structure. The most precise guarantee would, in the case of updating node's parent for path compression, require that the new parent be a past ancestor of the modified element.

The classic temporal logics, such as LTL [97, 211], CTL and CTL* [19, 56], or TLA [148, 147] reason over future states, but not the past. Recent work from the concurrent program logic community constructs rely-guarantee-style expositions of a temporal logic of execution *history* to reason about when explicit memory deallocation is safe in a lock-free data structure [96, 108]. Safely reclaiming memory in a fine-grained concurrent setting is notoriously subtle [121], so the fact that these techniques are effective here is impressive.

Adding temporal logic increases the sophistication required to use RGREFs so substantially that it is unlikely to be that useful to users. It would still be useful, however. Rather

than using it as a user-facing extension, it could be useful for proving soundness of other temporal extensions to the current RGref design, by way of embedding. In particular, extensions similar to the refiners and field observations (Chapter 5) and the `terminalValue` construct (Chapter 6) may be useful, and a direct embedding would make it easier to experiment with such extensions.

7.2.1 Intermediate Assertion Languages

Chapter 6 explores a weaker form of rely-guarantee references than that given in Chapters 4 and 5. This language seems more manageable — it is closer to the logic undergraduate computer science majors are taught in a discrete mathematics class — but still complex. The specifications expressible by reference immutability (Chapter 3) are considerably simpler, though they verify correspondingly weaker properties. Our intuition tells us there must be a useful specification language between the two. A key reason for the simplicity of reference immutability is not only the simpler rely and guarantee, but the fact that the same specifications apply to references to any type. By contrast, RGrefs require a predicate and relations that depend on the specific storage type of the reference. This alone necessitates a more sophisticated assertion language. The key to enabling additional type-agnostic rely and guarantee relations is to investigate additional type-agnostic properties, such as reachability. Actually finding such properties will require further experience applying RGrefs to a wide variety of programs.

7.3 Gradual Verification

A secondary aspect of this dissertation is its exploration of gradual verification: the idea that it is possible to verify strong properties of data accessed in some parts of the program without imposing any other restrictions on the rest of the program. They key to this is having a verification approach that can express not only the desired specifications, but also give a type that corresponds to what is normally considered "unverified" code. Reference immutability, RGrefs, and refinement types in general satisfy this criteria since refinements (predicates) and relations can trivially be instantiated to essentially $\top$. Other verification approaches such as separation logic do not satisfy this criteria, because even statically

accepting a heap access requires some (possibly existentially quantified) knowledge of the dereferenced cell's contents. Thus they can verify components in isolation, but then have no way to ensure that the environment of the verified code actually respects the verification assumptions such as preconditions.

Other verification approaches have the essential ingredient — the ability to ascribe an uninformative but sound "type" to "unverified" code — but the applications are unclear. For example, temporal logic can clearly verify properties of a prefix or postfix of an execution (depending on the temporal connectives) and many of those results are preserved by pre- or post-composition with further execution (naturally, formulas such as F — always eventually F — are not preserved by certain post-compositions). Exploring gradual verification in systems other than refinement types seems a promising approach to producing more pragmatic verification systems.

Chapter 8

CONCLUSION

This dissertation has proposed a family of techniques for program verification based on characterizing how individual references in a program may be used to interfere with each other. Through presentation of four iterations on this theme, crossing weak properties (read-only behavior) and strong properties (invariants and functional correctness), sequential and concurrent programs, and backed by a number of prototype implementations, we have established our original thesis:

> Reasoning about *interference* between individual *aliases* is a useful and powerful verification approach, and reduces the gap between verification of sequential and concurrent programs.

We have shown how starting from a weak characterization of interference between references in sequential programs, we can make only modest changes — adding two simple typing rules for introducing concurrency — to go from sequential reasoning about read-only behavior to concurrent reasoning about data race freedom (Chapter 3). We have shown that we can generalize the original weak characterization for more precise descriptions of interference (Chapter 4). We have shown that in this more expressive setting, we can again make only modest changes — weakening two rules and adding a rule for using a hardware concurrency primitive — to go from proving invariants on sequential data structures to proving invariants of sophisticated lock-free data structures (Chapter 5). And from this expressive basis, we have shown how to derive a proof technique for functional correctness (Chapter 5), and a slightly simplified version that is still useful for real Haskell programs (Chapter 6).

Together, these contributions and the prototype implementations backing each (including one built by Microsoft) demonstrate the great flexibility and utility in reasoning about interference between individual aliases. More importantly, the small transitions between

the sequential and concurrent variants of these systems demonstrate that, in contrast to decades of assumptions to the contrary, sequential and concurrent program verification do not need to be fundamentally different.

BIBLIOGRAPHY

[1] Agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Cited on page 44.

[2] Univalent Foundations of Mathematics, 2011. Invited talk at WoLLIC 2011 18th Workshop on Logic, Language, Information and Computation. Cited on page 47.

[3] Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006. Cited on pages 13, 49, and 207.

[4] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering (ICSE)*. IEEE, 2002. Cited on page 2.

[5] Paulo Sérgio Almeida. Balloon Types: Controlling Sharing of State in Data Types. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997. Cited on page 107.

[6] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A Categorical Semantics for Inductive-Inductive Definitions. In *Algebra and Coalgebra in Computer Science*, 2011. Cited on pages 122 and 143.

[7] Richard J. Anderson and Heather Woll. Wait-Free Parallel Algorithms for the Union-Find Problem. In *Symposium on Theory of Computing (STOC)*, 1991. Cited on pages vi, 164, 165, 169, 173, and 174.

[8] Andrew W Appel. Verified Software Toolchain. In *European Symposium on Programming Languages and Systems (ESOP)*, 2011. Cited on page 280.

[9] Andrew W Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. Cited on pages 20 and 280.

[10] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007. Cited on page 20.

[11] Robert Atkey and Conor McBride. Productive Coprogramming with Guarded Recursion. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ICFP '13, 2013. Cited on pages 142 and 145.

[12] Lennart Augustsson. Cayenne — A Language with Dependent Types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1998. Cited on page 129.

[13] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. Type Soundness and Race Freedom for Mezzo. In *International Symposium on Functional and Logic Programming (FLOPS 2014)*, June 2014. Cited on pages 22 and 244.

[14] Henk Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science, Volume 2*. Oxford University Press, Inc., 1993. Cited on pages v, 3, 128, 130, 139, 143, and 279.

[15] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Cambridge University Press, 2013. Cited on page 47.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003. Cited on page 237.

[17] Michael Barnett, Robert DeLine, Manuel Fähndrich, K Rustan M Leino, and Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004. Cited on page 15.

[18] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Communications of the ACM (CACM)*, 54(6):81, June 2011. Cited on pages 25, 103, 150, and 153.

[19] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The Temporal Logic of Branching Time. *Acta Informatica*, 20(3):207–226, 1983. Cited on page 248.

[20] Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal. Charge! A Framework for Higher-Order Separation Logic in Coq. In *Interactive Theorem Proving (ITP)*, 2012. Cited on page 44.

[21] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects (FMCO)*, 2006. Cited on page 43.

[22] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004. Cited on pages 44 and 133.

[23] Yves Bertot and Vladimir Komendantsky. Fixed Point Semantics and Partial Recursion in Coq. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, 2008. Cited on page 46.

[24] Kevin Bierhoff and Jonathan Aldrich. Modular Typestate Checking of Aliased Objects. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2007. Cited on pages 4, 12, 13, 26, 110, and 150.

[25] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI-hyperdoctrines, Higher-Order Separation Logic, and Abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):24–es, August 2007. Cited on pages 20 and 23.

[26] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause 'n' Play: Formalizing Asynchronous C♯. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012. Cited on pages 57 and 96.

[27] Adrian Birka. Computer-Enforced Immutability for the Java Language. Technical Report MIT-LCS-TR-908, MIT Program Analysis Group, 2003. Cited on page 30.

[28] Adrian Birka and Michael D. Ernst. A Practical Type System and Language for Reference Immutability. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 26–28, 2004. Cited on pages 9, 13, 29, 30, 49, 106, 112, 125, and 190.

[29] L. Birkedal, R.E. Mogelberg, J. Schwinghammer, and K. Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *IEEE Symposium on Logic in Computer Science (LICS)*, June 2011. Cited on pages 142 and 145.

[30] L. Birkedal and R.E. Møgelberg. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *IEEE Symposium on Logic in Computer Science (LICS)*, June 2013. Cited on pages 142 and 145.

[31] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed Kripke Models over Recursive Worlds. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2011. Cited on page 20.

[32] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. A Simple Model of Separation Logic for Higher-order Store. In *Automata, Languages and Programming*, pages 348–360. Springer, 2008. Cited on page 20.

[33] C. Blundell, E.C. Lewis, and M.M.K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), 2006. Cited on page 228.

[34] Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe Nondeterminism in a Deterministic-by-default Parallel Language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2011. Cited on pages 28, 49, and 111.

[35] Robert L. Bocchino, Mohsen Vakilian, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A type and effect system for deterministic parallel Java. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2009. Cited on pages 13, 28, 49, 57, 110, and 113.

[36] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission Accounting in Separation Logic. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2005. Cited on pages 13, 19, 25, 28, 117, 207, 244, 248, and 280.

[37] Richard Bornat, Cristiano Calcagno, and Peter W. O'Hearn. Local Reasoning, Separation and Aliasing. In *SPACE*, 2004. Cited on page 19.

[38] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2002. Cited on pages 32, 107, 108, and 207.

[39] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2001. Cited on pages 32, 107, 108, and 207.

[40] John Boyland. Alias Burying: Unique Variables without Destructive Reads. *Software Practice & Experience*, 31(6), 2001. Cited on page 105.

[41] John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS)*, San Diego, CA, USA, 2003. Cited on pages 13, 26, 27, 68, 110, 248, and 280.

[42] John Tang Boyland and William Retert. Connecting Effects and Uniqueness with Adoption. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005. Cited on pages 55 and 110.

[43] Stephen Brookes. A Semantics for Concurrent Separation Logic. In *Concurrency Theory (CONCUR)*, 2004. Cited on pages 13, 19, 24, 25, 38, 206, 207, and 280.

[44] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. Step-Indexed Kripke Model of Separation Logic for Storable Locks. *Electronic Notes in Theoretical Computer Science*, 276:121–143, September 2011. Cited on pages 19 and 24.

[45] Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM)*, 2011. Cited on page 43.

[46] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. In *Static Analysis Symposium (SAS)*, 2007. Cited on page 43.

[47] Venanzio Capretta. A Polymorphic Representation of Induction-Recursion. `http://www.cs.ru.nl/~venanzio/publications/induction_recursion.pdf`, 2004. Cited on page 143.

[48] Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2005. Cited on pages 16, 48, and 152.

[49] Renato Cherini and Javier O. Blanco. Local reasoning for abstraction and sharing. In *ACM Symposium on Applied Computing (SAC)*, March 2009. Cited on page 20.

[50] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, August 2012. Cited on pages 44 and 206.

[51] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013. Cited on pages 142 and 280.

[52] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, August 2009. Cited on pages 23, 44, 48, 119, 142, 146, 152, and 206.

[53] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic (JSL)*, 5(2):56–68, 1940. Cited on page 46.

[54] Dave Clarke, Sophia Drossopoulou, and James Noble. Aliasing, Confinement, and Ownership in Object-Oriented Programming. In *ECOOP 2003 Workshop Reader*, 2004. Cited on pages 32 and 107.

[55] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal Ownership for Active Objects. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008. Cited on page 109.

[56] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, April 1986. Cited on page 248.

[57] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics (TPHOL)*. 2009. Cited on page 153.

[58] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. In *Computer Aided Verification (CAV)*. 2010. Cited on page 153.

[59] Coq Development Team. The Coq Proof Assistant Reference Manual: Version 8.4, 2012. Cited on pages 44 and 119.

[60] Thierry Coquand. Metamathematical Investigations of a Calculus of Constructions. Technical report, INRIA, 1989. Cited on pages 3, 43, 47, and 279.

[61] Thierry Coquand. Infinite Objects in Type Theory. In *Types for Proofs and Programs*, pages 62–78. Springer, 1994. Cited on page 47.

[62] Thierry Coquand and Girard Huet. The Calculus of Constructions. *Information and Computation (I&C)*, 76(2-3):95—-120, 1988. Cited on pages 3, 43, 47, 128, 177, and 279.

[63] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. Cited on page 173.

[64] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*. 2014. Version 051. Cited on page 230.

[65] Dave Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *Workshop on Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, September 2007. Cited on pages 32, 107, and 108.

[66] Rob DeLine and K. Rustan M. Leino. BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005. Cited on page 15.

[67] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with Rep Exposure. Technical report, Digital Equipment Corporation, 1998. Cited on pages 4 and 12.

[68] Werner Dietl, Sophia Drossopoulou, and Peter Muller. Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007. Cited on pages 32, 43, 52, 59, 107, and 113.

[69] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM (CACM)*, 18(8):453–457, August 1975. Cited on pages 23, 206, and 239.

[70] T Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional Reasoning for Concurrent Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013. Cited on pages vi, 7, 9, 25, 38, 50, 71, 74, 75, 165, 183, 184, 191, 207, and 280.

[71] Thomas Dinsdale-young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010. Cited on pages 7, 19, 22, 37, 38, 39, 151, 165, 167, 207, 208, 244, 245, and 280.

[72] Dino Distefano and Matthew J. Parkinson J. jStar: Towards Practical Verification for Java. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2008. Cited on page 43.

[73] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-Guarantee Reasoning. In *European Symposium on Programing Languages and Systems (ESOP)*, 2009. Cited on pages 7, 13, 19, 36, 150, 151, 207, 243, and 280.

[74] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-theoretic Semantics. In Gerard Huet and Gordon Plotkin, editors, *Logical Frameworks*. Press Syndicate of the University of Cambridge, 1991. Cited on page 47.

[75] Peter Dybjer. Inductive Families. *Formal Aspects of Computing*, 6:440–465, 1994. Cited on page 132.

[76] Peter Dybjer. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *The Journal of Symbolic Logic (JSL)*, 65(2):525–549, 2000. Cited on page 47.

[77] Peter Dybjer and Anton Setzer. Indexed Induction-Recursion. In *Proof Theory in Computer Science*. 2001. Cited on page 143.

[78] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying Linearizability Proofs with Reduction and Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010. Cited on pages 198, 199, and 204.

[79] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A Calculus of Atomic Actions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009. Cited on pages 198, 199, and 204.

[80] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002. Cited on pages 13 and 110.

[81] Manuel Fähndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software*, 2011. Cited on page 103.

[82] Xinyu Feng. Local Rely-Guarantee Reasoning. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2009. Cited on pages 7, 13, 19, 33, 35, 42, 43, 117, 151, 152, 153, 168, 205, 207, 245, 247, and 280.

[83] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *European Symposium on Programing Languages and Systems (ESOP)*, 2007. Cited on pages 7, 13, 19, and 35.

[84] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for Concurrent Objects. In *European Symposium on Programing Languages and Systems (ESOP)*, 2009. Cited on pages 42, 199, and 205.

[85] Cormac Flanagan and Martín Abadi. Object Types Against Races. In *Concurrency Theory (CONCUR)*, 1999. Cited on pages 13, 49, 109, and 207.

[86] Cormac Flanagan and Martín Abadi. Types for Safe Locking. In *European Symposium on Programing Languages and Systems (ESOP)*, pages 91–108, Amsterdam, The Netherlands, 1999. Cited on pages 13, 49, 109, 126, and 207.

[87] Cormac Flanagan and Stephen N Freund. Type-Based Race Detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000. Cited on pages 13, 49, 109, 126, and 207.

[88] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting Purity for Atomicity. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2004. Cited on page 204.

[89] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. 2002. Cited on page 15.

[90] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003. Cited on pages 199 and 204.

[91] Robert W. Floyd. Assigning Meaning to Programs. In *Symposium on Applied Mathematics*, 1967. Cited on page 4.

[92] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear Regions Are All You Need. In *European Symposium on Programing Languages and Systems (ESOP)*, 2006. Cited on page 22.

[93] Matthew Fluet and Daniel Wang. Implementation and Performance Evaluation of a Safe Runtime System in Cyclone. In *SPACE*, 2004. Cited on page 23.

[94] Fredrik Nordvall Forsberg and Anton Setzer. Inductive-Inductive Definitions. In *Computer Science Logic (CSL)*, 2010. Cited on pages 47, 122, and 143.

[95] Tim Freeman and Frank Pfenning. Refinement Types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1991. Cited on pages 47, 115, 152, 164, 206, and 212.

[96] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning About Optimistic Concurrency Using a Program Logic for History. In *Concurrency Theory (CONCUR)*, 2010. Cited on pages 240 and 248.

[97] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the Temporal Analysis of Fairness. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1980. Cited on page 248.

[98] Philippa Gardner, Gian Ntzik, and Adam Wright. Local Reasoning for the POSIX File System. In *European Symposium on Programing Languages and Systems (ESOP)*. 2014. Cited on page 5.

[99] Herman Geuvers. The Church-Rosser Property for $\beta\eta$-reduction in Typed λ-calculi. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1992. Cited on page 139.

[100] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Universite Paris VII, 1972. Cited on page 222.

[101] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1—-102, 1986. Cited on page 21.

[102] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Static Lock Capabilities for Deadlock Freedom. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, January 2012. Cited on pages 21, 25, 207, and 245.

[103] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-Guarantee References for Refinement Types Over Aliased Mutable Data. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2013. Cited on pages 10, 40, 113, 151, 164, 168, 177, 180, 187, 196, 203, and 208.

[104] Colin S. Gordon, Michael D. Ernst, and Dan 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. Cited on pages 10, 113, 187, and 196.

[105] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2012. Cited on pages v, 10, 21, 23, 29, 45, 49, 103, 107, 113, 114, 118, 125, 126, 150, 151, 164, 168, 181, 184, and 190.

[106] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism (Extended Version). Technical Report MSR-TR-2012-79, Microsoft Research, October 2012. Cited on pages 10, 23, and 49.

[107] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2007. Cited on pages 19 and 24.

[108] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *European Symposium on Programing Languages and Systems (ESOP)*, 2013. Cited on pages 240 and 248.

[109] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 2002. Cited on pages 23 and 245.

[110] Christian Haack and Erik Poll. Type-Based Object Immutability with Flexible Initialization. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009. Cited on page 107.

[111] Philipp Haller. *Isolated Actors for Race-Free Concurrent Programming*. Phd thesis, École Polytechnique Fédérale de Lausanne, 2010. Cited on pages 21, 104, and 106.

[112] Philipp Haller and Martin Odersky. Capabilities for Uniqueness and Borrowing. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010. Cited on pages 12, 21, 43, 50, 55, and 109.

[113] Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM (JACM)*, 40(1):143–184, January 1993. Cited on pages 3, 47, and 278.

[114] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A Framework for Defining Logics. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1987. Cited on pages 3, 47, and 278.

[115] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, 1991. Cited on page 47.

[116] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Principles and Practice of Parallel Programming (PPoPP)*, 2005. Cited on page 228.

[117] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-lists. In *Distributed Computing (DISC)*, 2001. Cited on page 229.

[118] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, III Scherer, William N., and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems (OPODIS)*, 2006. Cited on pages 229 and 230.

[119] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A Lazy Concurrent List-based Set Algorithm. In *Principles of Distributed Systems (OPODIS)*, December 2005. Cited on pages 229 and 230.

[120] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991. Cited on pages 230 and 241.

[121] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. Cited on pages 40, 229, 245, and 248.

[122] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, July 1990. Cited on pages 41, 199, and 204.

[123] Stefan Heule, K. Rustan M. Leino, Peter Muller, and Alexander J. Summers. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013. Cited on pages 28 and 110.

[124] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. Actor Induction and Meta-Evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1973. Cited on pages 104 and 106.

[125] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with Safe Manual Memory-management in Cyclone. In *International Symposium on Memory Management (ISMM)*, October 2004. Cited on page 23.

[126] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)*, October 1969. Cited on pages 4, 12, 15, 117, and 279.

[127] Aquinas Hobor, Andrew W Appell, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programing Languages and Systems (ESOP)*, 2008. Cited on pages 19 and 24.

[128] Aquinas Hobor and Jules Villard. The Ramifications of Sharing in Data Structures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2013. Cited on pages 19, 23, 36, and 246.

[129] Martin Hofmann. *Syntax and Semantics of Dependent Types, in Semantics and Logics of Computation*, chapter 3. 1997. Cited on pages v, 128, and 155.

[130] W. A. Howard. The Formulae-as-Types Notion of Construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479—-490. 1980. Cited on pages 3 and 46.

[131] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and Checking of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, June 14–16, 2012. Cited on page 32.

[132] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, November 2012. Cited on pages 29, 32, 112, 113, 125, and 190.

[133] Galen C Hunt and James R Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41:37–49, 2007. Cited on page 21.

[134] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, March 2001. Cited on pages 12 and 17.

[135] G. Jaber, N. Tabareau, and M. Sozeau. Extending Type Theory with Forcing. In *IEEE Symposium on Logic in Computer Science (LICS)*, June 2012. Cited on pages 142 and 145.

[136] Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier, 1999. Cited on page 3.

[137] Jonas Braband Jensen and Lars Birkedal. Fictional Separation Logic. In *European Symposium on Programing Languages and Systems (ESOP)*, 2012. Cited on pages 38, 121, 165, 167, and 176.

[138] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying Functional Programs with Abstract Interpreters. In *Computer Aided Verification (CAV)*, 2011. Cited on pages 210, 214, and 221.

[139] C. B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, October 1983. Cited on pages 6, 13, 33, 34, 38, 114, 117, 153, 162, 183, 201, 207, 208, and 279.

[140] Ming Kawaguchi, Patrick Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic Parallelism with Liquid Effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012. Cited on pages 210, 214, and 221.

[141] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based Data Structure Verification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009. Cited on pages 210, 214, and 221.

[142] Ming Kawaguchi, Patrick M. Rondon, and Ranjit Jhala. Dsolve: Safety Verification via Liquid Types. In *Computer Aided Verification (CAV)*, 2010. Cited on pages 210, 214, and 221.

[143] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computing Systems (TOCS)*, 32(1):2:1–2:70, February 2014. Cited on page 5.

[144] Neelakantan R. Krishnaswami. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD Thesis, Carnegie Mellon University, 2012. Cited on page 20.

[145] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially Substructural Types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2012. Cited on pages 22, 167, and 244.

[146] Joachim Lambek and Philip J Scott. *Introduction to Higher-Order Categorical Logic*, volume 7. Cambridge University Press, 1988. Cited on page 3.

[147] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994. Cited on page 248.

[148] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Reading, 2002. Cited on page 248.

[149] John Launchbury and Simon L Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995. Cited on page 212.

[150] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994. Cited on page 212.

[151] K. Rustan Leino and Peter Müller. A Basis for Verifying Multi-threaded Programs. In *European Symposium on Programing Languages and Systems (ESOP)*, 2009. Cited on pages 110, 117, and 244.

[152] K Rustan M Leino and Rosemary Monahan. Dafny Meets the Verification Benchmarks Challenge. In *Verified Software: Theories, Tools, Experiements (VSTTE)*, 2010. Cited on page 244.

[153] K. Rustan M. Leino, Peter Muller, and Angela Wallenburg. Flexible Immutability with Frozen Objects. In *Verified Software: Theories, Tools, Experiements (VSTTE)*, 2008. Cited on pages 32 and 108.

[154] Xavier Leroy. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning (JAR)*, 43(4):363–446, 2009. Cited on page 280.

[155] Paul Blain Levy. Possible World Semantics for General Storage in Call-By-Value. In *Computer Science Logic (CSL)*, 2002. Cited on page 20.

[156] Ruy Ley-Wild and Aleksandar Nanevski. Subjective Auxiliary State for Coarse-grained Concurrency. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013. Cited on pages 43, 207, 244, and 247.

[157] Hongjin Liang and Xinyu Feng. Modular Verification of Linearizability with Non-fixed Linearization Points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013. Cited on pages 7, 41, 201, and 205.

[158] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM (CACM)*, 18(12):717–721, December 1975. Cited on pages 42, 198, and 204.

[159] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 1988. Cited on page 28.

[160] David MacQueen. Modules for Standard ML. In *ACM Symposium on LISP and Functional Programming*, 1984. Cited on page 2.

[161] David B. MacQueen. Using Dependent Types to Express Modular Structure. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 1986. Cited on page 2.

[162] Gregory Malecha and Greg Morrisett. Mechanized Verification with Sharing. In *Theoretical Aspects of Computing (ICTAC)*, 2010. Cited on pages 23 and 44.

[163] Simon Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. O'Reilly Media, Inc., 2013. Cited on page 228.

[164] Per Martin-Löf. Constructive Mathematics and Computer Programming. *Studies in Logic and the Foundations of Mathematics*, 104:153–175, 1982. Cited on page 3.

[165] Per Martin-Löf. An Intuitionistic Theory of Types. In *Twenty-Five Years of Constructive Type Theory*. 1998. Cited on pages 3, 43, 47, and 278.

[166] Per Martin-Lof and Giovanni Sambin. *Intuitionistic Type Theory*. Bibliopolis Naples, 1984. Cited on page 3.

[167] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight Linear Types in System F°. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2010. Cited on page 86.

[168] Nax Paul Mendler, Prakash Panangaden, and Robert L Constable. Infinite Objects in Type Theory. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1986. Cited on page 47.

[169] Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992. Cited on page 103.

[170] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Symposium on Principles of Distributed Computing (PODC)*, 1996. Cited on pages 164, 165, 169, 229, and 234.

[171] Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing Control with View-based Typestate. In *Formal Techniques for Java-like Programs (FTfJP)*, June 2010. Cited on pages 26 and 39.

[172] Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee View Typestate. Retrieved 8/24/12., July 2012. Cited on page 39.

[173] Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee View Typestate (Extended Version). Retrieved 8/24/12., July 2012. Cited on pages 39 and 40.

[174] Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-Guarantee Protocols. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. Cited on pages 14, 40, 151, and 208.

[175] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. Cited on page 3.

[176] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised Edition*. MIT Press, 1997. Cited on pages 2 and 3.

[177] E. Moggi. Computational Lambda-calculus and Monads. In *IEEE Symposium on Logic in Computer Science (LICS)*, 1989. Cited on page 23.

[178] Eugenio Moggi. Notions of Computation and Monads. *Information and Computation (I&C)*, 93(1):55–92, 1991. Cited on page 212.

[179] Katherine F. Moore and Dan Grossman. High-level Small-step Operational Semantics for Transactions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2008. Cited on page 228.

[180] Peter Müller and Arsenii Rudich. Ownership Transfer in Universe Types. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2007. Cited on pages 21, 32, and 108.

[181] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, volume 8, pages 267–280, 2008. Cited on page 199.

[182] Rasmus E. Møgelberg. A Type Theory for Productive Coprogramming via Guarded Recursion. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2014. Cited on pages 142 and 145.

[183] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A Type System for Borrowing Permissions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2012. Cited on pages 13, 26, 105, and 110.

[184] Hiroshi Nakano. Fixed-Point Logic with the Approximation Modality and Its Kripke Completeness. In *Theoretical Aspects of Computer Software*, pages 165–182, 2001. Cited on pages 142 and 145.

[185] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract Predicates and Mutable ADTs in Hoare Type Theory. In *European Symposium on Programing Languages and Systems (ESOP)*, 2007. Cited on pages 23, 44, 48, 146, 152, and 206.

[186] Aleksandar Nanevski, Paul Govereau, and Greg Morrisett. Towards Type-theoretic Semantics for Transactional Concurrency. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, January 2009. Cited on pages 23, 44, and 206.

[187] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *European Symposium on Programing Languages and Systems (ESOP)*, 2014. Cited on pages 43, 207, and 247.

[188] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and Separation in Hoare Type Theory. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2006. Cited on pages 23, 44, 48, 146, and 152.

[189] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent Types for Imperative Programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2008. Cited on pages 23, 44, 48, 119, 142, 146, 152, 153, and 206.

[190] Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the Verification of Heap-manipulating Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2010. Cited on pages 23, 44, 48, 152, 153, 206, and 247.

[191] Daiva Naudziuniene, Matko Botincan, Dino Distefano, Mike Dodds, Radu Grigore, and Matthew J. Parkinson. jStar-Eclipse: An IDE for Automated Verification of Java Programs. In *Foundations of Software Engineering (FSE)*, 2011. Cited on page 43.

[192] Zhaozhong Ni and Zhong Shao. Certified Assembly Programming with Embedded Code Pointers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006. Cited on page 20.

[193] Leonor Prensa Nieto. The Rely-Guarantee Method in Isabelle/HOL. In *European Symposium on Programing Languages and Systems (ESOP)*, 2003. Cited on pages 13, 34, and 44.

[194] Tobias Nipkow and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer Verlag, 2002. Cited on page 44.

[195] Ligia Nistor and Jonathan Aldrich. Verifying Object-Oriented Code Using Object Propositions. In *International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, 2011. Cited on pages 13, 25, 26, and 39.

[196] Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. Object Propositions. In *Formal Methods (FM)*, 2014. Cited on pages 26 and 39.

[197] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press Oxford, 1990. Cited on page 3.

[198] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained Types for Object-Oriented Languages. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2008. Cited on pages 48 and 152.

[199] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic (CSL)*, August 2001. Cited on pages 12, 17, 50, 56, and 279.

[200] Peter W. O'Hearn. Resources, Concurrency and Local Reasoning. In *Concurrency Theory (CONCUR)*, 2004. Cited on pages 19, 24, 25, 246, and 280.

[201] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying Linearizability with Hindsight. In *Symposium on Principles of Distributed Computing (PODC)*, July 2010. Cited on pages 5, 41, 167, 208, and 230.

[202] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and Information Hiding. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2004. Cited on pages 5, 18, and 280.

[203] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, Uniqueness, and Immutability. In *Objects, Components, Models and Patterns*, 2008. Cited on pages 55 and 109.

[204] Susan Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6(4):319–340, 1976. Cited on pages 7, 16, 34, 38, 114, 162, 207, and 279.

[205] M. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2006. Cited on pages 18 and 57.

[206] Matthew Parkinson and Gavin Bierman. Separation Logic and Abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2005. Cited on pages 18, 26, 37, and 280.

[207] Matthew J. Parkinson and Gavin M. Bierman. Separation Logic, Abstraction and Inheritance. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2008. Cited on page 18.

[208] Matthew J. Parkinson and Alexander J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In *European Symposium on Programing Languages and Systems (ESOP)*, 2011. Cited on page 16.

[209] David Lorge Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM (CACM)*, 15(12):1053–1058, 1972. Cited on page 1.

[210] Christine Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. In *Typed Lambda Calculi and Applications (TLCA)*, 1993. Cited on pages 47 and 132.

[211] Doron Peled. Model Checking Using Automata Theory. In M.Kemal Inan and RobertP. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 55–79. Springer Berlin Heidelberg, 2000. Cited on page 248.

[212] Rasmus Lerchedahl Petersen, Lars Birkedal, Aleksandar Nanevski, and Greg Morrisett. A Realizability Model for Impredicative Hoare Type Theory. In *European Symposium on Programing Languages and Systems (ESOP)*, 2008. Cited on pages 23 and 44.

[213] Frank Pfenning and Christine Paulin-Mohring. Inductively Defined Types in the Calculus of Constructions. In *Mathematical Foundations of Programming Systems (MFPS)*, 1990. Cited on page 43.

[214] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002. Cited on page 278.

[215] Alexandre Pilkiewicz and François Pottier. The Essence of Monotonic State. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, January 2011. Cited on pages 121, 165, 167, and 176.

[216] François Pottier and Jonathan Protzenko. Programming with Permissions in Mezzo. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2013. Cited on pages 22 and 244.

[217] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013. Cited on pages 46 and 47.

[218] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of Reference Immutability. In *European Conference on Object-Oriented Programming (ECOOP)*, July 9–11, 2008. Cited on page 32.

[219] Uday S. Reddy and John C. Reynolds. Syntactic Control of Interference for Separation Logic. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2012. Cited on pages 18, 19, 24, 25, 57, and 207.

[220] Bernhard Reus and Jan Schwinghammer. Separation Logic for Higher-Order Store. In *Computer Science Logic (CSL)*, 2006. Cited on page 20.

[221] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2002. Cited on pages 5, 12, 17, 21, 50, 56, and 279.

[222] John C. Reynolds. Towards a Theory of Type Structure. In *Programming Symposium*, 1974. Cited on pages 2 and 222.

[223] John C. Reynolds. Syntactic Control of Interference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 1978. Cited on pages 1, 18, and 113.

[224] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, 1983. Cited on pages 2 and 3.

[225] Jeffrey Richter. *CLR Via C♯, Second Edition*. Microsoft Press, 2006. Cited on page 101.

[226] Emily Riehl. *Categorical Homotopy Theory*, volume 24. Cambridge University Press, 2014. Cited on page 47.

[227] Patrick Rondon. *Liquid Types*. PhD thesis, University of California, San Diego, 2012. Cited on pages 210, 214, and 221.

[228] Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. CSolve: Verifying C with Liquid Types. In *Computer Aided Verification (CAV)*, 2012. Cited on pages 210, 214, and 221.

[229] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008. Cited on pages 13, 25, 210, 214, and 221.

[230] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2010. Cited on pages 13, 25, 48, 150, 152, 206, 210, 214, and 221.

[231] Andreas Rossberg, Claudio V Russo, and Derek Dreyer. F-ing Modules. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2010. Cited on page 2.

[232] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2009. Cited on page 49.

[233] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested Hoare Triples and Frame Rules for Higher-order Store. In *Computer Science Logic (CSL)*, 2009. Cited on page 20.

[234] Marco Servetto, David J. Pearce, Lindsay Groves, and Alex Potanin. Ballon Types for Safe Parallelisation over Arbitrary Object Graphs. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2013. Cited on page 107.

[235] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Symposium on Principles of Distributed Computing (PODC)*, August 1995. Cited on page 28.

[236] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997. Cited on page 228.

[237] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *ACM SIGPLAN Workshop on Haskell (Haskell)*, 2002. Cited on page 220.

[238] Frederick Smith, David Walker, and J Gregory Morrisett. Alias Types. In *European Symposium on Programing Languages and Systems (ESOP)*, 2000. Cited on pages 21 and 28.

[239] Morten Heine B. Sorensen and Pawel Urzyczyn. *Lectures on the Curry Howard Isomorphism*. Elsevier, 1998. Cited on page 3.

[240] Matthieu Sozeau. Program-ing Finger Trees in Coq. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2007. Cited on pages 120 and 142.

[241] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-Certification: Bootstrapping Certified Typecheckers in F* with Coq. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012. Cited on pages 23, 211, and 239.

[242] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular Reasoning About Separation of Concurrent Data Structures. In *European Symposium on Programing Languages and Systems (ESOP)*. 2013. Cited on pages 7, 37, 165, 167, 184, 207, 208, 244, 245, and 280.

[243] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-dependent Types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2011. Cited on pages 23, 47, 152, 206, 211, and 239.

[244] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):122–144, October 2006. Cited on pages 23 and 245.

[245] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013. Cited on pages 23, 48, 153, 206, 211, and 239.

[246] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic Type, Region, and Effect Inference. *Journal of Functional Programming (JFP)*, 2(2), 1992. Cited on pages 109, 110, and 245.

[247] Ross Tate. The Sequential Semantics of Producer Effect Systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2013. Cited on page 23.

[248] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1999. Cited on pages 3, 47, and 279.

[249] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1994. Cited on pages 12, 21, 22, 28, 109, 110, and 245.

[250] Jesse A. Tov and Riccardo Pucella. A Theory of Substructural Types and Control. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011. Cited on page 105.

[251] R Kent Treiber. Systems Programming: Coping with Parallelism. Technical report, IBM Thomas J. Watson Research Center, 1986. Cited on pages v, 164, 165, 169, 170, 172, 198, 203, and 229.

[252] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding Reference Immutability to Java. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2005. Cited on pages 9, 13, 29, 30, 49, 64, 101, 104, 106, 112, 113, 114, 125, 181, and 190.

[253] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2013. Cited on pages 42, 165, 167, 199, 205, 206, 240, and 244.

[254] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical Relations for Fine-Grained Concurrency. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013. Cited on pages 42, 164, 165, 167, 176, 199, 205, 206, 240, and 244.

[255] Aaron Joseph Turon and Mitchell Wand. A Separation Logic for Refining Concurrent Objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2011. Cited on pages 42, 165, 167, 176, 199, and 205.

[256] Viktor Vafeiadis. *Modular Fine-grained Concurrency Verification*. Phd thesis, University of Cambridge, 2007. Cited on page 41.

[257] Viktor Vafeiadis. RGSep Action Inference. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2010. Cited on page 43.

[258] Viktor Vafeiadis. Concurrent Separation Logic and Operational Semantics. In *Mathematical Foundations of Programming Systems (MFPS)*, 2011. Cited on pages 13, 19, 24, 25, 35, 152, 246, and 280.

[259] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving Correctness of Highly-concurrent Linearisable Objects. In *Principles and Practice of Parallel Programming (PPoPP)*, March 2006. Cited on pages 7, 41, 201, and 205.

[260] Viktor Vafeiadis and Matthew Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *Concurrency Theory (CONCUR)*, 2007. Cited on pages 7, 13, 19, 25, 35, 38, 43, 117, 151, 153, 207, and 280.

[261] Niki Vazou, Patrick Rondon, and Ranjit Jhala. Abstract Refinement Types. In *European Symposium on Programing Languages and Systems (ESOP)*, 2013. Cited on pages 44, 210, 214, and 215.

[262] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with Refinement Types in the Real World. In *Haskell Workshop*, 2014. Cited on pages 44, 210, and 214.

[263] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2014. Cited on pages 44, 210, 214, and 215.

[264] Philip Wadler. The Essence of Functional Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1992. Cited on page 212.

[265] David Walker. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3—-43. 2005. Cited on pages 12, 20, and 21.

[266] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. In *Types in Compilaton (TIC)*, 2000. Cited on pages 21 and 28.

[267] Daniel C Wang and Andrew W Appell. Type-Preserving Garbage Collectors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2001. Cited on page 23.

[268] Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar. Practical Permissions for Race-Free Parallelism. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012. Cited on page 111.

[269] John Wickerson, Mike Dodds, and Matthew Parkinson. Explicit Stabilisation for Modular Rely-Guarantee Reasoning. In *European Symposium on Programing Languages and Systems (ESOP)*, 2010. Cited on pages 7, 14, 37, 39, 117, 151, and 207.

[270] Andrew K Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation (I&C)*, 115(1):38–94, 1994. Cited on page 3.

[271] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2003. Cited on page 122.

[272] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, May 1998. Cited on pages 16 and 48.

[273] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999. Cited on pages 16, 48, 152, 164, and 206.

[274] Hongseok Yang. *Local Reasoning for Stateful Programs.* Phd thesis, University of Illinois, 2001. Cited on page 19.

[275] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and Reference Immutability Using Java Generics. In *Foundations of Software Engineering (FSE)*, September 2007. Cited on pages 9, 13, 29, 31, 43, 49, 50, 56, 64, 86, 88, 93, 104, 106, 107, 108, 112, 113, 114, 125, 181, and 190.

[276] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *ACM International Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2010. Cited on pages 9, 13, 29, 31, 43, 49, 64, 104, 106, 108, 112, 113, 114, 125, 181, and 190.

Appendix A

BACKGROUND READING

One unfortunate aspect of our field is that we tend to write technical papers assuming the reader has some pre-existing level of expertise. Stepping into full verification, this problem is exacerbated; the various verification communities have their own additional assumptions on background. Over the course of my studies, I have found that this made it difficult to start from a technical development I wished to understand, and map backwards to a route for acquiring the background knowledge required to comprehend it.

To that end, this section gives short reading lists that should get a motivated reader up to speed on various areas I build upon in this thesis, assuming completion of a graduate level introduction to programming languages (e.g., working through most of Pierce's *Types and Programming Languages* [214].

A.1 Dependent Types

To get up to speed on dependent type theory, I recommend reading, in roughly the following order:

- *An Intuitionistic Theory of Types* [165]. This early paper of Per Martin-Löf is a concise overview of one variant of what has come to be called MLTT. It covers the type rules of the system, uses for proving, strong normalization (which implies logical consistency), and more. It also contains an extended example of how adding a type of all types (also referred to as $\mathsf{Type} : \mathsf{Type}$) allows inconsistent derivations. If you only read one paper on dependent types ever, read this one.

- *A Framework for Defining Logics* [114, 113]. These papers describe LF. The journal version [113] contains detailed, very clear proofs of key results, including a very interesting proof by type-preserving translation that strong normalization of the simply

typed lambda calculus implies strong normalization for LF.

- *Type Theory and Functional Programming* [248]. This out of print but freely electronically available book covers the metatheory of Martin-Löf type theory, including data types and eliminators, and more, in great detail. The book is also very well written.

- *The Calculus of Constructions* [62] is the canonical reference for CC, the core theory underlying Coq, and the canonical example of an impredicative dependent type theory.

- *Metamathematical Investigations of a Calculus of Constructions* [60].

- *Lambda Calculi with Types* [14] is a thorough coverage of all pure type systems, which includes the simply typed lambda calculus, System F, System F_ω, and CC, all covered in the same style. This book is an excellent reference for comparing points of difference between different classic type systems.

A.2 Program Logics: Hoare Logic, Separation Logic, and Beyond

The following list of recommended readings is based heavily on a special topics course run by Viktor Vafeiadis and Derek Dreyer[1], and formed the bulk of my introduction to the subject. The credit for this list belongs mostly to them:

- *An Axiomatic Basis for Computer Programming* [126].

- *An Axiomatic Proof Technique for Parallel Programs I* [204].

- *Tentative Steps Toward a Development Method for Interfering Programs* [139].

- *Separation Logic: A Logic for Shared Mutable Data Structures* [221].

- *Local Reasoning about Programs that Alter Data Structures* [199].

[1] https://wiki.mpi-sws.org/star/cpl

- *Resources, Concurrency, and Local Reasoning* [200].
- *A Semantics for Concurrent Separation Logic* [43].
- *Concurrent Separation Logic and Operational Semantics* [258].
- *Checking Interference with Fractional Permissions* [41].
- *Permission Accounting in Separation Logic* [36].
- *Separation and Information Hiding* [202].
- *Separation Logic and Abstraction* [206].
- *A Marriage of Rely/Guarantee and Separation Logic* [260].
- *Local Rely-Guarantee Reasoning* [82].
- *Deny-Guarantee Reasoning* [73].
- *Concurrent Abstract Predicates* [71].
- *Views: Compositional Reasoning for Concurrent Programs* [70].
- *Impredicative Concurrent Abstract Predicates* [242].

Finally, Appel's recent book *Program Logics for Certified Compilers* [9] provides and excellent, in depth presentation of higher-order separation logics for higher order stores, with a heavy focus on the VST [8] toolchain which verifies C programs with respect to CompCert's operational semantics [154] (the cost of this is that a reader should be very familiar with Coq to fully appreciate the book, a skill that can be acquired from Chlipala's recent text [51]).

VITA

Colin Stebbins Gordon grew up in Ridgefield, CT, and spent varying amounts of time in Harbor Springs, MI with family; in Providence, RI at Brown University (ScB 2004); and in Sunnyvale and San Francisco, CA (at Network Appliance and Sun Microsystems) on the way to a job at Microsoft in Redmond, WA and later graduate school at the University of Washington (MS 2011, PhD 2014) in Seattle, WA.

He has a wife, Natalie, and son (his greatest accomplishment), Henry.